Automated Heuristic Generation By Intelligent Search

Andrew W. Burnett

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy

March 2021

Abstract

This thesis presents research that examines the effectiveness of several different program synthesis techniques when used to automate the creation of heuristics for a local search-based Boolean Satisfiability solver.

Previous research focused on the automated creation of heuristics has almost exclusively relied on evolutionary computation techniques such as genetic programming to achieve its goal. In wider program synthesis research, there are many other techniques which can automate the creation of programs. However, little effort has been expended on utilising these alternate techniques in automated heuristic creation.

In this thesis we analyse how three different program synthesis techniques perform when used to automatically create heuristics for our problem domain. These are genetic programming, exhaustive enumeration and a new technique called local search program synthesis. We show how genetic programming can create effective heuristics for our domain. By generating millions of heuristics, we demonstrate how exhaustive enumeration can create small, easily understandable and effective heuristics. Through an analysis of the memoized results from the exhaustive enumeration experiments, we then describe local search program synthesis, a program synthesis technique based on the minimum tree edit distance metric. Using the memoized results, we simulate local search program synthesis on our domain, and present evidence that suggests it is a viable technique for automatically creating heuristics.

We then define the necessary algorithms required to use local search program synthesis without any reliance on memoized data. Through experimentation, we show how local search program synthesis can be used to create effective heuristics for our domain. We then identify examples of heuristics created that are of higher quality than those produced from other program synthesis methods. At certain points in this thesis, we perform a more detailed analysis on some of the heuristics created. Through this analysis, we show that, on certain problem instances, several of the heuristics have better performance than some state-of-the-art, hand-crafted heuristics.

Acknowledgements

My deepest and sincere thanks go to my supervisors, Dr. Andrew Parkes and Professor Dario Landa-Silva, for their time, energy, and invaluable insight during the time it has taken me to complete my thesis. I am particularly thankful to Dr. Parkes, whose discussions have allowed me to focus my efforts, as well as helping to provide the necessary motivation required to complete this body of research.

I would also like to thank those within the University who have helped to keep me engaged, driven, and focused on the task at hand. Particular thanks go to Vlad, Alexandra, Jason and Liam.

Finally I would like to extend my deepest thanks towards my family and friends, who, through their continued belief and council, have provided me with the necessary tools to overcome the challenges encountered while completing this thesis. Particular thanks go to my parents, Branwell, Natalie, Michael, Becky and James. Finally I would like to thank Tilly, whose continued curiosity has provided me with a unique perspective on my work.

Contents

List of Figures **viii**

List of Tables **xiv**

List of Algorithms **xvii**

1 Introduction **1**

1.1 Structure of this Thesis 6

1.2 Academic Publications Produced 7

2 Literature Review **8**

2.1 Introduction 8

2.2 Boolean Satisfiability Problem 9

2.2.1 Algorithms to Solve SAT 12

2.2.2 SAT Problem Hardness 18

2.2.3 Summary & Discussion 21

2.3 LS-SAT Heuristics 23

2.3.1 GSAT & Variants 28

2.3.2 WalkSAT & Variants 31

2.3.3 Adaptive Heuristics 39

2.3.4 Clause Weighting Schemes 41

2.3.5 Probability Distribution Heuristics 46

2.3.6 Configuration Checking 49

2.3.7 Summary & Discussion 53

2.4 Automated Creation of Heuristics 54

2.4.1 Automated Creation of Heuristics for Hard Problems 55

2.4.2 Automated Design of LS-SAT Heuristics 58

2.4.3 Summary & Discussion 61

2.5 Program Synthesis . . . 63
2.5.1 Preliminaries . . . 64
2.5.2 Tree-based Program Representation . . . 69
2.5.3 Direct Search Techniques . . . 74
2.5.4 Genetic Programming . . . 79
2.5.5 Summary & Discussion . . . 84
2.6 Minimum Tree Edit Distance Problem . . . 86
2.6.1 Definition . . . 86
2.6.2 Algorithm . . . 88
2.6.3 Summary & Discussion . . . 95
2.7 Conclusions . . . 95

3 Heuristic Representation & Evaluation 97
3.1 Introduction . . . 97
3.2 Heuristic Representation . . . 98
3.2.1 Language Details . . . 105
3.2.2 Example Heuristics . . . 111
3.3 Running a Heuristic on a Problem Instance . . . 112
3.3.1 Deduction of a Heuristic's Requirements . . . 115
3.3.2 Evaluating the Heuristic Function . . . 119
3.3.3 Overarching Local Search Algorithm . . . 120
3.4 Evaluating a Heuristic's Performance . . . 126
3.4.1 Fitness Function . . . 126
3.4.2 Fitness of Known Heuristics . . . 130
3.4.3 Testing Set . . . 131
3.5 Discussions & Conclusions . . . 134

4 Exhaustive Enumeration & GP 136
4.1 Introduction . . . 136
4.2 Exhaustive Enumeration Experiments . . . 139
4.2.1 Search Space Size . . . 139
4.2.2 Methodology . . . 140
4.2.3 Results in Generated Order . . . 142
4.2.4 Fitness Results . . . 151
4.2.5 Timing Results . . . 157

4.2.6 Individual Results 159
4.3 Genetic Programming Experiments 167
4.3.1 Methodology 167
4.3.2 Results 168
4.3.3 Individual Results 174
4.4 Discussions & Conclusions 177

5 Analysing Heuristics Using the MTED 180
5.1 Introduction 180
5.2 Initial Observations 181
5.3 Analysing a Heuristic's Neighbourhood 187
5.3.1 Percentage of Fitter Neighbours 188
5.3.2 Size of Neighbourhood 191
5.3.3 Finding a Fitter Neighbour 191
5.4 Simulated Local Search Experiments 196
5.4.1 Methodology 198
5.4.2 Results 199
5.5 Discussions & Conclusions 208

6 Neighbourhood Generation 211
6.1 Introduction 211
6.2 Function Signature 213
6.2.1 Mock Language 215
6.3 Example Neighbourhood Generation Algorithms 218
6.3.1 Notation & Example Output 218
6.3.2 Naive Method 221
6.3.3 Identifying Common Patterns 224
6.4 Defining Generate Successors 227
6.4.1 Formalising Patterns 228
6.4.2 Abstracting Patterns 230
6.4.3 Size of TPPTs 235
6.4.4 Finding TPPTs 236
6.4.5 Pre-processing 243
6.4.6 Applying Edit Sequences 244
6.4.7 Compound Moves & Final Algorithms 252

6.5 Generate Successors Randomly 257
6.6 Searching for Edit Sequences 261
6.6.1 Finding all TPPTs 261
6.6.2 Searching TPPTs: Intuition 265
6.6.3 Searching TPPTs: Examples 272
6.6.4 Generating Edit Sequences 279
6.6.5 Testing 285
6.7 Discussions & Conclusions 287

7 Local Search Program Synthesis 290

7.1 Introduction 290
7.2 Initial Experiments 291
7.2.1 Bounded Experiments 291
7.2.2 Unbounded Experiments 297
7.2.3 Summary 303
7.3 Randomised Neighbourhood Generation 303
7.3.1 Observations 304
7.3.2 Using Randomised Neighbourhood Generation 308
7.3.3 Randomised Neighbourhood Generation Experiment 314
7.3.4 Summary 318
7.4 Using an Alternate Cost Function 319
7.4.1 Observations 319
7.4.2 Alternate Cost Function Experiment 326
7.4.3 Summary 331
7.5 Using an Alternate Language 332
7.5.1 Language B 332
7.5.2 GP Language B Experiment 337
7.5.3 LSPS Language B Experiment 341
7.5.4 Summary 347
7.6 Examples of Created Heuristics 348
7.6.1 Heuristics Created Using Language A 348
7.6.2 Heuristics Created Using Language B 353
7.6.3 Testing Set Results 359
7.6.4 Summary 364
7.7 Discussions & Conclusions 364

8 Conclusions **368**

8.1 Context 368

8.2 Summary 370

8.2.1 Chapter 3 370

8.2.2 Chapter 4 370

8.2.3 Chapter 5 - 7 371

8.3 Extensions & Future Work 372

8.3.1 Improved Fitness Function 372

8.3.2 Language 373

8.3.3 Improving the LS-SAT Solver 374

8.3.4 Using Other Program Synthesis Methods 375

8.4 Final Remarks 377

Bibliography **378**

A Exhaustive Enumeration Results **398**

B Genetic Programming Created Heuristics **407**

List of Figures

2.1 Two examples of propositional logic formula; one in CNF, one not. . . 10
2.2 Two examples of assignments to a SAT problem instance containing 8 variables. 18
2.3 Results from solving 3-SAT problems using the mapleSAT solver. . 20
2.4 An overview of the control flow of a system designed to automate the creation of heuristics. 55
2.5 The language used by Fukunaga [60, 61, 63] to automatically create LS-SAT heuristics. 59
2.6 Example of a program specification written in predicate logic. 65
2.7 The Language EX-1. 71
2.8 Two examples of program trees written using Language EX-1, shown in Figure 2.7. 72
2.9 An example of a search tree for the set of programs in Language EX-1, shown in Figure 2.7. 75
2.10 An example of the crossover operator being applied to two trees written in Language EX-1. 83
2.11 An example of the mutation operator being applied to a program tree written in Language EX-1. 84
2.12 Examples of tree edits between trees. 87
2.13 A recursive solution to the MTED problem, designed to operate on forests. 88
2.14 Two trees that are used in Section 2.6.2 to illustrate how the MTED algorithm works. 90

3.1 The set of principle types in the DSL used in this thesis to create LS-SAT heuristics. 103

3.2 Eight examples of previously described, hand-crafted heuristics formulated using the DSL described in Tables 3.1 and 3.2. 113
3.3 The WalkSAT heuristic, shown in Figure 3.2c, visualised as a program tree. 114
3.4 The internal-form of the heuristic gNovelty+, shown in Figure 3.2f. 121
3.5 Examples of two arbitrary N-True-Vars and N-True-Sets data structures at a point in an LS-SAT algorithm. 124
3.6 Examples of two arbitrary Var-Pos and Var-Pos-Pos data structures at a point in an LS-SAT algorithm. 127

4.1 The number of heuristics of a specific size in Languages A and A1. . . 141
4.2 Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 10. 144
4.3 Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 17. 145
4.4 Results from the exhaustive enumeration experiments, showing the fitness values for a subset of heuristics in Language A of size 14. . . . 147
4.5 Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 13. Each heuristic has been coloured according to its first term. 147
4.6 Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 13. We colour each heuristic according to its leading n terms. 148
4.7 Results from the exhaustive enumeration experiments, showing heuristics of size 14 in Languages A and A1. 150
4.8 Three examples of heuristics of size 10 in Language A that return the same fitness value due to the formulation of the language. 151
4.9 All heuristics in Languages A and A1 of size ≤ 14. 153
4.10 All heuristics in Languages A and A1, showing the fitness of each heuristic plotted against that heuristic's fitness variance. 158
4.11 All heuristics in Languages A and A1 with fitness > 10, showing the fitness of each heuristic plotted against that heuristic's nanoseconds-per-flip value. 160
4.12 Six heuristics that reported a high fitness value from the enumeration of Language A. 161

4.13 Six heuristics that reported a high fitness value from the enumeration of Language A1. 162
4.14 Fitness data from the best repetitions of the GP experiments performed using Languages A and A1. 171
4.15 Size data from the best repetitions of the GP experiments performed using Languages A and A1. 172

5.1 Four heuristics from the enumeration of Language A. 183
5.2 The distribution of the fitness values in four heuristic's MTED(n) sets. 184
5.3 The percentage of heuristics in neighbourhoods described by $N(n)$, where $n \in \{1 \ldots 4\}$. 189
5.4 The number of evaluations required to have a 50% chance of finding a neighbour fitter than the candidate heuristic in neighbourhoods described by $N(n)$, for $n \in \{1 \ldots 4\}$. 194
5.5 Results from the simulated local search experiments performed on various subsets of heuristics in Language A. 200
5.6 The number of evaluations performed in each simulated repetition of local search using heuristics in Language A of size ≤ 15. 204
5.7 Results from the simulated local search experiments performed on various subsets of heuristics in Language A, with duplicate results removed. 205
5.8 Graph showing the fitness distance data for the 310 unique heuristics found from the $1,000$ runs of the local search experiment described by the triple (Local-Search-Rnd, 3, 15). 209
5.9 Matrix showing the MTED between the fittest 20 unique heuristics returned from the $1,000$ runs of the local search experiment described by the triple (Local-Search-Rnd, 3, 15). 210

6.1 The Language EX-1. 216
6.2 An example program tree under Language EX-1 that type checks. . . 217
6.3 A pair of program trees written in Language EX-1. Together with the program tree in Figure 6.2, they can be viewed as a sequence. 219
6.4 A term-based edit sequence. It can be used to transform the program tree in Figure 6.2 into the program tree in Figure 6.3b via the program tree in Figure 6.3a. 221

6.5 An example of a program tree and a term-based edit sequence. 225
6.6 Three trees, each an abstract representation of a part of a program tree written in Language EX-1. They can be viewed as a sequence. Each tree is obtained by inserting a node into the previous tree. . . . 226
6.7 A start state PTPPT, end state PTPPT and edit sequence. The edit sequence can be used to transform the start state into the end state. . 229
6.8 A program tree written in Language EX-1. When the function GenerateSuccessors is given the program tree in Figure 6.2 and an n value of at least 2, this program tree would be an example of output returned. 230
6.9 Three PTPPTs that can be constructed by applying the edit sequences in Figure 6.10 to the PTPPT in Figure 6.6a. 231
6.10 Three term-based edit sequences that can be used to transform the PTPPT in Figure 6.6a into the PTPPTs shown in Figure 6.9. 232
6.11 A start state TPPT, end state TPPT and edit sequence to transform the start state into the end state. 234
6.12 The set of results stored in the *nextLevel* variable in the Find-RTPPTs algorithm when given a context set at the root of the program tree in Figure 6.2, and an n value of 2. 239
6.13 The set of TPPTs returned from the Find-RTPPTs algorithm when given a context set at the root of the program tree in Figure 6.2, and an n value of 2. 241
6.14 A program tree written in Language EX-1. When the function GenerateSuccessors is given the program tree in Figure 6.2 and an n value of at least 1, this program tree would be an example of output returned. 243
6.15 A start state and end state TPPT. The start state is representative of a pattern of nodes above the root in Figure 6.2. 244
6.16 The program tree shown in Figure 6.2 after it has been pre-processed. 245
6.17 Language EX-1 in its type-compressed form. 245
6.18 A start state PTPPT, end state PTPPT and edit sequence to transform the start state into the end state. 248
6.19 A start state TPPT, end state TPPT and edit sequence to transform the start state into the end state. 249

6.20 The set of term-based edit sequences that can be extracted from the type-based edit sequence shown in Figure 6.19c. 250
6.21 A type-based edit sequence. It has been changed when compared to the edit sequence shown in Figure 6.19c, to ensure that incorrect term-based edit sequences are not created from it. 251
6.22 Three program trees written in Language EX-1. When the function GenerateSuccessors is given the program tree in Figure 6.2 and an n value of at least 3, these program trees would be examples of output returned. 254
6.23 Diagram showing two ways in which edit sequences can be chained together to move via intermediary trees to transform the program tree in Figure 6.2 into the program tree in Figure 6.22c. 256
6.24 The set of unique vectors that are used as the root nodes when generating every possible TPPT for Language EX-1. 262
6.25 The set of results stored in the *nextLevel* variable in the Create-RTPPTs algorithm when given the $vector = [\texttt{Int}]$ and an n value of 2. For each $n_i \in \{0 \ldots n\}$, a set of results are shown for each element in *vector*. 266
6.26 A start state program tree and an edit sequence. When the edit sequence is applied to the program tree in Figure 6.26a, the same program tree is produced. 272
6.27 A TPPT. This TPPT type checks. 274
6.28 The six possible successor states created by inserting an unlabelled node into the TPPT shown in Figure 6.27. 275
6.29 Two TPPTs created by inserting a node into the TPPT in Figure 6.27. 276
6.30 A start state TPPT and a valid successor state TPPT created by relabelling a node. 278
6.31 A start state TPPT and a valid successor state TPPT created by deleting a node. 279

7.1 Results from Experiments A1 and A2. 293
7.2 Results from Experiments A3 and A4. 299
7.3 Final results from Experiment A4. The data points are coloured according to the size of the initial heuristic of that repetition. 301

7.4 Size of the neighbourhoods of the candidate heuristics from all repetitions of Experiments A3 and A4. 305
7.5 The fitness values of the final heuristics created from the simulations of Experiment A4. 311
7.6 The average number of neighbours in each N_k neighbourhood for all candidate heuristics from Experiments A3 and A4. 313
7.7 Results from Experiment A5. 315
7.8 A start state PTPPT and two end state PTPPTs written using Language A. 322
7.9 Results from Experiment A6. 329
7.10 Results from the 5th GP repetition performed using Language B. . . . 340
7.11 Results from Experiments B1 and B2. 343
7.12 Ten heuristics that reported a high fitness value from Experiments A3, A4, A5 and A6. 350
7.13 Four heuristics that reported a high fitness value from Experiments B1 and B2. 356

A.1 Results from the exhaustive enumeration experiment performed on Language A, as detailed in Chapter 4. 399
A.2 Results from the exhaustive enumeration experiment performed on Language A1, as detailed in Chapter 4. 403

B.1 The GP-A-5 heuristic. 408
B.2 The GP-A1-3 heuristic. 415
B.3 The GP-B-5 heuristic. 422

List of Tables

2.1 Detailed results of the 2018 SAT Competition, Random track. 22
2.2 The set of functions used in the pseudocode in Section 2.3. 27
2.3 The set of functions used in the heuristics in Section 2.3.5. 47
2.4 MTED subproblems calculated when computing the tree edit distance between the trees et_1 and et_2 shown in Figure 2.14. 93
2.5 Tree distance table used when calculating the MTED between the trees et_1 and et_2 shown in Figure 2.14. 94

3.1 The set of functions in the DSL used in this thesis to create LS-SAT heuristics. 99
3.2 The set of terminals in the DSL used in this thesis. 104
3.3 All of the gain type metrics defined in the DSL, described in terms of Definitions 21 and 22. 108
3.4 The set of member variables in the Data-Required structure, together with an explanation of their meaning. 115
3.5 The instantiation of the Data-Required structure for the heuristic gNovelty+. 119
3.6 The set of problems used in the fitness function, which are broken up into five subsets of problem instances. 129
3.7 The fitness values of the heuristics shown in Figure 3.2 according to Equations (3.6) and (3.7). 130
3.8 The set of problems used in the testing set, which are broken up into eleven subsets of problem instances. 132
3.9 Results from running the heuristics in Figure 3.2 on the testing set. . 133

4.1 Languages A and A1. 137
4.2 The number of heuristics of a specific size in Languages A and A1. . . 140

4.3 The set of heuristics of a small size in Languages A and A1. 143
4.4 The fitness distribution of heuristics in Language A of size ≤ 10 and 11 - 17. 154
4.5 The fitness distribution of heuristics in Language A1 not in Language A of size ≤ 10 and 11 - 15. 156
4.6 Results from running the heuristics in Figures 4.12 and 4.13 on the testing set. 164
4.7 The parameters used in the GP experiments in Section 4.3. 167
4.8 Statistical data pertaining to the GP experiments performed using Languages A and A1. 169
4.9 Statistical data concerning the frequency of terms used in the fittest heuristic returned from each repetition of the GP experiments performed using Languages A and A1. 173
4.10 Results from running the heuristics created from GP using Languages A and A1 on the testing set. 175

5.1 The size of the MTED(n) sets for the heuristics shown in Figure 5.1. 182
5.2 Statistical data pertaining to the neighbourhood size of different sized heuristics. 192

6.1 The Language TE-1. 220
6.2 All the collections of sequences of RTPPTs created from Find-RT-PPTs when given a context set at the root of the program tree in Figure 6.2, and an n value of 2. 240
6.3 Data gathered when calling Create-All-Edit-Sequences on different languages and n_{max} values. 286

7.1 Statistical data from Experiments A1 and A2. 294
7.2 Statistical data from Experiments A3 and A4. 300
7.3 Data from simulated re-runs of Experiment A4 using the termination mechanism described in Section 7.3.1. 310
7.4 Statistical data from Experiment A5. 316
7.5 Data concerning the frequency of different types of edit sequences in $N(3)$. 321
7.6 Data concerning the neighbourhood N_{alt} for Language A. 327
7.7 Statistical data from Experiment A6. 328

7.8 Language B. 333
7.9 Data gathered when calling CREATE-ALL-EDIT-SEQUENCES on Language B using the neighbourhoods $N(3)$ and N_{alt}. 336
7.10 Data concerning the frequency of different types of edit sequences in $N(3)$ and N_{alt} for Language B. 336
7.11 Statistical data pertaining to the GP experiments ran using Language B. 339
7.12 Statistical data from Experiments B1 and B2. 344
7.13 Statistical data concerning the frequency of terms used in the fittest heuristic returned from each repetition of the GP experiments performed using Language B. 354
7.14 Results from running the heuristics in Figures 7.12 and 7.13, and those created from the GP experiment using Language B on the testing set. 360

List of Algorithms

2.1 DPLL Algorithm 16
2.2 Local-Search Algorithm for SAT 17
2.3 GSAT Heuristic 28
2.4 HSAT Heuristic 30
2.5 GWSAT Heuristic 31
2.6 WalkSAT Heuristic 33
2.7 Novelty Heuristic 35
2.8 Novelty+ Heuristic 37
2.9 G^2WSAT Heuristic 38
2.10 GSAT+Weights Heuristic & Weight Update Function 42
2.11 SAPS Weight Update Function 44
2.12 PAWS Weight Update Function 45
2.13 gNovelty+ Weight Update Function 46
2.14 ProbSAT Heuristic 48
2.15 SW_{cc} Heuristic 51
2.16 Type-Check Algorithm 73
2.17 Top-Down-Search Algorithm 76
2.18 Bottom-Up-Search Algorithm 78
2.19 Genetic-Programming Algorithm 80
2.20 MTED Algorithm 90
2.21 Tree-Dist Algorithm 91
3.1 Detailed Local-Search Algorithm for SAT 120
3.2 LS-SAT Update-Data Function 128
5.1 Memoize-Neighbourhoods Algorithm 187
5.2 Local-Search-Greedy Algorithm 197
5.3 Local-Search-Rnd Algorithm 198
6.1 Neighbourhood-Generation Object Prototype 215

6.2 Find-RTPPTs Algorithm 237
6.3 Find-TPPTs Algorithm 242
6.4 Create-Output-Trees Algorithm 247
6.5 Process-Relabel Algorithm 251
6.6 Build-Trees Algorithm 252
6.7 GenerateSuccessors Algorithm 258
6.8 GenerateSuccessors-Rnd Algorithm 260
6.9 Create-All-Roots Algorithm 263
6.10 Create-RTPPTs Algorithm 264
6.11 Generate-TPPTs Algorithm 270
6.12 Create-All-Edit-Sequences Algorithm 280
6.13 Create-Edit-Sequences Algorithm 281
6.14 Create-Insertions Algorithm 282
6.15 Create-Relabels Algorithm 283
6.16 Create-Deletions Algorithm 284

Chapter 1

Introduction

Combinatorial problems require finding groupings, orderings or assignments of discrete objects that satisfy some conditions or constraints [83, Chapter 1]. They are most commonly defined as either decision problems or optimisation problems. A decision problem is formulated as a set of criteria, and the goal when solving such a problem is to find a satisfying solution - that is, a solution that satisfies all the criteria. An optimisation problem is defined in terms of an associated objective function, which assigns each solution a numerical value. Through these values, an ordering of solutions can be defined. The goal when solving an optimisation problem is to find the optimal solution - that is, the solution with the best value according to the objective function. For many combinatorial problems the fastest-known algorithms that are guaranteed to solve them have a time complexity that scales exponentially with the problem size. These complete algorithms, so called because they are guaranteed to definitively and completely solve the problem, are often impractical to use for large problem instances due to this worst-case running time. Combinatorial problems with this property are called hard combinatorial problems, due to the difficulty in solving them. Examples of hard combinatorial problems include the Boolean Satisfiability problem (a decision problem), the Travelling Salesman problem and the Knapsack problem (both optimisation problems) [93].

Incomplete algorithms are an alternate technique used to solve hard combinatorial problems. They provide no guarantee that they will solve the problem - that is to say, they may not find the optimal solution to an optimisation problem, or the satisfying solution to a decision problem. However, incomplete algorithms are able to navigate the search space in such a way that enables "good" solutions to be found quickly; in optimisation problems this can manifest itself as finding a near-optimal

solution, and in decision problems finding a satisfying solution more quickly than a complete algorithm would. Incomplete algorithms can be a viable method to use in domains where either an optimal solution is not a necessity, or where computational restrictions preclude the use of a complete algorithm.

One such incomplete algorithm, and the core basis for much of the work in this thesis, is local search. Local search is a generic algorithm used to solve hard combinatorial problems, and works by "start[ing] at some location in the search space and subsequently move[ing] from the present location to a neighbouring location in the search space" [83, Chapter 1]. Furthermore, unlike complete algorithms, "local search can visit the same location within the search space more than once". Through intelligently selected sequences of movements through the search space, local search can quickly arrive at high-quality solutions much more quickly than complete methods. Well-known variants of local search include simulated annealing [65, Chapter 1], tabu search [70, 65, Chapter 2] and hill-climbing [163].

In many incomplete algorithms (including local search), as well as complete algorithms, a heuristic function can be used to guide the overarching search process. Specifically, when a point in the search is arrived at where the overarching algorithm has no bias on how to progress the search, a heuristic function can be used to make the choice on how to proceed. They are typically simple, computationally inexpensive functions. For example, in a local search algorithm employed on an optimisation problem, a heuristic may be designed to move to the neighbour with the best score according to the objective function. By augmenting a search algorithm with a well-designed heuristic, it is possible to vastly improve the quality of solutions found, either by providing a better quality solution, or by arriving at good solutions more quickly. This can make the use of an effective heuristic an important component in designing well-performing algorithms to solve hard combinatorial problems.

While many complete and incomplete algorithms can be viewed as generic algorithmic frameworks, heuristic functions are almost exclusively problem-dependent; a heuristic that works well in one problem domain will usually not work well in another. In many cases, it is simply not possible to transform one heuristic described in terms of one problem domain into equivalent terms of another. Some heuristics which are designed for a specific problem domain are only effective on certain subclasses of that problem, and perform poorly on other subclasses.

The development of an effective heuristic can be a time consuming part of the

overarching algorithm design process, and usually requires domain-specific knowledge. A user may have to review the literature regarding previously described heuristics in a particular problem domain to find appropriate candidate heuristics, then test them to ascertain whether they provide the desired performance, and potentially refine them if they do not. These issues can be exacerbated for problems that have little or no research concerning the design of effective heuristic strategies. In these situations, additional work is required to evaluate the problem. It may be the case that computationally expensive experiments must be performed in order to determine what an effective heuristic strategy is.

To expedite the heuristic design process, there has been active research in the automation of creating, analysing and the selection of effective heuristics. For example, Fukunaga [60, 63, 61] presented a series of papers detailing systems that automate the creation of heuristics. These heuristics were used as part of a local search algorithm to solve the Boolean Satisfiability problem. Another example is research by Burke et al. [24], where the authors described a system that can automate the selection of heuristics for the timetabling problem. Historically, these efforts to automate the heuristic design process have been fragmented, however recently much of this work has been categorised under the term hyper heuristics. Hyper heuristics are "a set of approaches that are motivated by the goal of automating the design of heuristic methods to solve hard computational problems" [28]. Most hyper heuristics can be classified as one of two types; selective or generative. Selective hyper heuristics are those systems that aim to choose the most effective heuristic from a known set, while generative hyper heuristics are those systems that automatically create new heuristics. In general, there has been comparatively less research undertaken in generative hyper heuristics when compared to selective hyper heuristics.

The core research area of this thesis is in algorithms that automate the creation of heuristics. The heuristics created are used as part of a local search algorithm to solve the Boolean Satisfiability problem. The heuristic creation techniques used in this thesis can be classified as generative hyper heuristics.

The workload involved in designing a system to automate the creation of heuristics is much greater than that of designing a single heuristic. When designing a single heuristic, the heuristic is identified, tested, potentially refined, and then deployed. To automate the heuristic design process, a system needs to be designed that can represent heuristics, and automatically run heuristics against problem instances to

gauge their effectiveness. In addition to this, an overarching algorithm is needed that can create new heuristics. The first two of these components are nearly always domain-specific, but there are general, problem-independent techniques that have previously been used to automatically create heuristics.

Despite the additional work that is required in designing a system that automates the creation of heuristics, such systems have the potential to offer several advantages compared to the manual design process of constructing a single heuristic. For example, such a system could be used to offer tailor-made heuristics for specific problem instances or subsets of problem instances. Within the aviation industry, software is used to create schedules of aircraft departures and arrivals for airports. A schedule's effectiveness is judged by how closely it mirrors the desired departure and arrival times, while maintaining safety requirements. Every airport has a specific number of runways and terminals, with each of these having its own capacity. One can conceive of a general-purpose search algorithm that is designed to find the best scheduling of aircraft according to the given requirements. Such an algorithm may use a heuristic to direct its internal search mechanism. Automated heuristic creation software could be used to design bespoke heuristics for specific airports, with the aim of outperforming general-purpose heuristics.

When using automated heuristic creation systems on well-researched problems, there is also the potential to discover effective heuristics that have previously not been described. Such systems could also be used to test new ideas in formulating heuristics, to ascertain whether they are effective when combined with other, previously known heuristic strategies - in essence becoming a tool to aid in the rapid prototyping of heuristic design. On problems that have little or no previous literature regarding effective heuristic design, such an automated system could significantly reduce the time taken to find effective heuristics in these domains.

As stated, the heuristic creation algorithm is one component of an automated heuristic creation system that is not problem-dependent. Yet, the choice of heuristic creation algorithm largely depends on the choice of representation used for the heuristic. One popular representation is that of a tree-like data structure. This representation is designed to mimic a programming language. In the literature concerning algorithms designed to automatically create heuristics which are represented as tree-like structures, genetic programming and other closely related evolutionary algorithms have been predominantly used in previous research [16, 52, 27].

Genetic programming is a program synthesis technique based on natural selection that is used to evolve programs. In the context of this thesis, the "programs" are the heuristics created. This is not the only program synthesis technique used within wider computer science. Program synthesis is a fragmented discipline, with various subdisciplines having independently developed techniques to automatically create programs. For example, in artificial intelligence deductive programming has been used to create recursive programs [120], and in the functional programming community exhaustive enumeration has proved to be an effective strategy for creating data structures from input-output examples [22].

The core research question we ask in this thesis is what, if any, alternate program synthesis techniques are there that could be used in the automated creation of heuristics. As previously discussed, we will be testing these techniques on the Boolean Satisfiability problem. One of the primary reasons for choosing this domain is that there are several examples of previous work where the overarching goal was to automatically create heuristics for it [60, 63, 61, 9]. These examples of previous work provide us with a clear methodology in creating heuristics for this domain, and allow us to compare the heuristics created from our systems to those created from previous research. There are also many examples of hand-crafted, highly effective heuristics for this domain, and one of our goals in this work is to ascertain how effective our automatically created heuristics are compared to hand-crafted ones.

One of the program synthesis techniques we will use to create heuristics is exhaustive enumeration. We will then use the created heuristics to perform a large-scale search space analysis, utilising the minimum tree edit distance to compare heuristics to each other. To our knowledge, such research has not been undertaken before. From the observations made through this analysis, we will propose a new, novel program synthesis method called local search program synthesis. We will then perform experiments using this method and show that it can be used to create high-quality heuristics for our domain.

To summarise, this thesis will investigate the applicability several program synthesis techniques have in automating the creation of heuristics for use in solving the Boolean Satisfiability problem. Particular attention will be given to techniques that have previously not been used in the automated creation of heuristics.

1.1 Structure of this Thesis

The structure of this thesis is as follows:

- Chapter 2: Literature Review. This chapter provides an overview of the pertinent research to this thesis. Succinctly, this is research concerning the Boolean Satisfiability problem, heuristics for solving the Boolean Satisfiability problem through local search, the automated creation of heuristics, program synthesis methods and the minimum tree edit distance problem.

- Chapter 3: Heuristic Representation & Evaluation. This chapter details the underlying format used to automatically create heuristics in the experiments presented in this thesis. It provides technical information regarding the systems that use the heuristics to solve Boolean Satisfiability problem instances. This chapter also details the fitness function used throughout this thesis to gauge the effectiveness of a heuristic, and evaluates the performance of previously existing, hand-crafted heuristics.

- Chapter 4: Exhaustive Enumeration & GP. This chapter presents the methodology and results from experiments designed to automatically create heuristics using exhaustive enumeration and genetic programming. It provides greater emphasis on the experiments conducted using exhaustive enumeration. This chapter also looks at individual heuristics created from both methods, and presents data showing how they perform on larger problem instances.

- Chapter 5: Analysing Heuristics Using the MTED. This chapter presents a search space analysis performed using the results obtained from the exhaustive enumeration experiments described in Chapter 4. The minimum tree edit distance metric is used to compare heuristics, and observations are made about the landscape of the search space. This chapter also illustrates, through simulated experiments, how a local search algorithm on the heuristics themselves could work as a method of program synthesis.

- Chapter 6: Neighbourhood Generation. This chapter details the neighbourhood generation algorithms for program trees. The neighbourhood of a program tree is a concept based on the minimum tree edit distance metric. The algorithms this chapter presents are a vital component in the overarching local search program synthesis method proposed in Chapter 5.

- Chapter 7: Local Search Program Synthesis. This chapter presents the methodology and results from experiments designed to automatically create heuristics using the local search program synthesis method proposed in Chapter 5. The experiments use the algorithms detailed in Chapter 6. It also presents automated heuristic creation experiments that use an alternate language to represent heuristics. This chapter then highlights some of the created heuristics, and shows how they perform on larger problem instances.

1.2 Academic Publications Produced

- Andrew Burnett and Andrew Parkes. "Systematic search for local-search SAT heuristics". In: *Proceedings of the 6th International Conference on Metaheuristics and Nature Inspired Computing, (META '16).* Marrakech, Morocco, June 2016, pp. 268–270.
  - This short/abstract paper presents the preliminary research in Chapter 4.
- Andrew W. Burnett and Andrew J. Parkes. "Exploring the landscape of the space of heuristics for local search in SAT". In: *Proceedings of the 2017 IEEE Congress on Evolutionary Computation, (CEC 2017).* San Sebastián, Spain. June 2017, pp. 2518-2525.
  - This conference paper presents the research in Chapter 5.
- Andrew W. Burnett and Andrew J. Parkes. "Using local search program synthesis to create local search SAT heuristics". Provisional title, to be submitted 2022.
  - This journal paper presents the research in Chapters 6 and 7.

Chapter 2

Literature Review

2.1 Introduction

In this chapter we provide the reader with an overview of the relevant literature pertaining to the areas of research within this thesis. Succinctly, these are the Boolean Satisfiability (SAT) problem, heuristics for use in solving SAT through local search, previous research in the automated creation of heuristics, program synthesis techniques and the minimum tree edit distance (MTED) problem. The core aim of this thesis is to investigate the applicability of previously underused program synthesis techniques in the automated creation of heuristics. This chapter provides the context for this work, as well as aiding in the understanding of the experiments performed, and the domain the created heuristics are deployed in.

We introduce the SAT problem in Section 2.2. This is the domain that we will test our heuristic creation techniques on. We give a description of the problem, its uses and an overview of the most common algorithms used to solve it.

In Section 2.3 we give a detailed account of the research in heuristics used to drive local search algorithms to solve SAT. One of the goals of this thesis is to search for new, effective local search SAT heuristics by automatically creating them. The components we use as a basis for these created heuristics are inspired by the analysis of previously existing, hand-crafted heuristics, which are detailed in this section.

In Section 2.4 we focus on previous work in automating the design of heuristics, with an emphasis on techniques to automatically create heuristics. We give particular attention to work that has been performed on the same local search SAT domain that we will create heuristics for.

Section 2.5 contains an overview of different program synthesis methods. Program synthesis is an umbrella term that is used to describe techniques to create programs and program fragments automatically. Some of the techniques used in this thesis to automatically create heuristics come directly from the work detailed in this section.

In Section 2.6 we provide an overview of the MTED problem. In our work we use this as a metric to compare created heuristics, and as a basis for the program synthesis technique described in Chapters 5 to 7. Finally in Section 2.7 we present the conclusions we draw from the research presented in this chapter.

2.2 Boolean Satisfiability Problem

The SAT problem is a decision problem that asks, given a propositional logic formula F containing variables $v_1 \dots v_n$, does there exist an assignment of variables to values in the domain $\{False, True\}$ such that F evaluates to $True$. We say that if an assignment exists, then F is *satisfiable* and that $F \in$ SAT. If no assignment exists, we say that F is *unsatisfiable* and $F \notin$ SAT, or that $F \in$ UNSAT. It is common for SAT problem instances to be described in conjunctive normal form (CNF)[1], which can be defined as follows; a propositional formula in CNF contains a set of *clauses* $\{c_1 \dots c_m\}$ distributed over conjunction. Each clause contains a set of *literals* $\{l_1 \dots l_k\}$ distributed over disjunction. A literal is either the occurrence of a variable v_i, or its negation $\neg v_i$. We will be using the convention of writing negated literals as $\overline{v}_i$ for the remainder of this thesis. Figure 2.1 shows two propositional formula; one in CNF and one not. From this point on we will assume that (unless stated otherwise) when referring to a SAT problem, we specifically mean a problem represented as a propositional logic formula in CNF. One notable consequence of presenting SAT problems in CNF is that an assignment that satisfies the formula requires all clauses to be satisfied; that is, each clause must evaluate to $True$.

A restricted variant of the SAT problem called k-SAT refers to those formula described in CNF whose clauses contain exactly k literals. Figure 2.1a shows a 3-SAT problem. It should be noted that some authors, when referring to k-SAT, use the convention that each clause contains at *most* k literals. We use the former definition. k-SAT is a known $\mathcal{NP}$-complete problem [45] for all $k \geq 3$. Using current methods

[1]Any arbitrary propositional formula can be converted to CNF through the rules of propositional logic. Schöning and Torán [155, Chapter 1] provide a detailed overview of the algorithms used to do this.

$$(\overline{v}_1 \lor v_2 \lor v_4) \land$$
$$(\overline{v}_2 \lor \overline{v}_3 \lor \overline{v}_4) \land$$
$$(\overline{v}_1 \lor v_3 \lor v_4)$$

(a) Propositional formula in CNF.

$$(\overline{v}_1 \lor v_2 \lor v_4) \land$$
$$(\overline{v}_2 \land \overline{v}_3 \lor \overline{v}_4) \lor$$
$$(\overline{v}_1 \lor v_3 \land v_4)$$

(b) Propositional formula not in CNF. This is due to the disjunction between the second and third clause, and conjunctions in each of these clauses.

Figure 2.1: Two examples of propositional logic formula; one in CNF, one not.

this makes it a computationally expensive problem to solve, with the fastest algorithm known, PPSZ, running in $\mathcal{O}(1.308^n)$ [80] on 3-SAT problems that contain n variables.

Despite this exponential worst-case running time, software that can solve SAT problem instances continues to be used to solve real-world problems through the use of reductions to SAT. For instance, within the automated design of electronic circuits, SAT is used to great success. It is used extensively in Automated Test Pattern Generation (ATPG), a technique used to find faults in circuits [104]. SAT is also used in the verification of circuit designs through bounded model checking [44]. Marques-Silva [122] provides more information regarding the various uses of SAT in the automated design of electronic circuits.

Within the scope of operations research, techniques to solve SAT have important use-cases in answer set programming [69] and the solving of constraint satisfaction problems (CSPs) [169]. In the wider context of computer science, SAT has uses in such areas as planning [97] and scheduling, program verification [20] and cryptanalysis [162], to name but a few. A broad overview of the real-world use of SAT (and its extensions) can be found in *Handbook of Satisfiability* [18].

A piece of software designed to answer the question of whether an arbitrary SAT problem is satisfiable or not is informally called a *SAT solver*. One common way of proving a formula is satisfiable is through the construction of an assignment that satisfies all clauses. Showing a formula is unsatisfiable is more difficult, as it requires proving that it is impossible to construct a satisfying assignment. One commonly used method of proving unsatisfiability is through a resolution proof. Sometimes, due to the properties of a SAT formula, a proof of satisfiability or unsatisfiability can be trivially constructed [155, Chapter 1].

Usually to construct a proof (of either satisfiability or unsatisfiability) a search is

conducted on the space of solutions of a SAT problem instance. Like in other hard combinatorial problems, there exist two predominant search techniques underpinning the algorithmic design used in SAT solvers; complete and incomplete. A complete search technique is guaranteed to consider the entire search space of solutions and, when given enough computational resources, provide an answer as to whether a problem instance is satisfiable. Incomplete solvers offer no such guarantee; there is no systematic methodology to their search and therefore no guarantee that an answer will be found. However, some incomplete solvers can be highly effective at quickly finding satisfying assignments to SAT problem instances that complete solvers struggle to find in a reasonable amount of time.

These two competing search methods are perhaps best represented by the two most common algorithms used as the basis for many SAT solvers; the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and local search. DPLL is a complete search algorithm that constructs a partial assignment as it explores the search space. It is used to construct proofs of satisfiability and unsatisfiability. Local search, an incomplete, perturbative search algorithm, aims to find a satisfying assignment to a SAT problem instance - and therefore, in this form, it is only able to solve satisfiable SAT problem instances, and cannot provide a proof of unsatisfiability.

However in truth, this relationship between search paradigms, proof techniques and algorithms is not always exact. For example, Audemard et al. [6] described a local search SAT solver that proves unsatisfiability by building a resolution proof. It is not complete, but offers functionality that is not normally expected of a local search SAT solver. There are also examples of SAT solvers that (at the time of their creation) have been considered state-of-the-art which use a hybridisation of local search and DPLL in their construction. An example is the CADICAL solver which placed 4$^{\text{th}}$ in the 2019 SAT Race [89][2]. Other examples of hybrid SAT solvers include the SparrowToRiss solver [11], the MoRsat solver [54] and the HBISAT solver [43].

Though not directly relevant to this thesis, there exist several variants of the SAT problem which are of interest to us, as some of the techniques used to solve them are related to those discussed in this thesis. MAX-SAT is an optimisation variant of SAT where, under CNF, rather than trying to find an assignment that satisfies all clauses, the aim is to maximise the number of satisfied clauses. Weighted MAX-SAT

[2]Webpage detailing the competition, entrants and results located at `http://sat-race-2019.ciirc.cvut.cz/`

is a further extension of MAX-SAT which assigns each clause a weight. The goal when solving a Weighted MAX-SAT problem instance is to maximise the sum of the score of the satisfied clause's weights.

As an example of this close relationship between techniques, we will consider a MAX-SAT solver called SATLike-c. This solver is of particular interest as it won one of the subcompetitions, or *tracks*, at the International Conference on the Theory and Application of Satisfiability Testing MAX-SAT Competition held in 2018[3]. The particular subcompetition it won required the submitted solvers to solve Unweighted MAX-SAT problem instances where a global optimum was not known. SATLike-c is a hybrid MAX-SAT solver that initially performs local search, then in a secondary stage uses a complete solver in an attempt to find an optimal solution. In its local search stage it uses the solver CCLS. This is a local search MAX-SAT solver based on configuration checking [34] (an overview of which is provided in Section 2.3.6), a technique that had previously been found to be an effective basis for solving conventional SAT problems through local search.

The format of the rest of this section is as follows; in Section 2.2.1 we provide an in-depth overview of DPLL and local search. In Section 2.2.2 we discuss a specific subclass of SAT problems that local search is highly effective at solving. Finally in Section 2.2.3 we discuss the conclusions that can be drawn from the literature presented in this section.

2.2.1 Algorithms to Solve SAT

In this subsection we outline the two most common algorithms used as templates to design SAT solvers. The two algorithms are DPLL and local search. DPLL, and the closely related Davis-Putnam algorithm, were described first. Consequently they were considered the standard method for solving SAT problem instances for nearly thirty years. However, after research appeared showing that the local search-based SAT solver GSAT [161] could provide improved performance compared to DPLL, research interest piqued in this area. As subsequent advancements were made in the development of the heuristics driving local search-based SAT solvers, which yielded improved performance, comparatively less attention was given to DPLL. Yet, DPLL's inability (at the time) to perform as well as the local search-based

[3]Webpage detailing the competition, entrants and results located at `https://maxsat-evaluations.github.io/2018/`

SAT solvers needs to be taken in context; this development came at a time when the computational resources available meant that many subsequently developed techniques used to improve DPLL's performance were not feasible at the time. Despite this, DPLL continued to remain relevant as it could prove unsatisfiability, and, through improved technologies and additional techniques, DPLL-based solvers are now generally preferable to local search-based SAT solvers in domains where either a proof of unsatisfiability is required, or on problems perceived to be difficult for local search-based SAT solvers to solve.

To facilitate the understanding of these two algorithms, we begin by introducing a series of definitions about SAT problems and assignments, before introducing each algorithm. The definitions are as follows:

Definition 1 (Variable Set)
The variable set of a SAT problem F is the set of all variables in that SAT problem. For a SAT problem containing n variables, this is usually expressed as $\{v_1 \dots v_n\}$. To refer to F's variable set, we write $\textsc{Vars}(F)$.

Definition 2 (Clause Set)
The clause set of a SAT problem F is the set of all clauses in that SAT problem. For a SAT problem containing m clauses, this is usually expressed as $\{c_1 \dots c_m\}$. To refer to F's clause set, we write $\textsc{Clauses}(F)$.

Definition 3 (Literal Set)
The literal set of a clause c is the set of literals in that clause. For a clause containing k literals, this is usually expressed as $\{l_1 \dots l_k\}$. To refer to c's literal set, we write $\textsc{Lits}(c)$.

Definition 4 (Literal's Sign/Variable)
A literal l is made up of a sign - either $True$ or $False$ - and a variable v.

A literal's sign is $True$ if the literal l containing variable v is in the form v, and $False$ if in the form $\overline{v}$. To refer to l's sign we write $\textsc{Sign}(l)$.

A literal's variable is the underlying variable in the literal. To refer to l's variable we write $\textsc{Var}(l)$.

Definition 5 (Assignment)
An assignment A for a SAT problem F is a map of variables $v \in \textsc{Vars}(F)$ to values $\{True, False\}$. A complete assignment is an assignment where all variables in $\textsc{Vars}(F)$ *have a value associated with them. A partial assignment is an assignment*

in which some variables may not have associated values set. $\emptyset$ *represents the empty partial assignment. We write* $A[v]$ *to obtain the assignment of* v *in* A*. If* A *is a partial assignment, and* v *is unassigned in* A*, then* $A[v] = \text{Unset}$*. We write* $B = A_{v=True}$ *to signify the partial assignment* B *obtained by taking* A *and setting the variable* v *to* $True$*. We write* $B = A_v$ *to signify the complete assignment* B *obtained by taking the complete assignment* A *and changing the truth variable of* v *to* $\neg A[v]$*. We refer to changing a variable's assignment in this manner as "flipping" the variable's assignment.*

Definition 6 (Satisfied, Unsatisfied and Unset Literal)
A literal l *containing a variable* v *is satisfied under an assignment* A *if* $A[v] = \text{Sign}(l)$*. A literal is unsatisfied if* $A[v] \neq \text{Sign}(l)$*. If* A *is a partial assignment, and* $A[v] = \text{Unset}$*, then we say that a literal containing* v *is* Unset*. To refer to the information about* l*'s satisfied state, we write* $\text{Satisfied}(A, l)$*.*

Definition 7 (Satisfied, Unsatisfied and Unset Clause)
A clause c *is satisfied under an assignment* A *if* $\exists l \in \text{Lits}(c)$, $\text{Satisfied}(A, l) = True$*. It is unsatisfied if* $\forall l \in \text{Lits}(c)$, $\text{Satisfied}(A, l) = False$*. If* A *is a partial assignment then* c *is* Unset *if* $\neg(\exists l \in \text{Lits}(c), \text{Satisfied}(A, l) = True) \wedge (\exists l \in \text{Lits}(c), \text{Satisfied}(A, l) = \text{Unset})$*. To refer to the information about* c*'s satisfied state, we write* $\text{Satisfied}(A, c)$*.*

Definition 8 (Satisfying Assignment)
A SAT formula F *is satisfied under an assignment* A *if* $\forall c \in \text{Clauses}(F)$, $\text{Satisfied}(A, c) = True$*. To refer to the information about* F*'s satisfied state, we write* $\text{Satisfied}(A, F)$*.*

DPLL

Historically the earliest SAT solvers were based on the Davis-Putnam algorithm [50] and a refined variant called the DPLL algorithm [49]. Many modern-day, state-of-the-art complete SAT solvers still use the DPLL algorithm at their core.

When visualising the solution space of a SAT problem as a tree, DPLL can be thought of as starting at the root of the tree and traversing it in a depth-first search manner. As it moves downwards through the tree, it constructs a partial assignment to the problem. If a conflict is found in the current partial assignment - that is to say, a clause is found to be unsatisfiable - then the algorithm stops and backtracks to

a previous decision point. The SAT problem is unsatisfiable when all branches in the tree have been traversed and no satisfying assignment found.

DPLL uses two methods of simplification to reduce overall computation, outlined as follows:

- **Pure Literal Elimination:** Given all currently UNSET variables vs under a partial assignment A and SAT problem F, a variable $v \in vs$ is a pure literal if the following holds. For all clauses cs currently UNSET in F under A, v only appears in cs either in the form v or the form $\overline{v}$. Pure literals can be assigned to make all clauses they appear in satisfied, without making any currently satisfied clauses unsatisfied.

- **Unit Clause Propagation:** Given all currently UNSET clauses cs under a partial assignment A and SAT problem F, a clause $c \in cs$ is a unit clause if the following holds. There is exactly one literal $l \in \text{LITS}(c)$ which satisfies the logical statement $\text{SATISFIED}(A, l) = \text{UNSET}$. For a solution to be found, the UNSET literal in c must be assigned a value in such a manner so as to make c satisfied.

In Algorithm 2.1 we show an outline of the DPLL algorithm. Modern-day DPLL-based solvers use sophisticated data structures, heuristics, and techniques such as conflict driven clause learning (CDCL) to make them highly effective at finding proofs of satisfiability and unsatisfiability in large SAT problems.

There exist other complete search methods for SAT such as Stålmarck's method [164], which are beyond the scope of this thesis. For an overview of modern techniques to improve the performance of DPLL including detailed examples of CDCL, as well as the most recent advances in complete solvers, we point the reader to *Handbook of Satisfiability* [18] and *The Satisfiability Problem: Algorithms and Analyses* [155].

Local Search for SAT

The algorithm that underpins most local search SAT (LS-SAT) solvers can be described as follows; it is an iterative algorithm that begins by initialising a complete assignment of all variables in the problem. On each iteration, a perturbation of the previous assignment is obtained by changing the truth value of one of the variables; we "flip" a variable's assignment from $False$ to $True$ or vice versa. A check is performed on each iteration to deduce whether the assignment now satisfies the

Algorithm 2.1 DPLL

Input: F SAT problem instance in CNF.

Output: $True$ if $F \in SAT, False$ if $F \notin SAT$.

```
algorithm DPLL(F)
    assignment = ∅                                          ▷ Initial empty assignment.
    return DPLL-Internal(assignment, F)

algorithm DPLL-Internal(assignment, F)
    clauses = Clauses(F)
    assignment = Unit-Propagation(assignment, clauses)
    assignment = Find-Pure-Literals(assignment, clauses)
    if (∀c ∈ clauses, Satisfied(assignment, c) = True) then ▷ All clauses True.
        return True
    if (∃c ∈ clauses, Satisfied(assignment, c) = False) then ▷ Clause is False.
        return False
    var = Pick-Var(assignment, F)                           ▷ Pick an unassigned variable.
    return
      DPLL-Internal(assignment_{var=True}, F) ∨
      DPLL-Internal(assignment_{var=False}, F)
```

Algorithm 2.2 LOCAL-SEARCH for SAT

Input:	F	SAT problem instance in CNF.
	$maxFlips$	Maximum number of iterations to run for.
Output:		$True$ if a solution is found. **null** if no solution is found.

```
algorithm LOCAL-SEARCH(F, maxFlips)
    assignment = INITIALISE-ASSIGNMENT(F)          ▷ Create initial assignment.
    if (SATISFIED(assignment, F)) then
        return True
    for (iteration ∈ {1 ... maxFlips}) do
        varToFlip = PICK-VAR(assignment, F)
        assignment = assignment_varToFlip
        if (SATISFIED(assignment, F)) then         ▷ Check if F is satisfied.
            return True
    return null
```

formula - specifically, whether all clauses are satisfied. An outline of this algorithm is shown in Algorithm 2.2. An example of two problem assignments can be seen in Figure 2.2. The assignment in Figure 2.2b is a perturbation of the assignment in Figure 2.2a, obtained by flipping the 6th variable.

A local search SAT solver answers the question of whether a formula is satisfiable by attempting to find an assignment that satisfies it. The difference between complete and incomplete algorithms is in their guarantee that a definitive answer will be returned. Local search does not stipulate any ordering on how the variables are changed, there is no guarantee that all possible solutions will be evaluated, and it is not uncommon for solutions to be revisited. This is why local search for SAT is an incomplete search method. Succinctly, if a solution is found, then a formula is satisfiable. If one is not found, that does not mean that the formula is unsatisfiable.

There exist many variations and augmentations of local search. Many of these are generic methodologies that are not problem-specific, and some of them have been used to augment LS-SAT solvers; for example Spears [166] applied simulated annealing to SAT. However, in much of the work describing advancements in the effectiveness of LS-SAT solvers, the improvements have come from new *heuristics*. When we refer to heuristics in the domain of LS-SAT solvers, we specifically mean the functions designed to pick the next variable to flip - in Algorithm 2.2, the heuristic would be called by the function PICK-VAR. The way in which this variable is chosen

Variable	1	2	3	4	5	6	7	8
Assignment	*False*	*True*	*False*	*True*	*True*	*True*	*False*	*True*

(a) An example of an assignment to a SAT problem instance containing 8 variables.

Variable	1	2	3	4	5	6	7	8
Assignment	*False*	*True*	*False*	*True*	*True*	*False*	*False*	*True*

(b) An example of an assignment to a SAT problem instance containing 8 variables. This has been obtained by taking the assignment in Figure 2.2a and changing the 6th variable from *True* to *False*.

Figure 2.2: Two examples of assignments to a SAT problem instance containing 8 variables. The assignment in Figure 2.2b has been obtained by taking the assignment in Figure 2.2a and flipping a variable in it.

can drastically effect the performance of an LS-SAT solver. While the initialisation function can be considered a heuristic function as well, in this thesis we specifically concentrate on heuristics designed to direct the internal search mechanism.

Local search-based SAT solvers are sometimes referred to as stochastic local search SAT (SLS-SAT) solvers, and their internal heuristics as SLS-SAT heuristics. Stochasticity when discussing local search in general refers to variants of local search that employ non-determinism, and will not always return the same result. This can be in the form of random restarts, randomness within heuristics, the initialisation function, and other techniques. In truth nearly all heuristics designed for an LS-SAT solver use some form of stochasticity in their construction, and there are few examples of purely deterministic LS-SAT heuristics. Throughout this thesis we use the term local search SAT (and the acronym LS-SAT), rather than stochastic local search SAT (and the acronym SLS-SAT).

2.2.2 SAT Problem Hardness

Though DPLL-based solvers can give a definitive answer as to whether a SAT formula is satisfiable, they find some problems difficult to solve quickly. Local search SAT solvers excel at solving some of these types of problems. In this subsection we look at satisfiable k-SAT problem instances in and around the *phase transition* region that local search-based solvers generally outperform DPLL-based solvers on.

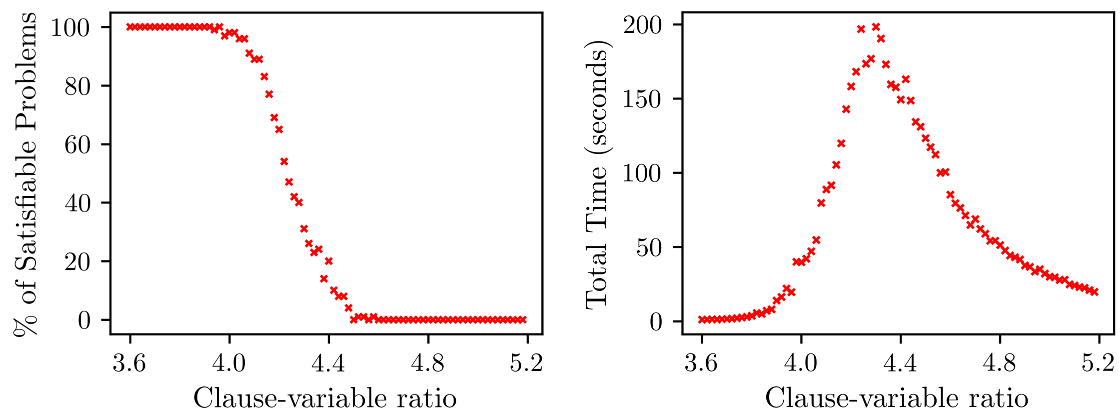
A phase transition region is an observed phenomenon in the problem space of many

$\mathcal{NP}$-hard problems, where a significant change in problem difficulty is found. Phase transition regions have been observed in many domains, such as graph colouring, the Travelling Salesman problem (TSP) and the finding of Hamilton circuits [42], not to mention within SAT [67]. Zeng and Lu [179] state that "Many experimental results have shown that for a class of $\mathcal{NP}$-hard problems one or more "order parameters" can be defined, and hard instances occur around the critical values of these order parameters". This instance hardness is not measured by computational complexity, but by empirical evaluation of the run-time of complete algorithms on problem instances. When a complete algorithm is used to solve problem instances further from a phase transition region, as described by these order parameters, they exhibit shorter run-times than those problems closer to it.

Within SAT one of the motivations behind investigating local search methods was the observation that complete algorithms did not perform well on certain randomly generated k-SAT problem instances [129, 161]. As researchers experimented with different problem instances, they found that DPLL-based algorithms took much longer to solve some instances than others. These problems could be categorised by their ratio of clauses to variables. For 3-SAT, problem instances with a ratio of around ≈ 4.27 were found to exhibit this behaviour. This region is the phase transition region for 3-SAT. Phase transition regions have been found for other k values; for example, 5-SAT's phase transition region is known to exist at ≈ 21.12, and for 7-SAT at ≈ 87.79.

These regions mark a stark contrast between not just hardness, but of satisfiability itself. To illustrate this phenomenon, we created a set of 3-SAT problem instances, and ran a complete solver on them. Each problem instance had 250 variables, and from between 900 to $1,300$ clauses with 5 step increments (a total of 81 different clause values). For each unique clause value, we generated 250 problem instances. The MAPLESAT solver was then ran on each problem instance, until a definitive answer about satisfiability was found. In Figure 2.3 we show the percentage of satisfiable formula in, and the run-time required to solve, the set of 250 instances at each clause value. The reader can clearly see that as the clause-variable ratio approaches 4.27, the number of satisfiable problems changes from nearly all to nearly none. The reader can also see that the amount of time it takes for the algorithm to terminate greatly increases as we approach this clause-variable ratio, before decreasing again.

Historically, these problems at the phase transition region in SAT have been



(a) Percentage of the SAT formula that are satisfiable at each clause-variable value.

(b) Total time to solve all SAT formula for each clause-variable value.

Figure 2.3: Results from solving 3-SAT problems using the MAPLESAT solver. Each instance contains 250 variables and between 900 and $1,300$ clauses (step increase of 5). For each clause value, we created a set of 250 random formula. We show the % that are satisfiable and the time taken to solve each set.

difficult for DPLL-based algorithms to solve, but easier for local search-based algorithms to solve. As an example of this, we will look at the results of the 2018 SAT Competition [151][4]. The SAT Competition is a regularly held event where different SAT solvers are tested against each other to determine which is the most effective. In the 2018 edition of the competition several tracks were held, each of which tested the submitted solvers on different types of problem instance. One of these tracks, called the Random track, focused on randomly created instances, some of which can be categorised as problems in and around the phase transition region.

The competition was ran as follows; given 255 satisfiable problem instances, each solver was allowed 30 minutes to solve each problem. The 255 problems were split into various sets. These sets were named for the origin of the problems contained within; for example the set named “3SAT” contains problems in and around the phase transition region for 3-SAT. Other sets contain randomly created SAT problems that have a non-uniform number of variables in each clause, and therefore a clause-variable ratio that is not a good indicator of their problem hardness. The largest problems

[4]Webpage detailing the competition, entrants and results located at `http://sat2018.forsyte.tuwien.ac.at/`

have millions of variables. In Table 2.1 we present a detailed overview of the results.

Though the results show that SPARROW2RISS, a hybrid solver, won the competition overall, the detailed results also show that it was not the best performing on all sets of problem instances. In fact, several of the local search solvers were able to outperform it (as well as the CDCL-based solvers) on certain subsets of the 255 instances; specifically those containing instances at the phase transition region for 3, 5 and 7-SAT. In subsequent competitions the Random track has not been held. We feel that, while these results show that state-of-the-art hybrid solvers, and CDCL solvers, are able to outperform LS-SAT solvers on some randomly generated instances, they also show that local search-based solvers are still more effective at solving k-SAT problem instances in and around the phase transition region. In addition to this, their use in hybridised solvers suggests that the continued development of effective local search-based solvers still has a role to play in the creation of general-purpose solvers.

2.2.3 Summary & Discussion

In this section we have provided a brief overview of the Boolean Satisfiability problem, detailed two algorithms used to solve it, touched on its extensions, and discussed a specific subclass of SAT problem in and around the phase transition region.

When solving a SAT problem instance, currently it is common practice for a complete algorithm to be invoked at some stage of the solving process. This is predominantly due to the difficulty that incomplete algorithms, such as local search, have in constructing a proof of unsatisfiability. Though local search does perform well when solving certain classes of SAT problem when compared to DPLL, on its own its use is limited.

Despite this, we feel that we have shown that local search algorithms, and by extension the heuristics that direct the search within them, still have a role to play in solving real-world SAT problems. The evidence presented in Section 2.2.2 showed that local search-based SAT solvers can still solve problems that DPLL-based solvers struggle to solve. We also provided evidence of the continued role that hybrid solvers have to play in solving SAT. Well-designed heuristics that drive the local search algorithm are still required to design such solvers.

Further to this, some algorithms to solve SAT's optimisation variants make use of local search and heuristics to guide their internal search mechanisms. We have

Table 2.1: Detailed results of the 2018 SAT Competition, Random track. Each problem set is named, and the total number of instances in that set shown. For each solver, the type of that solver is indicated; CDCL is based on conflict driven clause learning and DPLL, LS is based on local search, and Hybrid is a combination of the two. The data shows how many problem instances each solver could solve. Bold typeface of a result indicates a solver that solved the most of all solvers for that set of instances.

Solver	Type	Random								Total (255)
		cnf				bz2				
		afla-qhid (55)	rnd-komb (55)	fla-barthel (55)	Balint (30)	3SAT (20)	5SAT (13)	7SAT (20)	alt (7)	
Sparrow2Riss	Hybrid	**55**	**55**	**55**	12	0	3	8	0	**188**
gluHack	CDCL	**55**	**55**	**55**	0	0	0	0	0	165
glucose-3_PADC_10	CDCL	**55**	**55**	**55**	0	0	0	0	0	165
glucose-3_PADC_3	CDCL	**55**	**55**	**55**	0	0	0	0	0	165
expGlucoseSilent	CDCL	**55**	**55**	**55**	0	0	0	0	0	165
CPSparrow	LS	21	**55**	**55**	16	0	6	10	0	163
dimetheus	LS	12	12	**55**	**20**	**20**	**13**	**16**	**7**	155
probSAT	LS	12	14	**55**	17	18	11	11	0	138
YalSAT	LS	12	9	**55**	15	16	12	10	0	129
lawa	LS	12	6	**55**	0	8	0	0	0	81

provided an example of how solvers for SAT's optimisation variants can make use of heuristics that utilise ideas originally designed for SAT. A system to automatically design SAT heuristics could be easily and quickly modified to work with SAT's optimisation variants. Therefore, the research in such systems could prove useful for real-world applications that require software to solve these optimisation variants.

In Section 2.2.2 we discussed k-SAT problem instances in and around the phase transition region. These are the types of problem instances we will be testing the automatically created heuristics on. They have previously been shown to be difficult for DPLL-based solvers to solve, and comparatively easy for LS-SAT solvers to solve.

SAT is itself a broad research area and, as we only concentrate on a small area of it in this thesis, this section can only be considered to be an introduction to the topic at large. For more information, the reader is directed to *The Satisfiability Problem: Algorithms and Analyses* [155], which provides an introduction to many core research areas in SAT, and discusses in detail the methods used to solve it. *Handbook of Satisfiability* [18] provides a detailed and in-depth review of many areas of research in SAT and its extensions.

2.3 LS-SAT Heuristics

In this section we provide an overview of previously described LS-SAT solvers in the research literature that have either proven to be effective at solving the SAT problem, or we deem the work undertaken relevant to this thesis. Our focus is almost exclusively on the heuristics that drive these LS-SAT solvers. It is through the literature presented in this section that we identify many of the components we use to create heuristics in later chapters. We include examples of the pseudocode describing the heuristic component of several LS-SAT solvers, and not the overarching LS-SAT algorithm of the described solvers. Each given heuristic has been presented in a way that, if the provided pseudocode were substituted for the function Pick-Var in Algorithm 2.2, an LS-SAT solver utilising the substituted heuristic would be created. Furthermore, at certain points in this section we describe LS-SAT solvers that require some additional mechanism to function correctly. These additional mechanisms exclusively concern update functions for auxiliary data structures. These update functions are clearly labelled and, if they were to be used as part of the overarching LS-SAT algorithm in Algorithm 2.2, they would be inserted at the end

of each iteration of the overarching local search loop.

We present the heuristics in a semi-chronological order, as this allows the reader to identify the research trends as they become more and less relevant. We have attempted to group the heuristics into several subsections based upon the common techniques used in their formulation. However this is not always possible as competing ideas sometimes appeared at the same time, or appeared as bit-parts in heuristics before later being reused as effective, stand-alone heuristics.

We make an attempt to use uniform terminology in our descriptions, as well as uniform pseudocode where it is provided. An explanation of the commonly used functions in the pseudocode is found in Table 2.2. We also provide a set of definitions about metrics and properties of SAT problems used throughout this section. The set of definitions in Section 2.2.1 are also used, and we refer the reader to them if unfamiliar with SAT. These are not the only metrics and descriptions used, but it does define all those that transcend multiple subsections. The definitions are as follows:

Definition 9 (True Literals)
For a SAT problem F, clause $c \in \textsc{Clauses}(F)$ and complete assignment A, this is the number of satisfied literals in c under A. It can be defined as $|\{l \in \textsc{Lits}(c), \textsc{Satisfied}(A, l) = True\}|$. To refer to this value we write $\textsc{TrueLits}(A, c)$.

Definition 10 (Variable's Clause Set)
For a SAT problem F and variable $v \in \textsc{Vars}(F)$, this is the set of clauses that contain a literal that is either the positive occurrence or negative occurrence of v. It can be defined as $\{c \in \textsc{Clauses}(F),\ (\exists l \in \textsc{Lits}(c), \textsc{Var}(l) = v)\}$. To refer to this set we write $\textsc{ClauseSet}(F, v)$.

Definition 11 (Variable's True/False Literal Set)
For a SAT problem F, variable $v \in \textsc{Vars}(F)$ and complete assignment A, v's True Literal set is defined as the set of clauses that contain a literal l that is both a satisfied literal and contains v. v's False Literal set is the set of clauses that contain a literal l that is both unsatisfied and contains v. It follows that when the variable v is flipped in a local search algorithm, these sets also flip. The two sets can be described as $\{c \in \textsc{ClauseSet}(F, v), (\exists l \in \textsc{Lits}(c), \textsc{Satisfied}(A, l) = True \wedge \textsc{Var}(l) = v)\}$ and $\{c \in \textsc{ClauseSet}(F, v), (\exists l \in \textsc{Lits}(c), \textsc{Satisfied}(A, l) = False \wedge \textsc{Var}(l) = v)\}$ respectively. To refer to the True Literal set we write $\textsc{TrueLitSet}(F, A, v)$ and $\textsc{FalseLitSet}(F, A, v)$ to refer to the False Literal set.

Definition 12 (Clause Weighting)

A clause weighting scheme W under a SAT problem F is a matrix of positive numbers, each of which is associated with a clause $c \in$ CLAUSES(F). Each value represents the "weight" of the clause c, and a weight of n is analogous to having n copies of c in F. The "base" (representing the original problem F) weighting has all values set at 1, and is written as so. We write $W_c = x$ to change a clause c's weight to x, $W = x$ to set all weights to x and $\overline{W}$ to obtain the mean of all the weights.

Definition 13 (Positive Gain)

For a SAT problem F, variable $v \in$ VARS(F), weighting scheme W and complete assignment A, this is a metric associated with a variable that represents the number of currently unsatisfied clauses that will become satisfied if v is flipped. It is also known as makes. It can be computed as:

$$\sum_{c \,\in\, \textsc{FalseLitSet}(F,\, A,\, v)} \begin{cases} W_c & \textit{if } \textsc{TrueLits}(A,\ c) = 0 \\ 0 & \textit{otherwise} \end{cases} \tag{2.1}$$

To refer to this value, we write $\textsc{PosGain}_W(A,\, F,\, v)$. If the assignment and SAT formula are obvious from the context, we write $\textsc{PosGain}_W(v)$. If a set of variables are ordered according to their $\textsc{PosGain}_W$, they are ordered from smallest to largest; the variable with the largest $\textsc{PosGain}_W$ is considered the best. A variable's $\textsc{PosGain}_W$ is always a positive integer.

Definition 14 (Negative Gain)

For a SAT problem F, variable $v \in$ VARS(F), weighting scheme W and complete assignment A, this is a metric associated with a variable representing the number of currently satisfied clauses that will become unsatisfied if v is flipped. It is also known as breaks. It can be computed as:

$$\sum_{c \,\in\, \textsc{TrueLitSet}(A,\, v)} \begin{cases} W_c & \textit{if } \textsc{TrueLits}(A,\ c) = 1 \\ 0 & \textit{otherwise} \end{cases} \tag{2.2}$$

To refer to this value, we write $\textsc{NegGain}_W(A,\, F,\, v)$. If the assignment and SAT formula are obvious from the context, we write $\textsc{NegGain}_W(v)$. If a set of variables are ordered according to their $\textsc{NegGain}_W$, they are ordered from largest to smallest; the variable with the smallest $\textsc{NegGain}_W$ is considered the best. A variable's $\textsc{NegGain}_W$ is always a positive integer.

Definition 15 (Net Gain)

For a SAT problem F, variable $v \in \textsc{Vars}(F)$, weighting scheme W and complete assignment A, this is a metric associated with a variable representing the total difference in number of satisfied clauses if v is flipped. It is also known as score. It can be computed as:

$$\textsc{PosGain}_W(A,\ F,\ v) - \textsc{NegGain}_W(A,\ F,\ v) \tag{2.3}$$

To refer to this value, we write $\textsc{NetGain}_W(A,\ F,\ v)$. If the assignment and SAT formula are obvious from the context, we write $\textsc{NetGain}_W(v)$. If a set of variables are ordered according to their $\textsc{NetGain}_W$, they are ordered from smallest to largest; the variable with the largest $\textsc{NetGain}_W$ is considered the best. A variable's $\textsc{NetGain}_W$ can be a positive or negative integer.

Definition 16 (Age)

For a current complete assignment A, initial assignment B, SAT problem F, sequence of variables vs that detail the variables flipped to obtain A from B, and variable $v \in \textsc{Vars}(F)$, this is a metric associated with a variable that represents the number of flips since v was last changed. Specifically this is the distance in vs from the end to the last occurrence of v. If $v \notin vs$, then the age of v equals the length of vs. A variable's age is always a positive integer.

To refer to this value, we write $\textsc{Age}(v)$. We assume the current assignment is obvious from the context. If a set of variables are ordered according to their Age, *they are ordered from smallest to largest; the variable with the largest* Age *is considered the best.*

At various points in the following subsections, we make references to *variable metrics*. These can be viewed as metrics that provide some description of a property of a variable. We use these metrics to order sets of variables. By variable metric, we specifically refer to the PosGain, NegGain, NetGain and Age metrics defined above, as well as some additional metrics defined in Section 2.3.6 that are used exclusively in that subsection.

The reader should also note that, at various points in the following subsections, we make references to the SAT Competition. This is a regularly held benchmarking competition that tests various SAT solvers on different types of problems. Rather than providing the webpage of each individual competition, we point the reader to `http://www.satcompetition.org/`, which contains links to all of the SAT Competition results referenced in this section.

Table 2.2: The set of functions used in the pseudocode in Section 2.3. Each entry shows the function's name, the function's type signature, and a short explanation of that function's operation.

<table>
<tr><td>Order-Vars(f, vs)</td><td>Variable Metric → [Variable] → [Variable]</td></tr>
<tr><td colspan="2">Given the collection of variable metrics f, each of which has an associated ordering, the collection of variables vs, Order-Vars does the following; the variables in vs are sorted in descending order according to the variable metric values in f - the variable with the best variable metric(s) is placed at the front of the returned list. To determine the ordering of two variables, the variable metric at the 0th element of f is used. If the two variables are equal under this variable metric, then the next element in f is used. If, after exhausting all variable metrics in f, two or more variables are deemed equal, then the variables are ordered randomly. We return the variables in vs in this new ordering.</td></tr>
<tr><td>Pick-Random(vs)</td><td>[Variable] → Variable</td></tr>
<tr><td colspan="2">Given the collection of variables vs, this function picks one at random from vs.</td></tr>
<tr><td>With-Probability(p)</td><td>Probability → Bool</td></tr>
<tr><td colspan="2">With probability p returns $True$ and with probability $1 - p$ returns $False$.</td></tr>
<tr><td>Pick-Broken()</td><td>[Variable]</td></tr>
<tr><td colspan="2">This function picks a single currently unsatisfied clause, and returns all variables within that clause. Each unsatisfied clause has an equal chance of being picked.</td></tr>
<tr><td>Pick-Weighted-Var(W)</td><td>[(Weight×Variable)] → Variable</td></tr>
<tr><td colspan="2">Given the collection of pairs of weights and variables W, this function performs a weighted pick on the variables from W using the associated weights, returning the chosen variable.</td></tr>
<tr><td>Filter(vs, f)</td><td>[Variable] → (Variable → Bool) → [Variable]</td></tr>
<tr><td colspan="2">Given the collection of variables vs and the function f, which takes a variable and returns a boolean, this function filters vs using f. It returns all (if any) variables from vs that satisfy f.</td></tr>
</table>

Algorithm 2.3 GSAT Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
Output:		The variable to be flipped.

```
algorithm GSAT(F, A)
    vars = Order-Vars([NetGain_1], Vars(F))
    return vars[0]
```

2.3.1 GSAT & Variants

Greedy SAT, commonly known as GSAT [161], was one of the first effective local search-based SAT solvers described. Also independently described by Gu [72] (called SAT6.0 in that work), it utilises a simple heuristic based on the hill-climbing algorithms used in generic local search. The heuristic works by choosing the variable that, if flipped, will maximise the number of satisfied clauses. In the terminology we use, this is the variable with the maximum $\textsc{NetGain}_1$. The pseudocode for the GSAT heuristic is shown in Algorithm 2.3. Like all heuristics in this section, this pseudocode can be substituted into the local search algorithm shown in Algorithm 2.2 to obtain the complete GSAT algorithm.

If it is found that two or more variables have the same $\textsc{NetGain}_1$, ties are broken by choosing one of them with uniform randomness, thus making the whole heuristic technically stochastic. The reader should note that, unlike conventional hill-climbing algorithms in local search, GSAT can pick moves that decrease the number of satisfied clauses (in terms of how local search is applied to optimisation problems, this would be analogous to reducing the current fitness), or leave the total number unchanged.

The authors tested GSAT on satisfiable 3-SAT problem instances around the phase transition region, and other SAT problem instances created from reductions from the n-queens problem and the Boolean induction problem. For the 3-SAT problem instances it was found that GSAT was able to solve more than the compared complete Davis-Putnam algorithm. Of the problems that the Davis-Putnam algorithm could solve, GSAT was shown to solve them much more quickly. Through these results, research interest piqued in local search methods for solving SAT, and its development inspired subsequent research that was primarily focused on improving performance.

This heuristic, in hindsight, can be considered an intensification technique; the heuristic directs the search to areas of the solution space containing assignments that satisfy as many clauses as possible, and potentially where a satisfying assignment may be found.

Although GSAT was shown to be an effective method for solving SAT problem instances, the authors noted that it could struggle to find a satisfying assignment under certain circumstances; specifically those where the algorithm got stuck in local optima. This occurred when the heuristic made a series of choices that put it in an area of the search space that contained no satisfying assignment, and it could not escape from the local optima as the heuristic was unable to choose the correct variable(s) to allow it to do so.

Some of the subsequent research based on GSAT concentrated on mechanisms to escape local optima. Selman and Kautz [159] described three extensions to GSAT; one that added random walk, one that added clause weighting, and one that used a bespoke initialisation function. We discuss the first two of these mechanisms in Sections 2.3.2 and 2.3.4 respectively.

In this early stage of LS-SAT heuristic research, questions were also asked of how important the design choices of GSAT were. Gent and Walsh [66, 68] performed experiments to ascertain the importance of choosing the most greedy variable; that is to say, whether it is important to pick the variable with the *highest* $\textsc{NetGain}_1$. In an attempt to answer this, several augmentations of GSAT were considered as follows:

- CSAT: In this heuristic two sets of variables are considered. Set 1 contains all variables in the problem with $\textsc{NetGain}_1 > 0$. Set 2 contains all variables with $\textsc{NetGain}_1 = 0$. The heuristic works as follows; if Set 1 is not empty, a variable is chosen at random from it. Else if Set 2 is not empty, a variable is chosen at random from it. Else a variable is chosen from $\textsc{Vars}(F)$ at random.

- TSAT: This heuristic works in the same way as CSAT, except the variable chosen from Set 1 is the variable with the minimum $\textsc{NetGain}_1$. Ties are broken randomly.

- ISAT: This heuristic works as follows: Set 1 contains all variables with a $\textsc{NetGain}_1 \geq 0$. If Set 1 is not empty, then a variable is chosen at random from it. Else a variable from $\textsc{Vars}(F)$ is chosen at random.

Algorithm 2.4 HSAT Heuristic

Input: F SAT problem instance in CNF.
A Current assignment.
Output: The variable to be flipped.

```
algorithm HSAT(F, A)
    vars = Order-Vars([NetGain_1, Age], Vars(F))
    return vars[0]
```

- SSAT: In this heuristic, the variables are split into two sets as in CSAT. The heuristic works as follows; if Set 2 is not empty, a variable is chosen at random from it. Else if Set 1 is not empty, a variable is chosen at random from it. Else a variable from VARS(F) is chosen at random.

The results presented by the authors showed that the performance of these heuristics, with the exception of SSAT, was comparable to GSAT. Specifically, these heuristics were tested on random 3-SAT problem instances near the phase transition region, and n-queens problems reduced to SAT. This suggested to the authors that picking the most greedy variable was not necessary; simply picking a greedy variable was sufficient to progress the search. In Section 2.3.6 we present examples of state-of-the-art LS-SAT heuristics with mechanisms that split variables into sets according to their NETGAIN$_1$ value. The heuristics described here can be considered precursors to that work.

Gent and Walsh also experimented with variants of these heuristics (including GSAT) where tie-breaks were broken by the AGE of a variable. The pseudocode for GSAT with this change is called HSAT, and is shown in Algorithm 2.4.

A heuristic called IHSAT was also described, which added this additional tie-breaking mechanism to ISAT. Both IHSAT and HSAT were found to perform better than GSAT on the previously used set of SAT problems. This additional mechanism of incorporating a variable's AGE metric would be further developed in later described heuristics.

In this subsection we have provided an overview of some of the initial research in LS-SAT heuristics. Several common techniques have been introduced that have gone on to inspire mechanisms in more effective LS-SAT heuristics, as we will see in the following subsections.

Algorithm 2.5 GWSAT Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	p	Noise parameter.
Output:		The variable to be flipped.

```
algorithm GWSAT(F, A, p)
    if (withProbability(p)) then
        return pickRandom(pickBroken())
    else return GSAT(F, A)
```

2.3.2 WalkSAT & Variants

In the early days of research into LS-SAT heuristics, several competing ideas were developed that would later be revisited and combined to create new heuristics. In the previous subsection we presented the beginnings of LS-SAT heuristic research, where intensification strategies were found to work well on SAT. Yet, choosing the best neighbour was known to not be necessary for a heuristic to perform well. In this subsection we concentrate on the WalkSAT family of heuristics, which use a different mechanism to progress the search.

Selman and Kautz [159] considered an augmentation to the GSAT heuristic that added a "random walk" component to it. This heuristic, called GWSAT, can be described as follows; using a pre-determined probability p[5], a random variable from a randomly chosen unsatisfied clause (or "broken clause") is chosen as the variable to flip, and with probability $1 - p$ the variable returned from calling GSAT is chosen. Pseudocode for GWSAT is shown in Algorithm 2.5.

Random walk, in the context of LS-SAT heuristic research, refers to a heuristic mechanism that chooses a variable randomly from all the variables in a currently broken clause. Flipping one of these variables is guaranteed to satisfy the clause it was chosen from, but may reduce the overall number of satisfied clauses. Yet, by picking one of these variables, it may allow the overarching search algorithm to escape local optima. In this way, it can be considered a diversification technique. The inspiration for the random walk mechanism came from work by Papadimitriou [140], where it was shown that, for a satisfiable 2-SAT problem instance (a subclass of SAT problems known to be solvable in polynomial time) containing n variables, a

[5] In the original work, a value of 0.35 was suggested.

process that randomly picks a variable from a broken clause will find a satisfiable solution in $\mathcal{O}(n^2)$ steps with probability approaching 1.

In experiments performed by the authors, it was found that the GWSAT heuristic could outperform the GSAT heuristic on SAT problem instances derived from Boolean induction formulas and planning problems. A further augmentation of GWSAT with clause weighting (see Section 2.3.4) was able to solve more of these problem instances than GWSAT.

GWSAT is the first example we've shown of a heuristic that uses a pre-determined parameter in its description. The parameter in this case can be changed to make the heuristic favour its intensification strategy (GSAT), or its diversification strategy (random walk). Though the authors did not show experiments using different values for this parameter, other researchers would build on this work to show the effect that different parameter values could have on the performance of similar heuristics. Specifically, through experimentation "tuned" parameter values could be found which provide the best performance for certain problem instance, or types of problem instances.

Selman, Kautz, and Cohen [160] further developed this idea of random walk to describe a heuristic based solely off it. They called this heuristic WalkSAT. Like GWSAT, WalkSAT uses a pre-determined noise parameter called p in its construction. WalkSAT works as follows; from a randomly broken clause c, find the variable v with the smallest NegGain_1. If the NegGain_1 of $v = 0$, then v is returned. Else, with probability p, v is returned and, with probability $1 - p$, a random variable from c is returned. We show the pseudocode for this heuristic in Algorithm 2.6.

The design of WalkSAT is interesting for several reasons. Primarily, both its intensification and diversification strategies return variables from an unsatisfied clause. In its intensification step, rather than try to reduce the overall number of unsatisfied clauses, it performs what can be described as a "soft" greedy step; it will pick a variable that does not break any other clauses. Since it is picking a variable from an unsatisfied clause, it will satisfy at least a single clause. Therefore, the NetGain_1 of the chosen variable will be at least 1. This aligns with the observations of Gent and Walsh [68], who showed that GSAT variants that did not pick the variable with the best NetGain_1, but chose a variable with a positive NetGain_1, were still effective heuristics. If the NegGain_1 of the variable with the lowest NegGain_1 is

Algorithm 2.6 WalkSAT Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	p	Noise parameter.
Output:		The variable to be flipped.

```
algorithm WalkSAT(F, A, p)
    vs = Pick-Broken()
    vars = Order-Vars([NegGain_1], vs)
    v = vars[0]
    if (NegGain_1(v) = 0) then return v
    else if (With-Probability(p)) then return v
    else return Pick-Random(vs)
```

not 0, WalkSAT will pick a variable according to its diversification strategy. The diversification strategy in WalkSAT is itself made up of two individual strategies. How frequently either of these are used is controlled by the pre-determined noise parameter p. The first strategy still picks the variable with the lowest NegGain_1. The second strategy picks a random variable from the clause. Irrespective of which strategy is used, the NegGain_1 of the chosen variable will be > 0. Therefore it is not possible to know the overall effect that flipping it will have on the number of unsatisfied clauses.

One component of LS-SAT heuristic design that we have not touched upon in this, or the previous, subsection is the speed at which the heuristics perform. Or more specifically, how fast the auxiliary data structures that are required to compute the values needed for the heuristic to operate are updated. GSAT-like heuristics require the overarching local search algorithm to maintain a partial ordering of all variables in the problem relative to some variable metric. If the problem is large, this can be computationally expensive to maintain. Heuristics such as WalkSAT only require knowing the variable metric values of the variables from a single clause, which is less computationally expensive to maintain. Because of this, they are able to perform an iteration of local search more quickly, which can allow a solver based on such a heuristic to solve problem instances quicker.

Through experimentation, Selman, Kautz, and Cohen were able to show that WalkSAT could outperform (in both number of solved problems and time taken)

GSAT and GWSAT on the problem sets it was tested on. These were circuit synthesis problems and circuit diagnosis problems. Seitz, Alava, and Orponen [158] performed further experiments on WalkSAT to investigate the role of the parameter, and found that for randomised 3-SAT instances a value of 0.57 appeared to provide the best performance.

McAllester, Selman, and Kautz [123] observed that many of the (then) recently described heuristics made use of an intensification and diversification strategy that could be controlled by a parameter, which when changed could affect the performance of the heuristic. The authors described additional heuristics that also made use of a parameter. These heuristics made use of metrics and ideas previously described in LS-SAT heuristic research, and introduced a new notion of finding the second best variable according to a variable metric. We provide a description of two heuristics from this work, Novelty and R_Novelty. These heuristics were chosen as they provided the best performance of those described by the authors. In the given descriptions, the variable p is the pre-determined noise parameter. The heuristics can be described as follows:

- Novelty: From an unsatisfied clause c, the variables are ordered by their NetGain_1, breaking ties using the Age of a variable. For the two best variables v_1 and v_2 under this ordering do the following; if v_1 does not have the minimum Age among the variables in c return v_1. Else with probability p select v_2 and with probability $1 - p$ pick v_1. The pseudocode for Novelty is shown in Algorithm 2.7.

- R_Novelty: From an unsatisfied clause c, the variables are ordered by their NetGain_1, breaking ties using the Age of a variable. For the two best variables v_1 and v_2 under this ordering do the following; if v_1 does not have the minimum Age among the variables in c return v_1. Else, let $n = \text{NetGain}_1(v_1) - \text{NetGain}_1(v_2)$ and perform one of the following steps:

  1. If $p < 0.5$ and $n > 1$ pick v_1.
  2. If $p < 0.5$ and $n = 1$ with probability $2p$ pick v_2, else pick v_1.
  3. If $p \geq 0.5$ and $n = 1$ pick v_2.
  4. If $p \geq 0.5$ and $n > 1$ with probability $2(p - 0.5)$ pick v_2, else pick v_1.

Algorithm 2.7 NOVELTY Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	p	Noise parameter.
Output:		The variable to be flipped.

algorithm NOVELTY(F, A, p)
  vs = PICK-BROKEN()
  return NOVELTY-INTERNAL(F, A, p, vs)

algorithm NOVELTY-INTERNAL(F, A, p, vs)
  $vars$ = ORDER-VARS([NETGAIN$_1$, AGE], vs)
  $v_1 = vars[0], v_2 = vars[1]$
  if $((\min_{v \in vs} \text{AGE}(v)) \neq v_1)$ **then return** v_1
  else if (WITH-PROBABILITY(p)) **then return** v_2
  else return v_1

Additionally on every 100 flips, a random variable from the problem is chosen to be flipped. This is to stop the algorithm getting stuck flipping the same sequence of variables.

The authors performed experiments using these heuristics and WALKSAT which were designed to determine the effect that the noise parameter had on the performance of the heuristics. The heuristics were tested on a set of satisfiable 3-SAT problem instances. From the experiment's results, the authors observed that there appeared to be a tuned noise parameter value that existed for each pair of algorithm and SAT problem instance which yielded the best performance. However, these tuned parameter values were different for each pair of heuristic and problem instance and, to find these values, computationally expensive experimentation was required.

The authors performed further experiments with the goal of identifying characteristics of the parameter value that were "less sensitive to the details of the various strategies". Several different measures of the behaviour of LS-SAT heuristics were described, and it was found that, for a specific SAT problem instance, the mean variance of the number of satisfied clauses over a run directly correlated with the noise parameter that yielded the best performance. It was suggested that, when the optimal mean variance is found for a specific SAT problem instance, that value can

be used to quickly tune the parameter values of any heuristic for that SAT problem instance. It was thought of as an alternative to the potentially computationally expensive testing of many different parameter values. Though we are aware of no work that makes use of this mechanism, this idea of automatically tuning heuristics that make use of a parameter would be revisited. We discuss research in this area in Section 2.3.3.

In the overall context of the continued development of LS-SAT heuristics, the two heuristics we have described outperformed the heuristic which was considered the state-of-the-art at the time - WalkSAT. The experiments showing this were performed on random 3-SAT satisfiable instances, and SAT problems constructed from planning problem instances and graph colouring instances.

Hoos [85] further developed the theory regarding the performance of LS-SAT heuristics. It had been empirically observed that for many of the (then) state-of-the-art LS-SAT heuristics, getting stuck in local optima was a reoccurring issue. To this end, Hoos described a characteristic of LS-SAT heuristics called probabilistic asymptotic completeness (PAC). A heuristic with PAC applied to a satisfiable SAT problem instance will find a satisfying solution with probability approaching 1 when given enough time. In essence, a heuristic with the PAC property is able to sufficiently explore the search space, and not get stuck in local optima. It was shown that GSAT, Novelty and R_Novelty are not PAC, and that GWSAT is.

Through these results, Hoos described a mechanism to augment Novelty and R_Novelty to make them PAC. The mechanism uses an additional probability parameter *wp* (which should be very small) to choose a random variable from the chosen broken clause, else run like the original heuristic. These augmented heuristics are called Novelty+ and R_Novelty+[6] respectively. The mechanism was designed to diversify a heuristic in a non-deterministic way. An example of the mechanism applied to Novelty to create Novelty+ is shown in Algorithm 2.8.

These new heuristics were shown to outperform Novelty and R_Novelty on SAT instances derived from graph colouring problem instances and random 3-SAT instances. For Novelty+, values of 0.01 and 0.35 for the parameters *wp* and *p* were said to provide good performance.

Li and Huang [107] described two further mechanisms that were used to augment Novelty and build what would prove to be better performing LS-SAT heuristics.

[6] In the original paper these heuristics were called Novelty^{+} and R_Novelty^{+}, but some authors refer to them as just Novelty+ and R_Novelty+. We use the latter form in this thesis.

Algorithm 2.8 NOVELTY+ Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	p	Noise parameter.
	wp	Random walk parameter.
Output:		The variable to be flipped.

```
algorithm NOVELTY+(F, A, p, wp)
    vs = PICK-BROKEN()
    if (WITH-PROBABILITY(wp)) then return PICK-RANDOM(vs)
    else return NOVELTY-INTERNAL(F, A, p, vs)
```

The first mechanism was used to augment NOVELTY to create the NOVELTY++ heuristic. The mechanism works by using a parameter wp to determine whether to use the new strategy or the original heuristic. The new strategy picks the variable in the broken clause with the highest AGE. This mechanism was inspired by observations of how NOVELTY+ performs on certain instances. When picking a random variable from the broken clause, it would usually not pick the "correct" variable to escape local optima. NOVELTY++ was tested against NOVELTY and NOVELTY+ and, when used with effective parameter values (values of $p = 0.3$ and $wp = 0.05$ were suggested), outperformed the other heuristics.

The second mechanism described is an intensification strategy that can be used to augment previously existing heuristics. This mechanism is called G^2WSAT and uses a set of variables we call DECRVARS. DECRVARS is a dynamic set of variables that can be described as follows; on initialisation, it contains all variables with $\text{NETGAIN}_1 > 0$. After a variable v is flipped, v is removed from DECRVARS (if it was contained in the set). Any variables currently in the set that now have $\text{NETGAIN}_1 \leq 0$ are removed, and any other variables that now have $\text{NETGAIN}_1 > 0$ are added. If the variable to flip is continually chosen from DECRVARS, then the overarching algorithm will reach local optima quickly. When DECRVARS is empty, and a diversification strategy then used, the algorithm can escape local optima and DECRVARS will not contain the variable just flipped when it is next probed. Succinctly, the mechanism is designed to reach a state of local optima, escape that state of local optima and not choose the same variables to return to that state.

The heuristic NOVELTY++ augmented with G^2WSAT can be described as follows;

Algorithm 2.9 G^{2}WSAT Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	p	Noise parameter.
	wp	Random walk parameter.
Output:		The variable to be flipped.

algorithm G^{2}WSAT(F, A, p, wp)
  if (DecrVars$\neq \emptyset$) **then**
    return Order-Vars([NetGain$_1$, Age], DecrVars)[0]
  else return Novelty++(F, A, p, wp)

if DecrVars is not empty, pick the variable with the highest NetGain$_1$, breaking ties by Age. Else pick the variable according to Novelty++. The pseudocode for this heuristic is shown in Algorithm 2.9. When referring to G^{2}WSAT in later parts of this thesis, (unless stated otherwise) we mean this specific instantiation of G^{2}WSAT using the Novelty++ heuristic.

This mechanism can be considered to be almost GWSAT-like, in that it has a strong intensification mechanism based on picking a variable from the entire problem, and then a diversification mechanism based on picking a variable from a broken clause. The reader should note that the instantiation of the G^{2}WSAT heuristic shown in Algorithm 2.9 may need its parameters to be tuned for the best performance, as it uses the Novelty++ heuristic as its diversification strategy.

Li and Huang tested the G^{2}WSAT heuristic against the entrants to the 2004 SAT Competition, where it was found to outperform most of those it was tested against. However the authors did note that SDF, a solver based on clause weighting [156], had comparable performance. G^{2}WSAT was entered at the 2005 SAT Competition [17] where, in the Random track, it placed 2$^{\text{nd}}$.

The last two heuristics presented in this subsection, G^{2}WSAT and Novelty++, serve as examples of the changing landscape behind the design of LS-SAT heuristics in the early to mid-2000s. New effective heuristics were found that were either combinations of previously existing ideas, or the addition of new ideas to previously found, effective heuristics. As Fukunaga [60] noted, “humans excel at identifying good potential components of methods to solve problems, but combining them seems to be a more difficult undertaking”. Many of the components for these heuristics had

already been described, but the ingenuity to design effective heuristics using them to maximise performance followed later.

Despite the confusing nomenclature, in this subsection we have shown how effective heuristics can be designed when choosing a variable from a random unsatisfied clause. This subsection also shows how the continued development of effective LS-SAT heuristics was an iterative process, combining previous ideas in new and novel ways to create new "augmented" variants. Many of the ideas presented are still used in state-of-the-art LS-SAT heuristics, as we will see in the forthcoming subsections.

2.3.3 Adaptive Heuristics

In the previous subsection we highlighted several heuristics that use a random walk mechanism in their construction. Every heuristic we highlighted in that subsection makes use of a pre-determined probability parameter to control how much bias is given to either its intensification or diversification strategies. Hoos [85] noted how there is a balance between these two strategies; if there is too much intensification, a heuristic cannot escape local optima. If there is too little, it cannot find good (or satisfying) solutions. In this subsection we discuss a further set of heuristics, all variants of those seen in the previous subsection, that use mechanisms to tune their parameters automatically as the heuristics are running.

The first of these, described by Hoos [84], is a variant of the NOVELTY+ heuristic. In this new heuristic instead of the noise parameter being static throughout the lifetime of the algorithm, it is a variable that can change as the algorithm progresses. The author referred to it as an adaptive noise parameter. Inspired by observations gathered through previous research [85], Hoos proposed a mechanism that allowed the algorithm to become aware of when it was stuck in local optima. The mechanism would then change the parameter value so that the heuristic would have more bias towards one of its strategies. The following rules were proposed regarding the changes in the noise parameter:

- If no improvement in the objective function - that is, the number of satisfied clauses - has been observed in the last $\Theta \cdot m$ flips (where $m =$ the number of clauses), then change the parameter according to $wp = wp + (1 - wp) \cdot \phi$.
- If an improvement has been seen since the parameter was last updated, then change the parameter according to $wp = wp - wp \cdot 2 \cdot \phi$.

Hoos implemented this adaptive noise parameter mechanism in NOVELTY+ to create the heuristic ADAPTNOVELTY+. In ADAPTNOVELTY+ values of $\Theta = 1/6$ and $\phi = 0.2$ were used.

Hoos then compared the performance of the ADAPTNOVELTY+ heuristic to the NOVELTY+ heuristic. To do this, a set of 3-SAT problem instances and SAT instances derived from other hard combinatorial problems were used to evaluate the performance of both heuristics. NOVELTY+ was ran on each problem instance several times using different parameter values until one was found that provided the best performance for that specific problem instance - in essence computationally expensive parameter tuning was performed. For each problem instance the performance of ADAPTNOVELTY+ was compared to the performance of NOVELTY+ using the tuned parameter value. While the results showed that the two heuristics had generally similar performance, overall the tuned NOVELTY+ heuristic outperformed ADAPTNOVELTY+. However, the authors noted that ADAPTNOVELTY+ required no computationally expensive tuning, and had much better performance than an un-tuned NOVELTY+ heuristic. A variant of ADAPTNOVELTY+, called R+ADAPTNOVELTY+, was entered in the Random track at the 2005 SAT Competition, where it placed 1st.

Adaptive parameter tuning would continue to be used in the creation of effective LS-SAT heuristics in the late 2000s. Li, Wei, and Zhang [108] presented research detailing attempts to improve on the G^2WSAT heuristic using this mechanism. Three new heuristics were proposed by the authors; ADAPTG^2WSAT, G^2WSAT$_P$ and ADAPTG^2WSAT$_P$. ADAPTG^2WSAT was identical to G^2WSAT, except that it used adaptive parameter tuning in its construction. G^2WSAT$_P$ was created by making two changes to the G^2WSAT heuristic. The first of these changes augmented the way in which a variable was chosen from the DECRVARS set. The second change substituted the original diversification strategy used, NOVELTY++, for a new strategy called NOVELTY+$_P$. ADAPTG^2WSAT$_P$ was an augmentation of G^2WSAT$_P$ that used adaptive parameter tuning in its construction. In testing the performance of these heuristics, Li, Wei, and Zhang found that the performance of the adaptive heuristics was comparable to their non-adaptive variants when used with a tuned parameter value.

These heuristics were entered in the Random track at the 2007 SAT Competition, where ADAPTG^2WSAT and ADAPTG^2WSAT$_P$ placed 3rd and 5th respectively. The heuristic ADAPTG^2WSAT0 was also entered, and placed 2nd. ADAPTG^2WSAT0

is nearly identical to ADAPTG^{2}WSAT, with the only difference being that its diversification and intensification strategies have been slightly altered. A further augmentation of ADAPTG^{2}WSAT0, called ADAPTG^{2}WSAT2009++, was also created. ADAPTG^{2}WSAT2009++ used the diversification strategy NOVELTY++. This was the diversification strategy originally used by G^{2}WSAT. ADAPTG^{2}WSAT2009++ was entered in the Random track at the 2009 SAT Competition, where it placed 3rd.

These adaptive heuristics, though they were shown to not perform as well as their non-adaptive variants with tuned parameter values, offered better overall performance on a range of problem instances. The heuristics presented in this subsection also show how the continued application of new ideas to previously described heuristics drove forward LS-SAT heuristic research, allowing better performing heuristics to be created. The use of adaptive parameters has continued to be used in LS-SAT heuristic design, including in state-of-the-art LS-SAT solvers such as those in the SPARROW [10] family (see Section 2.3.5).

2.3.4 Clause Weighting Schemes

In the previous subsections we have discussed intensification and diversification techniques, as well as adaptive parameter tuning mechanisms used in LS-SAT heuristic design. In this subsection we introduce another important mechanism used in modern LS-SAT heuristic design called clause weighting.

Dynamic local search (DLS) is a local search technique that involves "modify[ing] the evaluation function whenever a local optimum is encountered in such a way that further improvement steps become possible" [83, Chapter 2]. Clause weighting in SAT is a specific form of DLS. Selman and Kautz [159] (and independently Morris [131]) suggested techniques for enhancing GSAT through adding clause weighting.

Clause weighting has a simple premise. Each clause is assigned a numerical weight that can change as the overarching local search algorithm progresses. The weight w of a clause c is used to simulate there being w copies of c in the problem. By changing the topology of the problem in this way, certain behaviours can be encouraged without changing the overall satisfiability of the problem instance. For example, by having w copies of an unsatisfied clause c containing a variable v, v's POSGAIN$_1$ is increased. For heuristics that pick variables based on their POSGAIN$_1$ value, this could favour v over other variables, and potentially stop the algorithm getting stuck in cycles, or help it escape local optima.

Algorithm 2.10 GSAT+WEIGHTS Heuristic & Weight Update Function

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	W	Clause weights.
Output:		The variable to be flipped.

```
algorithm GSAT+WEIGHTS(F, A, W)
    v = ORDER-VARS([NETGAIN_W], VARS(F))
    return v[0]
algorithm UPDATE-WEIGHTS(F, A, W)
    for (c ∈ {cl ∈ CLAUSES(F), SATISFIED(A, cl) = False}) do
        W_c = W_c + 1
```

Selman and Kautz [159] created a variant of GSAT called GSAT+WEIGHTS. GSAT+WEIGHTS uses clause weighting and can be described as follows; on initialisation the weights of all clauses are set to 1. The NetGain_W of all variables under the current assignment A and weighting scheme W are computed. The variable with the greatest NetGain_W is chosen to flip. After each iteration, any clauses that are currently unsatisfied have their weight increased by 1. Pseudocode to show the heuristic and weight update function are presented in Algorithm 2.10. The weight update function is called at the end of each iteration of the overarching local search algorithm.

GSAT+WEIGHTS was found to be more effective at solving problem instances than its weightless counterpart GSAT. However, it was also shown to not be as effective as WALKSAT. Despite this, researchers continued to experiment with different strategies for utilising clause weighting in an attempt to create more effective heuristics. For example, the Discrete Lagrangian Method (DLM) [173] uses an additional mechanism to stop weights becoming too large and dominating the others. This mechanism was termed “smoothing” and is designed to ensure that every weight is within a small factor of the average of the weights. The SDF [173] heuristic uses a multiplicative, rather than an additive, expression to update its clause weights. The Exponentiated Sub-Gradient (ESG) method [157] is a variant of SDF which uses weight update criteria that is dependent on whether a local optima has been found - in essence it only sometimes updates the clause weights.

The weight smoothing mechanic used by the heuristics ESG and DLM is a

computationally expensive function that, in those heuristics, is invoked on every iteration of the local search algorithm. Some variable metrics use dynamic clause weighting in their formulation. A clause weight update function, such as that shown in Algorithm 2.10, is relatively computationally inexpensive to perform as it usually only changes the weights of a small number of clauses. Under these circumstances only a small number of variable metric values need to be updated. Generally it is less computationally expensive to update these using the difference between the old and the new weight when compared to re-computing the value. As a smoothing mechanism can change all the weights in a problem, it is preferable to re-compute every effected variable metric when smoothing weights. To be clear, a weight smoothing function is far more computationally expensive than a weight update function that only changes a small number of weights. Due to this, a heuristic that uses a smoothing mechanic which is invoked on every iteration of the overarching local search algorithm may reduce the overall effectiveness of the heuristic, as it cannot perform as many flips as other heuristics that do not use the same type of smoothing mechanism.

Hutter, Tompkins, and Hoos described the LS-SAT heuristic SAPS [86], a heuristic similar to ESG that uses a probabilistic smoothing mechanic. A probabilistic smoothing mechanic only has a small chance of performing the weight smoothing function on any given iteration. The SAPS heuristic can be described as follows; all variables that appear in an unsatisfied clause c in the problem F under an assignment A and weighting scheme W are ordered by their NetGain_W. If the variable with the largest NetGain_W in c is greater than 0 then it is chosen. Else, with probability wp a random variable from c is chosen, otherwise the weights are updated. The weight update function uses a multiplicative expression in its construction and the parameter P_{smooth} to decide how often to smooth the weights. Detailed pseudocode of SAPS's weight update function is shown in Algorithm 2.11. The authors suggested values of $\alpha = 1.3$, $p = 0.8$ and $P_{smooth} = 0.05$ as examples of parameters that provide good performance. The authors also presented an adaptive version of SAPS, called RSAPS. RSAPS uses adaptive parameter tuning to change its P_{smooth} parameter. The adaptive mechanism used in RSAPS is similar to that described in Section 2.3.3.

Hutter, Tompkins, and Hoos performed experiments comparing the performance of SAPS, RSAPS and ESG to one of the (then) state-of-the-art LS-SAT heuristics Novelty+. It was found that none of the heuristics based on clause weighting were able to match the performance of Novelty+, and it was concluded that heuristics

Algorithm 2.11 SAPS Weight Update Function

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	W	Weights.
	p	Smoothing factor.
	P_{smooth}	Smoothing probability.
	α	Scaling factor.
Output:		None.

algorithm UPDATE-WEIGHTS(F, A, W, p, P_{smooth}, α)
  for ($c \in \{cl \in$ CLAUSES(F), SATISFIED(A, cl) $= False\}$) **do**
    $W_c = W_c \times \alpha$
  if (WITH-PROBABILITY(P_{smooth})) **then**
    for ($c \in$ CLAUSES(F)) **do**
      $W_c = W_c \times p + (1 - p) \times \overline{W}$

based on clause weighting could still not compete with the then state-of-the-art LS-SAT solvers.

Thornton et al. [171] described a heuristic called PAWS that uses a clause weighting mechanism similar to that utilised in SAPS. PAWS uses an additive update expression, and a much simpler smoothing update mechanism. The weight update function for PAWS is shown in Algorithm 2.12. The authors found that PAWS was comparable in performance to SAPS, however when entered in the Random track at the SAT 2005 Competition, it placed lower than SAPS.

Researchers continued to experiment with clause weighting mechanisms in the creation of LS-SAT heuristics, and explore their use in augmenting previously described heuristics. This is perhaps best illustrated by the gNOVELTY+ heuristic, as described by Pham et al. [145]. gNOVELTY+ uses ideas from G^2WSAT, adaptNOVELTY+ and heuristics that use clause weighting mechanisms. gNOVELTY+ was entered in the Random track at the 2007 SAT Competition [143] where it placed 1st. gNOVELTY2 [144], a more efficient version of gNOVELTY+, was entered in the Random track at the 2008 SAT Competition, where it placed 2nd. We provide a general outline of the gNOVELTY+ heuristic in Section 2.3.5, and in Algorithm 2.13 we show the weight update function used in gNOVELTY+ [145]. The reader can see that it is similar to the PAWS weight update function, but uses the probabilistic smoothing mechanism

Algorithm 2.12 PAWS Weight Update Function

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	W	Weights.
	MAX_{inc}	Weight increase point.
	$nTimesWeightIncr$	Number of times weight has been increased.
Output:		None.

algorithm UPDATE-WEIGHTS(F, A, W, MAX_{inc}, $nTimesWeightIncr$)
  for ($c \in \{cl \in$ CLAUSES(F), SATISFIED(A, cl) $= False\}$) **do**
    $W_c = W_c + 1$
  $nTimesWeightIncr = nTimesWeightIncr + 1$
  if (($nTimesWeightIncr$ % MAX_{inc}) $= 0$) **then**
    for ($c \in$ CLAUSES(F)) **do**
      if ($W_c > 1$) **then**
        $W_c = W_c - 1$

originally described for SAPS.

Following the success of the GNOVELTY+ heuristic, many of the subsequently described heuristics use similar weight update functions to that shown in Algorithm 2.13. Examples of heuristics that have taken inspiration from this update function include SPARROW2011 (placed 1st in the Random track at the 2011 SAT Competition), SPARROW2RISS [11] (placed 1st in the Random track at the 2018 SAT Competition) and the BALANCEDZ solver [109] (placed 2nd in the Random track at the 2014 SAT Competition).

We want to be clear that the clause weighting mechanisms described in this subsection are not the only weighting mechanisms that have been used in the design of effective LS-SAT heuristics. For example Ishtaiwi et al. [87] introduced a clause weighting mechanism that swaps weights between clauses, whereas Prestwich [149] developed a separate mechanism where weights were assigned to variables rather than clauses, and the heuristic function chose variables based off these weights. An effective LS-SAT solver called TNM was described that used these two ideas together with ADAPTNOVELTY+ as a base for its heuristic component. TNM was entered in the Random track at the 2009 SAT Competition where it placed 1st.

Clause weighting has become an important technique used in modern LS-SAT

Algorithm 2.13 gNovelty+ Weight Update Function

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	W	Weights.
	sp	Smoothing probability.
Output:		None.

```
algorithm Update-Weights(F, A, W, sp)
    for (c ∈ {cl ∈ Clauses(F), Satisfied(A, cl) = False}) do
        W_c = W_c + 1
    if (With-Probability(sp)) then
        for (c ∈ Clauses(F)) do
            if (W_c > 1) then
                W_c = W_c − 1
```

solvers to diversify solutions, and help them to escape local optima. In conjunction with other techniques we have presented in previous subsections, it has been used to push forward the performance of LS-SAT solvers, and is still used in many modern-day LS-SAT heuristics.

2.3.5 Probability Distribution Heuristics

In the late 2000s, through the combination of ideas from WalkSAT, GSAT, adaptive parameter tuning and clause weighting, heuristics such as gNovelty+ were considered to be the state-of-the-art in LS-SAT heuristic design. A very basic outline of gNovelty+ can be given as:

1. Intensification using the G^2WSAT heuristic.
2. Diversification using the adaptNovelty heuristic.
3. Weight update function similar to that used by PAWS. Weights are only updated when the variable is chosen from the diversification strategy.

Balint and Fröhlich [10] observed that “One drawback of algorithms that use adaptNovelty+-like heuristics to escape from local minima is the lack of differentiation between the variables”. In essence Balint and Fröhlich were stating that, if an assignment A is considered a state of local optima, then whenever A is encountered it

Table 2.3: The set of functions used in the heuristics in Section 2.3.5. Each function takes a variable x as input and outputs some weight that is attributed to x. The *sparrow* function is used with the SPARROW heuristic, and the *exp*, *exp-break-only*, *poly* and *poly-break-only* functions are used with the PROBSAT heuristic. We also show example constant values that were given in the original descriptions of the heuristics, which were said to provide good performance. Where applicable, $\epsilon = 1$.

Function			Example Values
sparrow	=	$c_1^{\text{NETGAIN}_W(x)} \times (\frac{\text{AGE}(x)}{c_3})^{c_2} + 1$	$c_1 = 2, c_2 = 4,$ $c_3 = 10^5$
exp	=	$\frac{c_m^{\text{POSGAIN}_1(x)}}{c_b^{\text{NEGGAIN}_1(x)}}$	$c_b = 3.6, c_m = 0.5$
exp-break-only	=	$c_b^{-\text{NEGGAIN}_1(x)}$	$c_b = 3.6$
poly	=	$\frac{\text{POSGAIN}_1(x)^{c_m}}{\epsilon + \text{NEGGAIN}_1(x)^{c_b}}$	$c_b = 3.1, c_m = -0.8$
poly-break-only	=	$\epsilon + \text{NEGGAIN}_1(x)^{-c_b}$	$c_b = 2.3$

is likely that ADAPTNOVELTY+ will choose the same variable to flip. In turn, if this state of local optima is encountered more than once while the overarching local search algorithm is running, the algorithm may get stuck in a cycle and could be unable to find a satisfying solution. Though ADAPTNOVELTY+ is only one component of GNOVELTY+, this observation can still have a detrimental effect on GNOVELTY+'s performance.

Balint and Fröhlich developed a new heuristic in an attempt to tackle this shortcoming. The heuristic the authors designed is called the SPARROW heuristic. It is identical to the GNOVELTY+ heuristic, except that it uses a new diversification strategy. This strategy can be described as follows; a random broken clause c is chosen. Each of the variables in c have a weight attributed to them according to the *sparrow* function as shown in Table 2.3. A weighted pick is then performed on the variables to choose the variable to flip. In Algorithm 2.14 we show pseudocode that can be used to create this strategy by substituting f for *sparrow*.

The authors evaluated SPARROW against two of the best performing heuristics

Algorithm 2.14 ProbSAT Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	f	Function taking variable and returning weight.
Output:		The variable to be flipped.

```
algorithm ProbSAT(F, A, f)
    vs = Pick-Broken()
    VW = 0                                  ▷ Variable weights all set to 0.
    for (v ∈ vs) do
        VW_v = (f(v), v)
    return Pick-Weighted-Var(VW)
```

at the time, gNovelty2T and TNM. The three heuristics were tested on a set of instances from the Random track of the 2009 SAT Competition. It was found that Sparrow outperformed both heuristics. Later, a variant of Sparrow called Sparrow2011 was entered in the Random track at the 2011 SAT Competition, where it placed 1st. Sparrow is still considered to be a highly effective standalone heuristic, and variants of it continue to be used as components in state-of-the-art hybrid solvers. In Section 2.2.2 we discussed how Sparrow2Riss, a hybrid solver based partially on Sparrow, outperformed all other solvers in the Random track at the 2018 SAT Competition.

Balint and Schöning [12] described a heuristic called ProbSAT, which used a design that was inspired by Sparrow's diversification strategy. Rather than being contained in a G^2WSAT-like heuristic, ProbSAT works by choosing the variable to flip using only a weighted pick function. The authors presented four variants of ProbSAT. We can describe any of them using the pseudocode in Algorithm 2.14, by substituting f for one of the functions *exp*, *exp-break-only*, *poly* or *poly-break-only*, as shown in Table 2.3.

Of the four variants of the ProbSAT heuristic described by Balint and Schöning, two used an f function based around an exponential expression, and two used an f function based around a polynomial expression. In each pair, one f function used the NegGain_1 variable metric, and the other used both the NegGain_1 and PosGain_1 variable metrics. When testing the performance of the different variants of ProbSAT, the authors found those which only used the NegGain_1 variable metric

were able to complete a single iteration of local search more quickly than those which used both NegGain_1 and PosGain_1. This occurred as calculating two variable metrics is more computationally expensive than calculating one. The authors also found that there existed constants for each ProbSAT variant which provided the best performance. When these "tuned" variants were tested against each other, it was found that those which used NegGain_1 outperformed those which used both NegGain_1 and PosGain_1.

Balint and Schöning compared the performance of the four variants of the ProbSAT heuristic against each other, and against other state-of-the-art LS-SAT heuristics. The heuristics were ran on a set of problem instances used in the Random track of the 2011 SAT Competition. The results showed that all of the ProbSAT variants were highly effective heuristics when compared to other state-of-the-art LS-SAT heuristics. Of the four ProbSAT variants, *poly-break-only* performed the best.

A version of ProbSAT [13] placed 1st in the Random track at the 2013 SAT Competition, and a variant of it [14] optimised for parallel platforms won the Parallel Random track at the 2014 SAT Competition. In the Random track at the 2016 SAT Competition, an LS-SAT solver partially based off ProbSAT called Dimetheus [64] placed 1st.

Using a probability distribution to choose variables is a simple premise, but one that has been found to be highly effective at improving the performance of LS-SAT solvers. They have been used in augmenting previously described heuristics, and in the creation of new heuristics. Some modern-day, state-of-the-art LS-SAT solvers use a probability distribution as an underlying mechanism to drive their overarching search algorithms.

2.3.6 Configuration Checking

Configuration checking is a recently described technique that has been used to design intensification strategies in local search-based algorithms for solving hard combinatorial problems. It has been used to create heuristics which try to avoid flipping the same sequences of variables, a phenomena known as cycling [127]. The technique was originally described for the Minimum Vertex Cover problem [39], and recently several LS-SAT heuristics have been created that make use of it. Researchers have also used it to create heuristic strategies to solve the MAX-SAT problem [34].

Configuration checking is a mechanism that remembers the "circumstances" of

a problem when a variable is changed. By using this information, strategies can be created that only allow a variable to be changed when its circumstances have also changed. In SAT, researchers have proposed two configuration checking strategies; the neighbourhood variable configuration checking (NVCC) strategy and the clause state configuration checking (CSCC) strategy. The NVCC strategy makes use of a variable's *neighbourhood*, defined as follows:

Definition 17 (Neighbourhood of a Variable)
Given a SAT formula F and a variable $v \in \textsc{Vars}(F)$, the neighbourhood of v is the set of all other variables in $x \in \textsc{Vars}(F)$ that are contained in a clause that also contains v. It can be defined as $\{x \in \textsc{Vars}(F),\ x \neq v \wedge (\exists c \in \textsc{Clauses}(F),\ c \in \textsc{ClauseSet}(F, x) \wedge c \in \textsc{ClauseSet}(F, v))\}$. To refer to this set we write $N_v(v)$.

The NVCC and CSCC strategies can be visualised as metrics that attribute a boolean value to a variable. They can be described as follows:

- For a SAT problem F and variable $v \in \textsc{Vars}(F)$, the boolean variable represented by NVCC(v) is $True$ if any variables in $N_v(v)$ have had their assignment changed since v was last flipped.

- For a SAT problem F and variable $v \in \textsc{Vars}(F)$, the boolean variable represented by CSCC(v) is $True$ if any clause $c \in \textsc{ClauseSet}(F, v)$ has changed state - that is to say, gone from satisfied to unsatisfied or vice versa - since v was last flipped. As noted by Luo et al. [118], the set of variables whose CSCC value is set to $True$ is a subset of those variables whose NVCC value is set to $True$.

Cai and Su [38] described the first LS-SAT heuristic that utilised configuration checking. This heuristic, called SW_{CC}, is shown in Algorithm 2.15. SW_{CC} can be described as follows; the set $vars$ containing every variable v whose $\textsc{NetGain}_W(v) \geq 0$ and NVCC(v) $= True$ is generated. If $vars$ is non-empty, then the variable in $vars$ with the highest $\textsc{NetGain}_W$ is returned. Otherwise, the clause weights are updated, and the variable with the highest Age returned from a randomly chosen broken clause. Its design is similar to G^2WSAT, in that it moves to a state of local optima before employing a diversification strategy to move away from that state.

Cai and Su tested the SW_{CC} heuristic against the TNM heuristic on the instances used in the Random track at the 2009 SAT Competition, and against the Sparrow

Algorithm 2.15 SW_{CC} Heuristic

Input:	F	SAT problem instance in CNF.
	A	Current assignment.
	W	Weights.
Output:		The variable to be flipped.

```
algorithm SW_CC(F, A, W)
    vars = FILTER(VARS(F), (λv →NETGAIN_W(v) ≥ 0 ∧ NVCC(v) = True))
    if (vars ≠ ∅) then return ORDER-VARS([NETGAIN_W], vars)[0]
    else
        UPDATE-WEIGHTS()
        return ORDER-VARS([AGE], PICK-BROKEN())[0]
```

heuristic on the instances used in the Random track at the 2011 SAT Competition. While SW_{CC} outperformed TNM, it did not perform as well as SPARROW. Researchers continued to develop the SW_{CC} heuristic and subsequently designed a better performing version called SW_{CCA} [36]. Solvers based on the SW_{CCA} heuristic were entered in the Random track at the 2012 [32] and 2013 [78] SAT Competitions, where they placed 1st and 3rd respectively.

Researchers continued to experiment with different ways of utilising the NVCC strategy in LS-SAT heuristic design. Some used the SW_{CC} algorithm as a basic template, making small changes to it to create new heuristics. Two augmentations of SW_{CC} that are particularly relevant to our work are SWCCSubScore [37] and CScoreSAT [35]. These heuristics make use of the variable metric SUBNETGAIN, which can be described using the following definitions:

Definition 18 (Sub-Positive Gain)
For a SAT problem F, variable $v \in \text{VARS}(F)$, weighting scheme W and complete assignment A, this is a metric associated with a variable that represents the number of clauses that currently have 1 satisfied literal, and will have exactly 2 satisfied literals if v is flipped. It is also known as submakes. It can be computed as:

$$\sum_{c \,\in\, \text{FALSELITSET}(F,\, A,\, v)} \begin{cases} W_c & \text{if } \text{TRUELITS}(A,\ c) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

To refer to this value, we write $\text{SUBPOSGAIN}_W(A,\ F,\ v)$*. If the assignment and SAT formula are obvious from the context, we write* $\text{SUBPOSGAIN}_W(v)$*. If a set*

of variables are ordered according to their $\textsc{SubPosGain}_W$*, they are ordered from smallest to largest; the variable with the largest* $\textsc{SubPosGain}_W$ *is considered the best. A variable's* $\textsc{SubPosGain}_W$ *is always a positive integer.*

Definition 19 (Sub-Negative Gain)
For a SAT problem F*, variable* $v \in \textsc{Vars}(F)$*, weighting scheme* W *and complete assignment* A*, this is a metric associated with a variable representing the number of clauses that currently have* 2 *satisfied literals which will have* 1 *satisfied literal if* v *is flipped. It is also known as subbreaks. It can be computed as:*

$$\sum_{c \,\in\, \textsc{TrueLitSet}(A,\ v)} \begin{cases} W_c & \textit{if } \textsc{TrueLits}(A,\ c) = 2 \\ 0 & \textit{otherwise} \end{cases} \tag{2.5}$$

To refer to this value, we write $\textsc{SubNegGain}_W(A,\ F,\ v)$*. If the assignment and SAT formula are obvious from the context, we write* $\textsc{SubNegGain}_W(v)$*. If a set of variables are ordered according to their* $\textsc{SubNegGain}_W$*, they are ordered from largest to smallest; the variable with the smallest* $\textsc{SubNegGain}_W$ *is considered the best. A variable's* $\textsc{SubNegGain}_W$ *is always a positive integer.*

Definition 20 (Sub-Net Gain)
For a SAT problem F*, variable* $v \in \textsc{Vars}(F)$*, weighting scheme* W *and complete assignment* A*, this is a metric associated with a variable representing the total difference in number of clauses with exactly* 2 *satisfied literals if* v *is flipped. It is also known as subscore. It can be computed as:*

$$\textsc{SubPosGain}_W(A,\ F,\ v) - \textsc{SubNegGain}_W(A,\ F,\ v) \tag{2.6}$$

To refer to this value, we write $\textsc{SubNetGain}_W(A,\ F,\ v)$*. If the assignment and SAT formula are obvious from the context, we write* $\textsc{SubNetGain}_W(v)$*. If a set of variables are ordered according to their* $\textsc{SubNetGain}_W$*, they are ordered from smallest to largest; the variable with the largest* $\textsc{SubNetGain}_W$ *is considered the best. A variable's* $\textsc{SubNetGain}_W$ *can be a positive or negative integer.*

SWCCSubScore and CScoreSAT use the SubNetGain to differentiate between variables when a tie-break occurs. The authors found that both of these heuristics were comparable in performance to other configuration checking heuristics they were tested against. A SAT solver based off these strategies was entered in the Random track at the 2014 SAT Competition [33], where it placed 5th.

Though we do not present their construction here, there have been several effective LS-SAT heuristics described that are based off the CSCC strategy, such as SW_{QCC} [115] and FRwCb [119, 116]. Additionally, the DCCA solver utilises a heuristic strategy that uses both NVCC and CSCC in its formulation [118]. Highly effective LS-SAT heuristics were created from this avenue of research as well; a solver comprised of DCCA and FRwCb called CSCCSAT [117] placed 2nd at the 2016 SAT Competition.

Configuration checking, as a general technique, has been used to direct the intensification strategies of LS-SAT heuristics. Compared to other methods described in this section, it is a more complicated mechanism. Yet, when combined in the correct way with other techniques, it can be used to create highly effective LS-SAT solvers.

2.3.7 Summary & Discussion

Though we have provided a broad overview of LS-SAT solvers in this section, in truth there are many other techniques that have been used in the construction of LS-SAT solvers that have improved overall performance, not just heuristics. The choice of data structures and optimisations used can play a role in how effective an underlying LS-SAT heuristic is. For example, the solver PolyPower [31] is based on WalkSAT, but its heuristic is designed in an optimised way so as to improve its overall performance. These changes make it comparable to ProbSAT on some instances. Some solvers, such as Ranov [5] pre-process problem instances, which can have a positive effect on how well some LS-SAT heuristics perform. Additional techniques include utilising strategies from complete solvers such as unit clause elimination [81], the use of message passing frameworks [64], and evolutionary computation techniques such as genetic algorithms (GAs) [113].

We believe that we have given a detailed account of the many different techniques used by researchers to create LS-SAT heuristics. Much of the research presented here will be useful in understanding the systems built to represent LS-SAT heuristics, described in later chapters in this thesis.

2.4 Automated Creation of Heuristics

In Chapter 1 we discussed how heuristics can be an important component when designing effective algorithms to solve hard combinatorial problems, and in Section 2.3 we provided examples of hand-crafted heuristics that have been used to build effective local search algorithms to solve SAT. In this section we review the literature pertaining to the automated design of heuristics that are used to solve hard combinatorial problems. We specifically focus on research where the overarching goal was to automatically create new heuristics.

Some of the earliest work in the automated design of heuristics involved automatically selecting a heuristic from a given set of hand-crafted ones. For example, Fisher [56] and Crowston, Glover, Trawick, et al. [47] described techniques using probabilistic learning to automatically select heuristics for solving the job-shop scheduling problem. While the automated selection of heuristics has been an active part of research for over fifty years, historically comparatively less attention has been given to systems that are designed to automatically create new heuristics.

Recently, the term *hyper heuristics* has been used to classify research in the automated design of heuristics, as well as retroactively re-classify previously existing research. This term was first used to describe “heuristics to choose heuristics” [46], and recently defined by Gendreau and Potvin [65] as “an automated methodology for selecting or generating heuristics to solve computational search problems”. However, we feel that Burke et al.’s definition [28] of hyper heuristics as being “a set of approaches that are motivated by the goal of automating the design of heuristic methods to solve hard computational problems” best captures the intention behind hyper heuristic research.

In much of the literature concerning hyper heuristic research, the created systems are classified as one of two types. These are outlined as follows:

- Selective hyper heuristic. A methodology for choosing or selecting a heuristic.
- Generative hyper heuristic. A methodology for automatically creating new heuristics.

Using these definitions, the work in this thesis can be considered to be in the research area of generative hyper heuristics.

In the following subsections we review previous research whose aim was to automatically create heuristics - that is to say, research in the domain of generative hyper

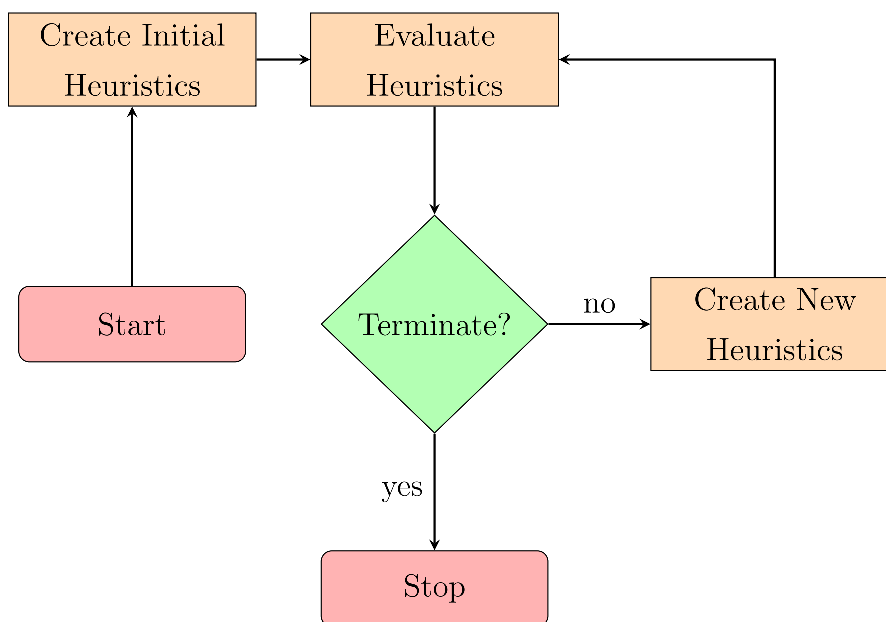


Figure 2.4: An overview of the control flow of a system designed to automate the creation of heuristics.

heuristics. In Section 2.4.1 we concentrate on research where the goal was to create heuristics that had a role in solving hard combinatorial problems, paying particular attention to the representation of heuristics and the techniques used to automatically create them. In Section 2.4.2 we focus on previous work in the automated creation of heuristics that are used as part of an LS-SAT solver - research that is of particular relevance to the work contained in this thesis. Finally in Section 2.4.3 we discuss how the research presented in this section relates to the overall body of work contained within this thesis.

2.4.1 Automated Creation of Heuristics for Hard Problems

Many systems designed to automatically create heuristics can be said to have a similar overarching design; an initial set of heuristics are created, the heuristics are then evaluated against some problem instances and a fitness value extrapolated from those results. The system then decides if some pre-determined termination criteria has been met. If it has not, then the system creates a new set of heuristics which are evaluated, and the process continues until the termination criteria is met. In Figure 2.4 we show a visualisation of this design.

When designing a system to automatically create heuristics, the way in which

the heuristic is represented, and how this representation will be used as part of an overarching algorithm to solve a hard combinatorial problem, must be taken into consideration. The techniques used to create new heuristics and the heuristic representation are also intrinsically linked; the choice of representation can dictate which creation methods are viable for that domain. There is generally no obvious, natural representation of a heuristic for a specific problem. Some ways of representing a heuristic are problem-specific, while others are general enough to be reused in other domains. In turn, generalised representations can be used with generalised methods to create new heuristics.

Some of the simplest representations imagine a heuristic as an assignment problem, whereby through some post-processing this representation is converted into a heuristic function. For example, Özcan and Parkes [138] described a system that automatically creates heuristics for the online bin packing problem. The authors represented heuristics as fixed-size two-dimensional matrices containing numerical data. An element's two indices correspond to the bin sizes remaining and the size of the element to be inserted. The matrices were used to choose which bins to put items into. When given an item of size b, the matrix was used as a lookup table, with the column of data for that b value used to assign a numerical value to each bin. The item was assigned to the bin with the highest numerical value. The matrices themselves were created using a simple GA.

One disadvantage to this approach is that the created heuristics are not general-purpose online bin packing heuristics - they are designed for bin packing problems with a fixed number of bins, meaning that new heuristics would have to be created for different sized problems. Yet the authors found this technique to be effective for those instances it was tested on.

It is more common in automated heuristic creation to use a representation that closely mirrors a programming language, as this is how heuristics are created by software engineers. The resulting heuristics can be generalised for arbitrarily sized problems. One commonly used representation is a list-based (or array-based) structure that simulates an imperative programming language, or a set of machine code instructions. Poli, Woodward, and Burke [148] represented a heuristic for the offline bin packing problem as a set of domain-specific machine instructions. The created program fragments represented a function which was used as part of a wider heuristic strategy. The authors used linear genetic programming (GP) [21] to create

new heuristics. This is a general-purpose program synthesis technique specifically designed for use with this type of program representation. Keller and Poli [98] also used an array-based representation with linear GP to create heuristics for the TSP. Unlike the work of Poli, Woodward, and Burke, the created heuristics were designed to be the entirety of the heuristic function, and not just a component of it.

Perhaps the most popular [26] representation technique used when creating heuristics is a tree-based representation, emulating a functional paradigm of programming. In such a representation, each node contains a term from a (usually) domain-specific language (DSL). Together they are used to describe heuristics, or heuristic components, for a specific problem domain. One advantage that tree-based representations have over array-based representations is that they can potentially be of unbounded size. Should an effective heuristic exist that is represented by a large tree, it is possible to create it using a tree-based representation. It also has an effect on the search space of heuristics that can be described using a specific language, as it can potentially become infinite.

There have been several types of tree-based representation used in automated heuristic creation, the simplest of which we term the "untyped" tree representation. These are tree-based representations used in conjunction with a DSL that has no additional rules regarding the composition of terms within that language. It is the most commonly used form of representation within systems that automatically create heuristics. For example, Burke, Hyde, and Kendall [27, 25, 150] used this representation to create heuristics for use with the online bin packing problem. The created heuristics were represented as arithmetic expressions, which were then used to assign numerical values to items. These numerical values were analysed to determine which bin to insert the item into. As the internal DSL only contained arithmetic terms, all of the terms having the same "type" was a valid choice for that domain. The authors used untyped "Koza-style" GP [102] to create the heuristics. Within automated heuristic creation, this way of representing and method of creating heuristics has historically been widely used, and continues to be used today. There are recent examples of similar research in the job-shop scheduling problem [174, 141], the multi-skill resource constrained project scheduling problem [110, 88, 82], resource constrained scheduling problems [40], in online resource allocation [170], bin packing [3] and in solving bi-level optimisation problems [101].

In an untyped language like that used by Privosnik, there are no specific rules

regarding which terms can be used as the arguments to other terms. That is to say, any term in the language can be used as the argument to another. However, this does not model some types of programming language very well. As a general example, a string cannot be used as an argument to a function that requires an integer. For some heuristic domains with more complicated control structures or typing, an untyped representation is not appropriate, as it may allow ill-formed heuristics to be created. To alleviate this, some DSLs are defined in ways that describe what terms can be used as the arguments to others. In automated heuristic creation, perhaps the most common way of defining a language which can prohibit ill-formed programs is through a context-free grammar (CFG). Examples of research which used a CFG to automatically create heuristics include work by Sosa-Ascencio et al. [165], who used a CFG with grammar-based GP [102, 125] to create heuristics to solve CSPs, Sabar et al. [153], who used a CFG with gene expression programming [55] to develop a generalised system to create heuristics for several hard combinatorial problems, and Fajfar, Bürmen, and Puhan [53], who used a CFG with grammatical evolution [152] to create heuristics for solving real valued optimisation problems.

Typically in the literature associated with heuristic creation, CFGs are used to simulate a strong type system. However, the program synthesis methods designed to work on CFGs are distinctly separate from those program synthesis methods designed to work on languages described in terms of a type system. We discuss this relationship in further detail in Section 2.5.2.

In this subsection we have presented various examples of research where the goal was to automatically create heuristics to solve combinatorial problems. We have discussed several different examples of the types of heuristic representation used, and the methods used to create heuristics for these representations. In the next subsection, we discuss examples of work where the goal was to create heuristics for an LS-SAT solver.

2.4.2 Automated Design of LS-SAT Heuristics

In this subsection we provide an overview of research in the automated creation of LS-SAT heuristics. This work is directly relevant to this thesis, as this is the domain that we will be creating heuristics for. The observations from the work examined in this subsection have a direct effect on the design choices for our experiments, the way we evaluate our created heuristics, and the way we represent heuristics.

$$
\begin{array}{lcl}
H & = & \texttt{IfRandLt}\ \ prob\ \ H\ \ H \\
 & | & \texttt{IfVarCompare}\ \ cmp\ \ gt\ \ H\ \ H \\
 & | & \texttt{IfVarCond}\ \ cmp\ \ gt\ \ int\ \ H\ \ H \\
 & | & \texttt{GetOldestVar}\ \ H\ \ H \\
 & | & \texttt{IfTabu}\ \ age\ \ H\ \ H \\
 & | & \texttt{IfNotMinAge}\ \ varset\ \ H\ \ H \\
 & | & \texttt{GetBestVar}\ \ varset\ \ gt \\
 & | & \texttt{GetBestVarSnd}\ \ varset\ \ gt \\
 & | & \texttt{GetBestVar2}\ \ varset\ \ gt\ \ gt \\
 & | & \texttt{PickRandomVar}\ \ varset \\
prob & = & \in \{0.0 \ldots 1.0\} \\
varset & = & \texttt{RBC-0} \mid \texttt{RBC-1} \mid \texttt{WFF} \\
gt & = & \texttt{NetGain} \mid \texttt{NegGain} \mid \texttt{PosGain} \\
age & = & \in \mathbb{N} \\
int & = & \in \mathbb{N} \\
cmp & = & < \mid \leq \mid =
\end{array}
$$

Figure 2.5: The language used by Fukunaga [60, 61, 63] to automatically create LS-SAT heuristics. Fukunaga used a type system when describing this language, however we present it as a CFG. There is no difference between the set of heuristics that can be created using this CFG and Fukunaga's original language. Note that some names of terms differ to those used in the original work, to align with the terminology we use in this thesis.

Fukunaga [60, 63, 61], in a series of papers, described research concerning algorithms used to automatically create LS-SAT heuristics. In this work, the heuristics were encoded using a tree-based structure, with an associated type system used to prohibit ill-formed heuristics. The created heuristics were designed to be general-purpose LS-SAT heuristics, and required no post-processing to be used for their intended domain. We show the language Fukunaga used in Figure 2.5, presented as a CFG for brevity.

The design of Fukunaga's language was inspired by previously described LS-SAT heuristics. In relation to our work, the heuristics which inspired Fukunaga's language are all of those described in Section 2.3.1 and a portion of those in Section 2.3.2 (up to NOVELTY). The author noted of the terms used in their language that "all of these

primitives were proposed in the literature by 1993, shortly after the introduction of GSAT" [60], and that the "history of SAT local search algorithms shows that significant advances do not require the invention of entirely new "ideas" – discovering a new combination of existing building blocks has resulted in some of the best known SAT local search algorithms" [61]. We made a similar observation in Section 2.3, where we provided evidence that some effective heuristics use components from previously described LS-SAT heuristics. The "building blocks" - that is to say, the metrics and control structures - were described separately, and it took several years of research before they were combined into the examples of effective heuristics seen. Since Fukunaga's work was published, and from our analysis in Section 2.3, it could be argued that this trend has continued; new metrics and control structures have been described for LS-SAT heuristics, yet expert knowledge was required to combine these ideas to create the most effective heuristics.

The fitness function used by Fukunaga to score the created heuristics was designed as follows; the heuristics were ran as part of an LS-SAT solver on two sets of problem instances. The second set contained larger problem instances than the first. The fitness function had an early termination mechanism built into it. This allowed it to terminate early if it was found that a heuristic could not solve many of the problem instances. This design choice was made to reduce the overall running time, as evaluating a heuristic is computationally expensive.

The heuristic creation technique used by Fukunaga was a bespoke GP algorithm which used a steady-state GP model. It also used a domain-specific crossover operator, which combined previously created heuristics with the functions `IfRandLt`, `IfVarCompare`, `IfVarCond`, `GetOldestVar`, `IfTabu` and `IfNotMinAge` to create the next set of heuristics for consideration. Fukunaga stated, regarding his bespoke GP algorithm, that "we currently lack principled, analytical meta-heuristics that can be used to guide a systematic meta-level search algorithm". Therefore, this is why the author used "a population-based search algorithm to search for good variable selection heuristics". Fukunaga also noted that some attempts were made with conventional GP, however these were unsuccessful. In Figure 3.2h we show an example of one of the heuristics created from this work.

Like Fukunaga, Bader-El-Den and Poli [9] performed experiments using evolutionary computation to create LS-SAT heuristics. However those authors used standard grammar-based GP in their work. While Fukunaga's goal was to create generalised

heuristics for LS-SAT, Bader-El-Den and Poli's goal was to design a system that could create an effective heuristic for solving fixed-sized 3-SAT problem instances near the phase transition region. However, the size of the problems the heuristics were trained on was relatively small, containing at most $1,000$ variables.

The authors noted of the heuristics created from their work that "individuals representing GSAT, HSAT and GWSAT were created ... in almost all experiments we did". Further to this, the authors stated that "GP was always able to eventually discover new and better heuristics". This work suggests that a standard GP algorithm is a viable technique for creating heuristics in this domain.

Some researchers experimented with using other techniques to automatically create LS-SAT heuristics. KhudaBukhsh et al. [100, 99] developed SATenstein, a system designed to create LS-SAT heuristics using automated algorithmic configuration. In that work, a "skeleton" of a heuristic was described, with specific terms and functions in the skeleton left blank. By using parameter tuning, these uninitialised functions and terms were filled, and heuristics created from the skeleton. However, this meant that compared to the work of Fukunaga, Bader-El-Den and Poli, the set of possible heuristics that could be created was finite. The heuristics created from this work were highly effective, and could compete with modern heuristics such as Sparrow. However, this research sits slightly outside the scope of our own; we intend to use program synthesis to create LS-SAT heuristics, rather than algorithmic configuration techniques. This work can also be considered a successor to SATZilla [175, 176], a SAT solver that selects a heuristic based on an analysis of the provided SAT problem instance.

In this subsection we have provided an overview of the pertinent research concerning the automated creation of LS-SAT heuristics. Though there are few examples of research in this area, the existence of such work allows us to contextualise the research in this thesis, and to draw inspiration from previously performed experiments when designing our own.

2.4.3 Summary & Discussion

In this section we have provided an overview of research in the area of automated heuristic creation. We have introduced several different types of heuristic representation, as well as discussed the associated techniques used to create heuristics. In Section 2.4.2 we focused on work with a similar goal to our own - that is, the

automated creation of LS-SAT heuristics.

From the research presented in this section, we can draw several conclusions that directly affect the trajectory of our research. In Section 2.3 we described several LS-SAT heuristics, and presented pseudocode for some of those described. We presented the pseudocode in an imperative programming style, which has traditionally been the way in which heuristics for LS-SAT have been created. This would suggest that a heuristic representation which mirrors this style, together with any associated heuristic creation techniques, would be a viable area to concentrate our research in when looking for alternative methods of program synthesis.

However, the research presented in Section 2.4.2 almost exclusively used tree-based representations for the heuristics created for LS-SAT. Specifically, those examples use languages that prohibit the combination of certain terms in the created heuristics. In the examples shown, this is done through a CFG and a type system. The work presented in that subsection, particularly that by Fukunaga, showed us that representing heuristics as program trees is a viable representation technique, and that GP can be used to create effective heuristics under this representation. This work also showed us that, by using a language that prohibits the combination of certain terms, complicated control structures can be safely combined to create new heuristics.

For these reasons, we choose to pursue a similar avenue to these previous researchers, and use a tree-based representation together with some constraints on the underlying language that prohibit the combination of certain terms. This in turn directs our research in the next section, which will look at program synthesis methods designed to operate on this representation. We also note that the early termination mechanism used in the fitness function in Fukunaga's work may be a useful technique for our research. Evaluating heuristics can be computationally expensive, and using such a methodology could help reduce the overall time spent evaluating heuristics that do not perform well.

This section only serves as an introduction to systems that automate the creation of heuristics, and hyper heuristic research at large. For more information regarding hyper heuristics, we direct the reader to the following resources; "Hyper-Heuristics: An emerging direction in modern search technology" [29] presents an early introduction to hyper heuristic research. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques* [23, Chapter 20] contextualises

the potential benefits of hyper heuristics, as well as providing detailed examples of previous research in both generative and selective hyper heuristics. "Hyper-heuristics: A survey of the state of the art" [28] provides an analysis on the (then) state-of-the-art in hyper heuristic research, and *Handbook of Metaheuristics* [65, Chapter 14] gives a modern overview of current research trends in hyper heuristics.

2.5 Program Synthesis

Program synthesis, otherwise known as synthetic programming or automated programming, refers to the task of automatically finding or creating programs that satisfy some user defined criteria [76]. It is a fragmented discipline, with areas of research in artificial intelligence [120, 132, 57, 121], programming theory [154, 94, 95, 22] and evolutionary computation [102, 152, 55]. Though program synthesis has been an active research area for over fifty years [120, 121], it is only relatively recently that, through a renewed focus, program synthesis techniques have been deployed in real-world applications.

In Section 2.4 we provided examples of how heuristic creation techniques have been used to automatically create heuristics for solving hard combinatorial problems. We believe that all of the techniques described in that section can be considered program synthesis techniques.

Automated heuristic creation is not the only domain that has made use of program synthesis. For example within data modelling, there is a continued need for functions that can transform data from one format to another - such as when normalising the textual content of strings to a consistent format. To the end user of a data modelling system with little experience of programming, it would not be easy to create such a function. Program synthesis techniques have been used to automatically create functions for these types of simple data modelling problems. End users provide examples of input and expected output of the required function, and the program synthesizer creates a function that meets this specification. FlashFill is a program synthesizer, provided as part of the Microsoft Excel program, that is able to create simple functions from user provided input-output data [73, 75].

Another example of the emerging use of program synthesis in the real-world is found in automated patch generation [133, 48, 112, 106]. Automated patch generators are systems that are designed to automatically fix incorrect software. When a defect

is detected in a piece of software, perhaps through the use of an automated testing suite, an automated patch generator can detect which parts of the software may have caused the defect, and can use program synthesis techniques to apply changes to these parts with the aim of producing a "patch" for the software. One example of a program synthesis technique used in automated patch generation is GP [106].

Researchers have also made use of program synthesis in other areas of computer science. Within programming theory, it has been used to create superoptimising compilers [154] and in the automated completion of computer code [180]. In graphic design, program synthesis has been used to aid in the completion of structured drawings [41]. We point the reader to several resources detailing other applications of program synthesis. Gulwani et al. [77] provided an overview of the real-world use-cases of inductive programming, a form of program synthesis. Koza [103] conducted a survey on areas in which GP has created programs that are competitive with human designed ones. Finally, Gulwani, Polozov, and Singh [76] gave a detailed account of various use-cases where program synthesis has been applied successfully.

Some readers may not be as familiar with the term "program synthesis" as they are with evolutionary computation techniques to create programs like GP. To be clear, GP is one of many program synthesis methods.

The format of this section is as follows; in Section 2.5.1 we discuss the key components of a program synthesis system. In Section 2.5.2 we detail the type of program representation that we work with in this thesis - a tree-based program representation. In Section 2.5.3 we discuss methods that navigate the search space of programs in a methodical manner, and in Section 2.5.4 we provide an overview of GP, a method of program synthesis realised through the sampling of the search space. Finally in Section 2.5.5 we present the discussions and conclusions from the research presented in this section.

2.5.1 Preliminaries

Due to the various research areas within computer science with a vested interest in program synthesis, there are several competing definitions of how a program synthesis problem is described. Gulwani [74] suggested three characteristics of all program synthesis systems. These are the user intent, the solution space, and the search technique. We describe what is meant by these terms below.

$$\begin{aligned} div(i,j), rem(i,j) \quad \Leftarrow \quad & find(y,z) \\ \text{such that} \quad & i = y \times j + z \wedge 0 \leq z \wedge z < j \\ \text{where} \quad & 0 \leq i \wedge 0 < j \end{aligned}$$

Figure 2.6: Example of a program specification written in predicate logic. It can be read as follows; from the inputs i and j, find a program that returns two outputs y and z which adhere to the logical predicates shown. This specification can be satisfied by a program that performs integer division, returning the quotient and remainder as output.

User Intent

This is the specification of the program to be created, or the criteria by which a created program is judged. Problem specifications can differ greatly depending on the domain they are to be used in, and some search techniques are only compatible with certain types of specification. Some of the earliest work [120] in program synthesis described the specification of the required program as a logical statement in predicate logic. An example of this is shown in Figure 2.6. With such a specification, the goal of a program synthesizer is to find a program that would provably have the required behaviour. This type of specification is not just limited to domains where the goal is to find a simple function like that shown in Figure 2.6, it can also be used to describe the requirements of recursive programs, as well as the desired behaviour of data structures. One major disadvantage of this type of specification is that it requires a great deal of expert knowledge to write correct logical statements that express the requirements of the program. It also requires systems with the ability to automatically reason about the created programs, which in turn requires additional expert knowledge.

Another method of defining a program's specification is through the use of input-output examples, also known as Programming by Example (PbE). The specification is expressed as a set of pairs, where each pair contains an example of input and expected output. The examples can also be expressed as a function representing a subset of possible input, and a property of the desired output. FlashFill uses PbE to allow its users to describe the required program's behaviour in this manner. PbE, like deductive programming, has also been used to learn recursive programs [59, 167] using a search technique called inductive programming. Specifications of programs

expressed using PbE require a lower threshold of expertise compared to those formal methods we discussed previously. However when using such a system, the desired program may not be created due to discrepancies in the specification - for example, incorrectly described output, or there not being enough examples provided to reliably describe the intended program's behaviour.

Both of these specification types are best used for problems where there is a clear definition of the correct behaviour a program should produce. However, in some domains there may be no clear description of desired behaviour - and therefore no binary test that can be constructed to determine the success of the created programs. A fitness function is an alternative method for measuring a candidate program's effectiveness that can be useful in domains where success is difficult to measure. It is typically realised through performing some experiment with the created programs and quantifying their performance as some numerical value. This type of specification is useful in domains where there is no "correct" or "best" program that can be created, instead it is only known that some programs are more effective than others. An example of specification criteria that utilises a fitness function can be found in the research discussed in Section 2.4.2. This work was primarily focused on automated heuristic creation for solving SAT through local search. To determine a heuristic's effectiveness, it was ran as part of a local search algorithm on a set of problem instances, and a numerical value computed from how many instances it solved.

Solution Space

A candidate solution in a program synthesis problem is an instance of a created program, and is defined by some abstract structure. This structure is called the problem's program representation. The set of all possible candidate solutions, also called the solution space or search space, is defined by this representation, and the characteristics of how the representation is defined describe the size and shape of the search space. Some representations of programs may describe an infinite search space, or search spaces that grow more quickly than others. The program representation is usually inspired by some real-world programming language or paradigm. There are always two distinct components of a program representation; the language - conceptually a set of unique identifiers - and the underlying data structure used to contain and compose elements of this language.

The definition of a language can be domain-specific to the problem that is to be

tackled, or it can be abstract enough to be used to solve multiple problems. The language is designed by a human expert, and should contain constructs that it is believed will allow the program synthesizer to solve the required problem. A language can contain general-purpose programmatic primitives, or can be constrained to a domain-specific language. The specific elements inside a language are usually chosen carefully; the language should be expressive enough to provide some flexibility in the possible solutions, yet not too flexible so as to make the search space difficult to navigate.

The choice of data structure that is used to represent candidate programs can have a direct effect on which search techniques can be used in the program synthesizer. Tree data structures are a commonly used underlying structure for program representation as they can provide a high-level of generalisation. For example, they can emulate both imperative and functional programming styles. Many search techniques in program synthesis are designed with a tree-based representation in mind. However, for most languages, when used with this type of representation, the search space is infinite. This can make it difficult to explore effectively. Graphs [128], lists and arrays [79] are other examples of data structure that have been used as a component in a program synthesizer.

Some languages come with an additional parameter that describes which composition of terms in a candidate program are considered to be valid. One example used with a tree-based representation would be a type system, designed to emulate a strongly typed programming language. Another would be a CFG, designed to emulate an imperative programming language.

Search Technique

A search technique in a program synthesizer is the methodology used to explore the search space of possible programs. Succinctly, its function is to create new candidate programs. Whether a search technique is appropriate for a program synthesizer can depend on the type of program representation and user specification used. Using the work of Gulwani, Polozov, and Singh [76, Chapter 1] as a basis, we separate the available search techniques into three broad categories. The reader should note that these categories are not distinct as some search techniques can be viewed as belonging to multiple categories. The categories are:

- **Direct Search Techniques:** These methods visualise the search space of programs as a search tree, and directly work on this representation of the solution space. Like algorithms for solving assignment problems, some search techniques work on complete solutions (complete programs), and some work on partial solutions (partial programs). The search techniques that work on partial programs require some ability to reason logically about them. By doing so, there exists the potential for additional search strategies to be employed. An example is deductive programming [120, 121] that, by reasoning about partial programs, is able to recognise program equivalence and prune search branches [120, 58]. One particularly effective (and perhaps surprising) search technique used in inductive programming [22] is the enumeration of the search space to find candidate program trees. More advanced techniques such as Monte Carlo tree search [95] have also been used to explore the solution space of programs.

- **Constraint Solving:** Methods that use constraint solving in program synthesis require the ability to describe the "specification and the syntactic program restrictions in a single formula so that any true model corresponds to a correct program" [76]. These types of search techniques require a formal specification of a program's requirements, as well as formal logic pertaining to the effects that a program or partial program has on the input parameters. Through the use of some external logic deduction technique, the model can be solved, and the correct program extracted. Examples exist where program synthesis is conducted in this manner using automated theorem provers [120] and Satisfiability modulo theories (SMT) solvers [4].

- **Stochastic Methods:** Stochastic methods refer to search techniques that sample the search space of programs and, through this sampling, attempt to learn the distribution of the search space and use this information to find candidate programs. Examples include GP [102], machine learning [126] and grammatical evolution [152].

In the next subsection, we formally define the program tree representation used in this thesis. The reasoning behind the use of this representation was laid out in Section 2.4.3. In Sections 2.5.3 and 2.5.4 we give a broad overview of two search techniques designed to work with this representation.

2.5.2 Tree-based Program Representation

Some of the first work in program synthesis (such as in deductive programming [121] and GP [102]) used a tree structure as the basis for their program's representation. In this early work, the program representation's language component was given as a simple set of identifiers. These identifiers, or *terms*, were split into two sets; the non-terminal set and terminal set. The only rules these representations had regarding term placement was that non-terminals had to inhabit the nodes in a tree, and terminals the leaves. This representation closely resembles a functional programming style approach, where the non-terminals were analogous to functions, and terminals to constants. A consequence of this representation is that any terminal or non-terminal should be able to be used as an argument to a non-terminal, an assumption those in the GP community call closure. This is the same untyped style of language discussed in Section 2.4.1.

However, this assumption of closure can make it difficult to design languages that, at least conceptually, have more than one data type. As noted by Montana [130]; "forcing a problem which uses multiple data types to fit the closure constraint can severely and unnecessarily hurt the performance of genetic programming on that problem". For example, in a language that uses an integer for all its arguments, it may be difficult to add an "if" statement non-terminal that requires a boolean as an argument. When running the created programs, there needs to be an understanding of how to convert an arbitrary integer to a boolean which, in the context of the domain the program is deployed in, may be unnatural.

In a similar way to how type systems were introduced in early programming languages to ensure that programs were correctly typed before running them, researchers began to introduce program representations that prohibited the combination of certain terms in the created programs. For example, strongly typed GP (STGP) [130] uses an associated type system that is designed in such a way as to prohibit ill-formed programs. STGP is designed with a monomorphic type system in mind that prohibits currying. It is this kind of type system that we will be using in this thesis in tandem with our language to describe LS-SAT heuristics.

As an example, Figure 2.7a contains the Language EX-1, which is described in our desired form. The reader can clearly see that each term has an associated type signature. We describe the type system of a given language as having several characteristics, which are described as follows:

- A set of principle types pt. Conceptually this is a set of unique symbols. This set describes all the identifiers used in the type signatures in the language.
- A type signature t is a non-empty list of size n containing principle types $\{t_1 \dots t_n\}$. The return type of t is t_n. The argument type of t is $\{t_1 \dots t_{n-1}\}$. The argument type can potentially be empty. The arity of t is $n - 1$. A type signature is typically written as $t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n$.
- A set of unique terms L. Each term $e \in L$ has an associated type signature.
- A tree structure where each node is labelled with a term $\in L$ is valid under L if the following rules hold for all nodes in the tree. For a given node a containing the term e that has the type signature containing principle types $\{t_1 \dots t_m\}$, and where a has a collection of children $\{c_1 \dots c_n\}$, the criteria are; firstly, $n = m - 1$. Secondly, for each child $c_i \in \{c_1 \dots c_n\}$, the return type of the term in c_i must equal t_i. The return type of the whole tree structure is the return type of the root node.

Two examples of trees written in Language EX-1 are shown in Figure 2.8. One adheres to the type rules in Figure 2.7a, and one does not. We show an algorithm to check that a program tree is valid under a language in Algorithm 2.16. This algorithm performs a process known as type checking.

A language designed without a type system - that is to say, an untyped language - can be augmented to have one by introducing a single dummy type d. Each terminal is given the type d, and each function with arity n is given a type signature of $d_1 \rightarrow \cdots \rightarrow d_n \rightarrow d_{n+1}$. The reader may also note that CFGs can be formulated in such a way as to emulate a language that uses this kind of type system [125]. As an example, Language EX-1 is shown as a CFG in Figure 2.7b.

Though not the goal of this subsection, we have introduced several notions used within the functional programming community, specifically concerning type systems. Program synthesis techniques have also been formulated for languages with more complicated type systems; for example, lambda calculus [177] and System F [22]. For more information regarding the theory of type systems and functional programming languages, we refer the reader to *The Implementation of Functional Programming Languages* [90].

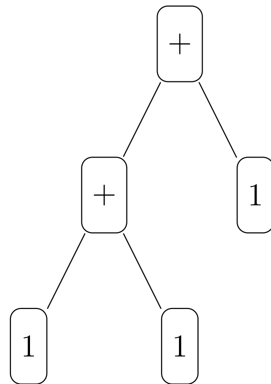
Term	Type Signature
0	`Int`
1	`Int`
$+$	`Int` $\rightarrow$ `Int` $\rightarrow$ `Int`
$-$	`Int` $\rightarrow$ `Int` $\rightarrow$ `Int`
`negate`	`Int` $\rightarrow$ `Int`
`coinFlip`	`Bool`
`intIf`	`Bool` $\rightarrow$ `Int` $\rightarrow$ `Int` $\rightarrow$ `Int`
`lessThan`	`Int` $\rightarrow$ `Int` $\rightarrow$ `Bool`

(a) The set of terms in Language EX-1. Each term is annotated with its type signature. The principle types of the language are {`Bool`, `Int`}.

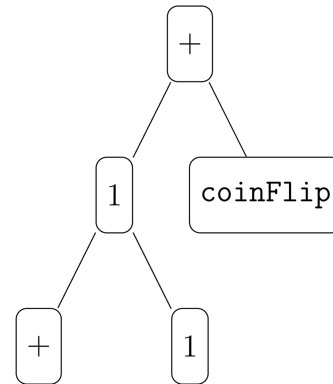
$$
\begin{aligned}
E &= \texttt{intIf}\ B\ E_1\ E_2 \\
&\mid\ +\ E_1\ E_2 \\
&\mid\ -\ E_1\ E_2 \\
&\mid\ \texttt{negate}\ E \\
&\mid\ 0 \\
&\mid\ 1 \\
B &= \texttt{coinFlip} \\
&\mid\ \texttt{lessThan}\ E_1\ E_2
\end{aligned}
$$

(b) Language EX-1, described in terms of a CFG.

Figure 2.7: The Language EX-1. We show it in two forms; the first as a list of terms, each with an associated type signature. The second as a CFG. Both forms of the language describe the same set of programs.



(a) An example of a program tree written using Language EX-1 that adheres to the rules of the language.



(b) An example of a program tree written using Language EX-1 that does not adhere to the rules of the language. Specifically, the left node under the root has a term requiring no arguments, yet it has two. The right term under the root has a return type of type `Bool`, but the root requires a term with an `Int` type from its second argument.

Figure 2.8: Two examples of program trees written using Language EX-1, shown in Figure 2.7. One is correct according to the rules of the language, one is not.

Algorithm 2.16 TYPE-CHECK Algorithm

Input:
- L — The language. Conceptually a map of terms to their associated type signature, which is represented as a vector.
- *tree* — A tree with an unbounded number of children. Each node is labelled with a term from L. It is this tree that the algorithm will type check.
- t — The required type to check against.

Output: *True* if the *tree* has the required type t, *False* otherwise.

```
algorithm TYPE-CHECK(L, tree, t)
    returnType = TYPE-INFERENCE(L, tree)
    if ((returnType = null) ∨ (returnType ≠ t)) then return False
    else return True

algorithm TYPE-INFERENCE(L, tree)
    termsType = L.AT(t)                          ▷ Finds the type of the node of the tree.
    children = tree.CHILDREN()
    if (termsType.SIZE() − 1 ≠ children.SIZE()) then return null
    if (children.SIZE() > 0) then
        for (i ∈ {0 ... termsType.SIZE() − 2}) do
            returnType = TYPE-INFERENCE(L, children[i])
            if ((returnType = null) ∨ (returnType ≠ termsType[i])) then
                return null
    return termsType[termsType.SIZE() − 1]       ▷ Returns the last index.
```

2.5.3 Direct Search Techniques

In this subsection we review the literature pertaining to direct search techniques for program synthesis. By direct search techniques, we specifically refer to those methods that imagine the search space of programs as a search tree, and directly navigate through it to find candidate solutions - that is, potential programs that are solutions to the overarching program synthesis problem.

To illustrate the methods described in this subsection, we will use Language EX-1, described in Figure 2.7. In Figure 2.9 we show a small portion of the search space described by Language EX-1. Since Language EX-1 describes a search space containing an infinite number of programs, it is not possible to show the complete search tree. However, the reader can clearly see that some small, full programs inhabit leaves in the search tree, and partial programs inhabit the nodes.

The search space, when visualised in this manner, shows us some interesting properties that may not be immediately obvious. We can see that there are distinct repeating patterns in the search tree. Specifically, smaller complete programs are reused in the formulation of larger ones. A simple example of this can be seen by noting that the program trees representing 0 and 1 are reused at deeper levels in the search tree as arguments to +. This pattern continues in the unseen portions of the search tree; the complete program tree of $+ \{0,\ 1\}$ will eventually be re-generated and reused as a child node in other complete program trees.

The examples of partial programs in the search tree introduce us to the concept of the *type hole*. A type hole is a node in a program tree that is not instantiated with a term. For the program to be correct according to the type rules (or the rules enforced by a CFG), it requires a term. In the program trees in Figure 2.9, type holes are shown using a $\star$.

Like other types of search trees, breadth-first search (BFS) and depth-first search (DFS) are the two most obvious ways of traversing this tree. We present a unified search algorithm, called Top-Down-Search, that is able to traverse this tree in either a BFS or DFS manner. It is shown in Algorithm 2.17. It is inspired by an identically named algorithm described by Gulwani, Polozov, and Singh [76]. Whereas that algorithm was designed for languages that use a CFG, ours is designed for languages that use a type system. Another difference between the two algorithms is in how they instantiate type holes; ours instantiates them one at a time, whereas Gulwani, Polozov, and Singh's algorithm instantiates them all at once. The node

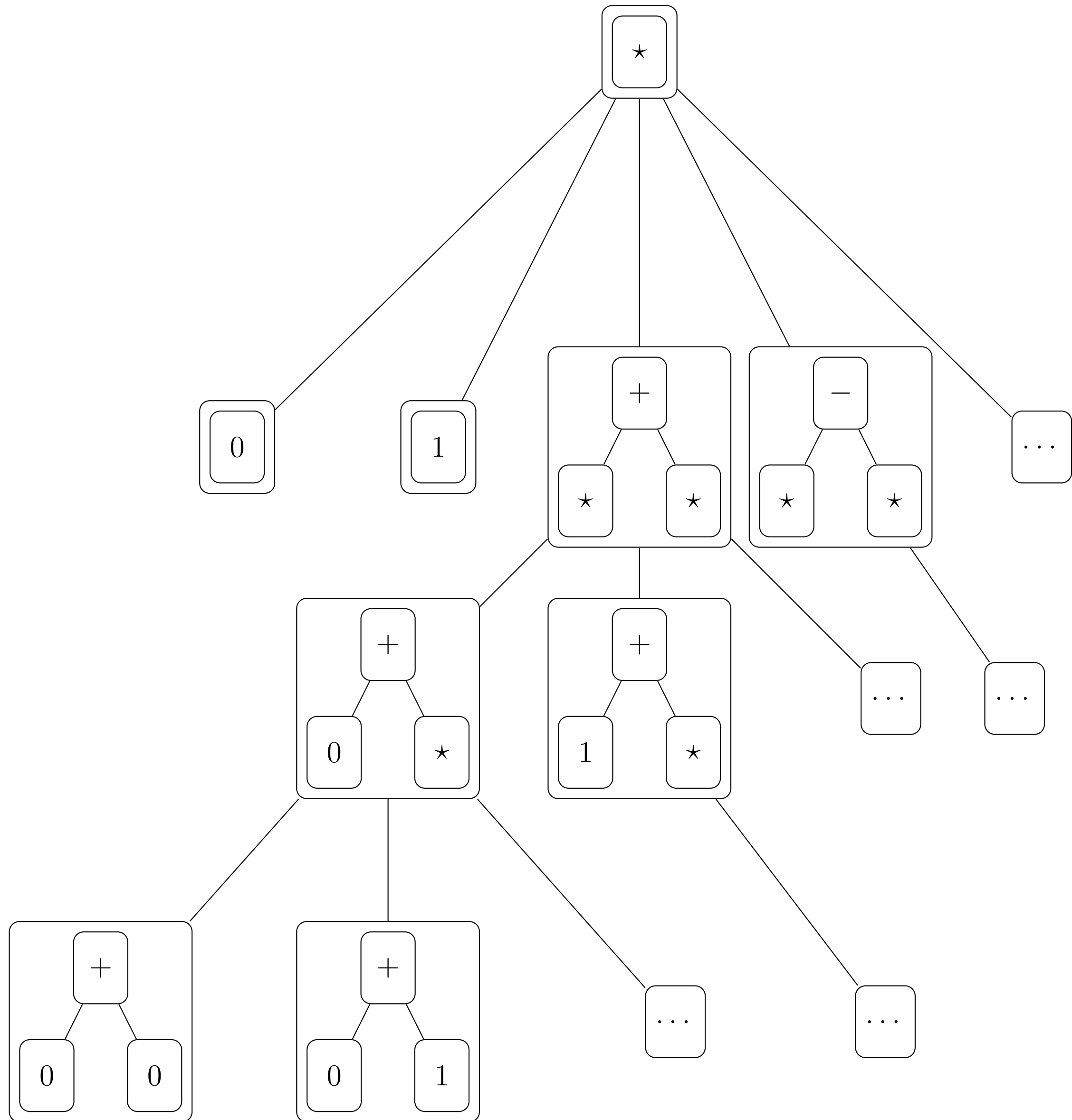


Figure 2.9: An example of a search tree for the set of programs in Language EX-1, shown in Figure 2.7. Each node in the tree contains a tree, representing either a partial or complete program tree. The complete program trees can be found in the leaves. The nodes with dots in them represent areas of the search space not visualised - as this is an infinite search space, there is no way to visualise the whole search tree. In each program tree, nodes which have no argument yet are given a placeholder, denoted by $\star$.

Algorithm 2.17 TOP-DOWN-SEARCH Algorithm

Input:
- L The language. Represented as a mapping of principle types t to a set of terms that have t as a return type.
- ϕ Specification criteria.
- r Required type of the program tree to be created.

Output: A program that satisfies ϕ.

```
algorithm TOP-DOWN-SEARCH(L, φ, r)
    P = [⋆]                                          ▷ List of tree nodes to search.
    P_v = {⋆}                                        ▷ Set of partial programs seen.
    while (P.SIZE() > 0) do
        p = P.POP()
        if (φ(p)) then return p                      ▷ Required program found.
        α = FIND-FIRST-TYPE-HOLE(p)                  ▷ Find type hole.
        t = FIND-REQUIRED-TYPE-HOLE(α, r)            ▷ Find required type of α.
        for (b ∈ L.AT(t)) do
            p′ = p[α → b]                            ▷ Replace α with b.
            if (¬SUBSUMED(P_v, p′, φ)) then
                P.INSERT(p′)
                P_v = P_v.INSERT(p′)
```

which we insert a new term into is the first type hole that is found when conducting an in-order traversal of the partial program tree.

TOP-DOWN-SEARCH works by maintaining a list of nodes P in the search tree that have yet to be explored, and searches them one by one to create successor program trees. On initialisation, the initial program tree ($\star$) is added to this list, and then the algorithm proceeds through a while loop. On each iteration, if P is non-empty, then an element p is removed from it. A check is performed to ascertain whether p passes the specification. If not, then the first (if there are any) type hole α in p is found by performing an in-order traversal of p, the required type t of α deduced and the language L probed for all terms that satisfy t. New partial programs are created by inserting any terms in the language that satisfy t into α's position, and the new partial programs are then checked to see whether they have been subsumed. If not, they are inserted into P.

A program - either partial or full - is subsumed if, for whatever reason, the

algorithm has determined that it should not be added to P. In essence it is pruned from the search space. Pruning criteria may be determined by partial program equivalence; as an example, a partial program in the form $+\ \{0,\ \alpha\}$ will generate the same set of programs as α, and a system with enough information about the domain could prune that program. Or, the system may be able to determine that two programs in the form $+\ \{\alpha,\ \beta\}$ and $+\ \{\beta,\ \alpha\}$ are equivalent, and remove one of them.

Where successor states are inserted into P changes how TOP-DOWN-SEARCH operates. If the new elements are inserted at the front of the list, the algorithm becomes a DFS. If they are inserted at the end of the list, the algorithm becomes a BFS. The reader should note that, for either method of insertion, there would have to be some limiting factor on how large the program trees could be, or the algorithm would not be guaranteed to terminate.

In Algorithm 2.18 we present BOTTOM-UP-SEARCH. BOTTOM-UP-SEARCH can create the same set of program trees that TOP-DOWN-SEARCH can, however it works in a fundamentally different way. BOTTOM-UP-SEARCH is an augmented version of an identically named algorithm presented by Gulwani, Polozov, and Singh [76]. The version of BOTTOM-UP-SEARCH presented in Algorithm 2.18 is designed to work on languages that are represented using a type system, whereas that presented by Gulwani, Polozov, and Singh is designed for use with CFGs.

The BOTTOM-UP-SEARCH algorithm uses a function called ENUMERATE-PROGRAMS. ENUMERATE-PROGRAMS should, when given an integer i, return all correctly typed full programs of exactly size i. It does this through recursion, and making use of memoized results. The overarching BOTTOM-UP-SEARCH algorithm works as follows; for each program size i from 1 to the maximum bound m, the programs are enumerated for i using ENUMERATE-PROGRAMS. After that set of programs ps has been enumerated, each program $p \in ps$ is checked to see whether it passes the specification criteria. If it does, then p is returned and BOTTOM-UP-SEARCH terminates. Otherwise, if p is not equivalent to any program in E, it is added to E. The E set is then used by ENUMERATE-PROGRAMS when it is next invoked, so that previously created programs can be used in the creation of new ones. This is done to reduce the overall computation time. As BOTTOM-UP-SEARCH only operates on full programs, it is unable to check for the equivalence of partial programs.

Both the TOP-DOWN-SEARCH and BOTTOM-UP-SEARCH algorithms have their

Algorithm 2.18 BOTTOM-UP-SEARCH Algorithm

Input:	L	The language. Represented as a mapping of principle types t in the language to sets of terms that have t as a return type.
	ϕ	Specification criteria.
	m	Maximum size of program.
Output:		A program that satisfies ϕ.

```
algorithm BOTTOM-UP-SEARCH(L, φ, m)
    E = []                                          ▷ Memoized results.
    for (i ∈ {1...m}) do
        ps = ENUMERATE-PROGRAMS(E, L, i)
        for (p ∈ ps) do
            if (φ(p)) then return p
            if (¬E.EQUIVALENT(p)) then
                E.INSERT(p)
```

advantages and disadvantages; TOP-DOWN-SEARCH requires less memory as results are not memoized, however it is generally more computationally expensive. On the other hand, BOTTOM-UP-SEARCH can be memory intensive, but much faster at returning results. BOTTOM-UP-SEARCH's ability to prune the search space is less powerful when compared to TOP-DOWN-SEARCH, as it does not operate on partial programs.

Some research [96, 22, 1] has found that a simple enumeration of the search space can be used to perform program synthesis. Instead of pruning parts of the search tree or using heuristics to direct the search, all valid programs are found and returned. They are then tested to check whether they meet the specification criteria. However, this method can be computationally expensive.

Another example of direct tree search that has proven to be effective is bi-directional search. TOP-DOWN-SEARCH can be described as a forward search - setting the first term in a program first - and BOTTOM-UP-SEARCH can be described as a backwards search, as it first finds the smallest subtrees that will be inserted at the bottom of the synthesised program tree. Bi-directional search utilises these two algorithms working together, and has proven to be effective in certain domains [75, 146]. A further example of an effective search technique employed in program synthesis is Monte Carlo tree search, which has been used to sample portions of the

search space. Through this sampling, the overarching algorithm is then guided to areas of the search space where it believes programs exist which meet the problem specification [95].

2.5.4 Genetic Programming

GP is a program synthesis technique inspired by natural evolution [102]. Unlike the program synthesis techniques we looked at in the previous subsection, it is a population-based, sampling algorithm. GP is closely related to the GA. Whereas a GA operates on fixed-sized strings, GP operates on solutions that represent programs as trees. In this subsection we provide an overview of GP.

In Algorithm 2.19 we show the basic design of a GP algorithm. It can be described as follows; at the start of the algorithm, an initial population of candidate solutions are constructed, and each solution in that population tested against the specification criteria. The algorithm then proceeds in an iterative manner; it creates a new population from the previous one through the use of genetic operators, and tests this new population against the specification criteria. The termination criteria is a problem dependant mechanism that denotes when the algorithm terminates. Some examples of commonly used termination criteria include those that are satisfied when a set number of iterations have been performed, and those that are satisfied when a solution has been created that sufficiently meets the specification criteria.

The algorithms we discussed in the previous subsection are designed with a generic specification criteria in mind. GP differs from these, as it is usually associated with program synthesis problems whose specification is defined in terms of a fitness function. A fitness function attributes a value to each candidate solution that signifies how well it meets the specification. Some of the mechanisms used in GP require the fitness value of a program to operate correctly. However, GP can be used with program synthesis problems whose candidate solutions either pass or fail their associated specification [124]. In cases where the specification has been defined in this manner, a fitness value can be extracted from how well a candidate program performs - for example, if a specification was given as a set of input-output examples, and a candidate program could only pass the specification if it satisfied all the examples, then a fitness value for that program could be extrapolated from how many of the examples it satisfied.

In its original description, GP was described in terms of an untyped language [102]. The associated language was split into a terminal set and a function set. Elements

Algorithm 2.19 GENETIC-PROGRAMMING Algorithm

Input: L Language.
ϕ Specification criteria.
r Required type of the program trees to be created.
p Parameters to GP algorithm.

Output: A program that satisfies ϕ.

```
algorithm GENETIC-PROGRAMMING(L, φ, r, p)
    nextPop = INITIALISE(L, r, p)
    pop = EVALUATE-POPULATION(nextPop, φ)
    for (e ∈ pop) do
        if (φ(e)) then return e            ▷ An individual passes the specification.
    while (¬TERMINATION-CRITERIA-MET(p)) do
        nextPop = CREATE-NEW-POPULATION(L, pop, r)
        pop = EVALUATE-POPULATION(nextPop, φ)
        for (e ∈ pop) do
            if (φ(e)) then return e
    return pop.BEST()                      ▷ Returns the best individual found.
```

in the function set could only inhabit nodes in a program tree, and elements in the terminal set could only inhabit the leaves. We retain the use of these definitions to remain consistent with the literature.

In the remaining part of this subsection we describe the types of genetic operators, initialisation functions and selection functions commonly used in GP. In this thesis our focus is on languages that use an associated type system. We pay particular attention to any additional constraints or mechanisms that must be taken into account when using STGP [130], the GP variant designed for such languages.

Initialisation

There are three commonly referenced methods of initialising a population of candidate solutions in GP. All of them work with a parameter d, which refers to the depth of a node in a tree. The "full" method works by initially choosing a random function from the language, then working recursively to populate the required children. Only when the depth of the node being assigned $= d$ can a terminal be chosen. The "grow" method works similarly, except that at any point a terminal can be chosen. A terminal must be chosen when the depth of the node being assigned $= d$. Finally, the "ramped half-and-half" method works by populating half the population with the grow method, and the other half with the full method. The created programs from the full method are balanced trees, whereas those built using the grow method may be unbalanced.

In STGP, when choosing terminals and functions, the initialisation method must take care to only choose language terms that will create trees that are type correct. Sometimes, it is not possible to create trees in this manner, depending on the terms in the language and the types that need to be fulfilled. Therefore, the user must be aware of the formulation of their language when choosing which initialisation function to use, to ensure that the desired effect is achieved.

Selection Function

Genetic operators are used to create the next population in a GP algorithm. Some genetic operators use individuals from the previous population in their formulation, and a selection function is used to pick these individuals. There are many different selection functions described in the literature. Objectively, the three most commonly used are random selection, proportional selection and tournament selection. Random

selection is simple, as it just chooses an individual from the previous population at random. Proportional selection assigns each individual a weight based on its fitness value, then chooses one using a weighted pick function. In tournament selection, a set number of individuals are chosen at random from the previous population, and the best of these according to their fitness is chosen as the selected individual.

Genetic Operators

On each iteration of the GP algorithm, a new population is created through the use of genetic operators. Most genetic operators work by applying some algorithm to a set of program trees from the previous generation to create a new set of program trees. The selection function (described above) is used to select these individuals.

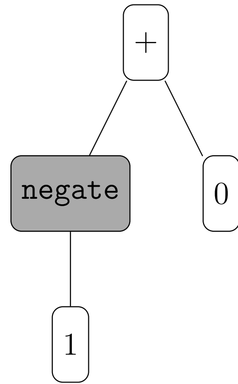
The four most commonly used genetic operators are crossover, mutation, reproduction and elitism. They can be described as follows:

Crossover In standard crossover, two individuals are selected and they are recombined to create two new individuals. To create the two new individuals, a point in each program tree is selected. The two new individuals are created by swapping the subtrees rooted at the selected points with each other. Figure 2.10 shows an example of how this is performed.

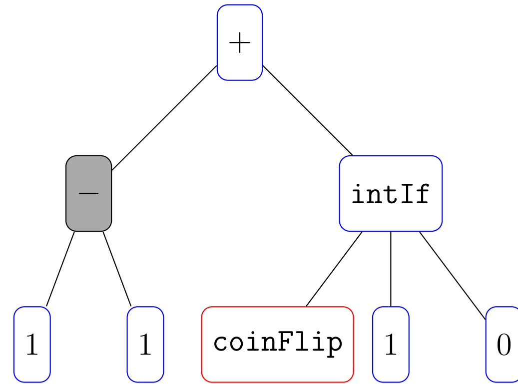
In STGP, additional care must be taken to ensure that the created offspring are type correct. This is achieved by, after picking the point in the first tree, ensuring that the subtree selected in the second program has the same return type as the subtree selected in the first.

Mutation The mutation operator uses one program tree from the previous population, and creates a single new program tree. A node is chosen at random in the original program tree, it is removed and a new subtree is created in its place. We show an example of this in Figure 2.11. This operator is primarily used to ensure that new genetic material is inserted into the population, as sometimes a population can be dominated by many similar individuals.

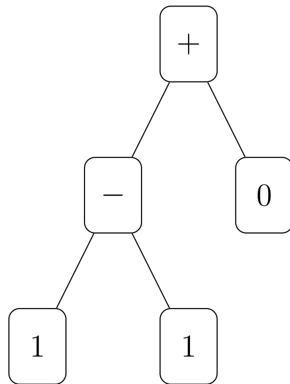
Reproduction Reproduction refers to a mechanism that creates new program trees using the initialisation function. To be clear, it requires no individuals from the previous population. It is designed to insert new genetic material into the population.

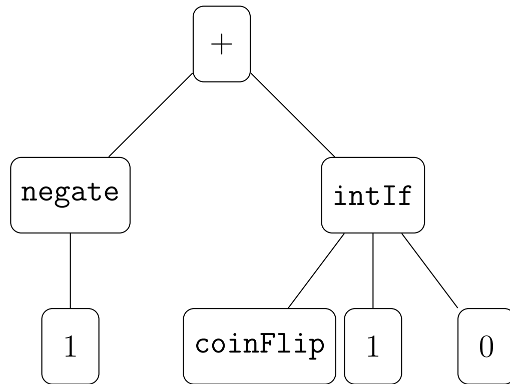


(a) An example program tree selected for crossover. The subtree selected as the crossover point is highlighted in grey.



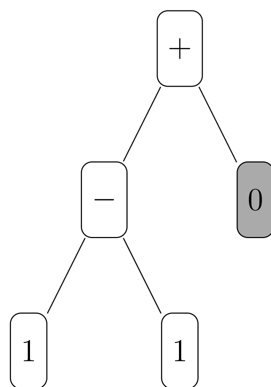
(b) A second example program tree that has been selected for crossover, with the subtree selected as the crossover point highlighted in grey. Nodes highlighted blue could have been selected to be a crossover point. Nodes highlighted red could not.



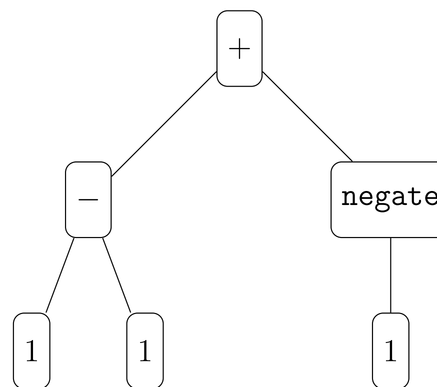


(c) The program trees created from an application of crossover using the trees in Figures 2.10a and 2.10b.

Figure 2.10: An example of the crossover operator being applied to two trees written in Language EX-1. In the input trees in Figures 2.10a and 2.10b the crossover point is indicated by the grey labelling of a node. To be clear, the subtrees being substituted are those rooted at the highlighted nodes.



(a) An example program tree selected for mutation. The subtree that will be removed and replaced is highlighted in grey.



(b) The program tree created from an application of mutation. The highlighted subtree in Figure 2.11a has been removed and a new subtree created and inserted in its place.

Figure 2.11: An example of the mutation operator being applied to a program tree written in Language EX-1.

Elitism Elitism refers to when an element is chosen from the previous generation to be copied to the new one. It is common for this operator to be used to retain one or more of the best individuals from the previous generation, to ensure that good program trees are not lost from generation to generation.

This subsection does not aim to be a complete guide to GP, only providing a brief introduction to its concepts. For more information regarding GP, as well as an overview of other closely related techniques such as gene expression programming [55], and grammatical evolution [152], we refer the reader to *A Field Guide to Genetic Programming* [147].

2.5.5 Summary & Discussion

In this section we have discussed program synthesis from a high-level perspective, then highlighted two specific examples of techniques that create programs automatically - direct search techniques and GP. Throughout the proceeding subsections, we have provided the reader with additional resources to each of the methods that we have focused on. We point the reader to “Program synthesis” [76], which we used as a basis for our descriptions in Section 2.5.1, for a general overview of program synthesis.

In the context of our work in LS-SAT, and in the wider research area of combinatorial problems, we note several similarities between program synthesis problems and hard combinatorial problems. The way in which a hard combinatorial problem is usually defined is in terms of either an optimisation problem or a decision problem. This categorisation mirrors the different types of specification criteria of a program synthesis problem; some program synthesis problems are defined in terms of a specific expected behaviour of the created program, and the specification given as a binary criteria. Others use a fitness function to gauge the created program's effectiveness.

From the major characteristics of a program synthesizer laid out in Section 2.5.1, it is clear that several of the given methodologies do not apply to the creation of heuristics. As far as we are aware, there is no "correct" specification of a heuristic in the domain of LS-SAT solvers or indeed any hard combinatorial problem. Or more specifically, any function that could correctly direct the search to the exact sequence of changes to move to a satisfying solution would be, to our knowledge, exponential in its running time. Therefore, like the previous work in creating LS-SAT heuristics [60, 62, 61, 9], it is reasonable to assume that the specification criteria for our candidate programs (or heuristics) should be a fitness measure of how well they perform on actual problem instances.

We noted in Section 2.4.2 that GP had previously been used to create LS-SAT heuristics, and its inclusion in this section is due to that research, as it is one of the previously used methods in our domain. Concerning the direct search techniques discussed in Section 2.5.3, we believe that some of them may be applicable to our research - specifically techniques that navigate the search tree of programs, and strategies for pruning parts of the search space. However, while creating a system that can reason about partial heuristics and prune parts of the search space is possible, it would require a high-level of expertise. This is exacerbated by the stochasticity employed by LS-SAT heuristics, as such a system would be reasoning about stochastic partial programs. On the other hand, exhaustive enumeration appears to be a practical technique to use as it requires relatively little expert knowledge, and may also provide us with insight into alternative methods that can be used to effectively navigate the search space of heuristics.

2.6 Minimum Tree Edit Distance Problem

In this section we give a broad overview of the minimum tree edit distance problem. We provide details concerning how it is defined, and present an efficient algorithm to solve it. The MTED problem discussed in this section is used extensively in Chapters 5 to 7.

The format of this section is as follows; in Section 2.6.1 we provide a definition of the MTED problem. In Section 2.6.1 we present an efficient algorithm used to solve it. Finally in Section 2.6.3 we present the conclusions to this section.

2.6.1 Definition

Originally described by Tai [168], and also known as the tree-to-tree problem, the minimum tree edit distance problem can be described as follows; given an alphabet Σ, a cost function γ and two ordered, labelled trees t_1 and t_2 (where each label in each tree is an element of Σ), find the minimum cost of tree edits, as given by γ, to transform t_1 to t_2. A tree edit can be one of the following:

- **Relabel**(l_1, l_2): A node in the tree has its label changed from l_1 to l_2. We write $\gamma(l_1 \rightarrow l_2)$ to show the cost of a relabel.

- **Insert**(l, i, j): A new node l is inserted into the tree. Specifically, it is inserted at some point in the tree under a previously existing node n, at position i in the sequence of n's children and taking a subsequence of n's children (from point i onward, and of size j) as its own. If n had k children originally described as $\{n_1 \dots n_k\}$, then after insertion, n now has $(k-j)+1$ children in the form $\{n_1 \dots n_{i-1}, l, n_{i+j}, \dots, n_{k-j+1}\}$. l's j children are in the form $\{n_i \dots n_{i+j}\}$. We write $\gamma(\rightarrow l)$ to show the cost of an insertion. If no nodes exist in the tree, this edit inserts the node l as the root of the new tree.

- **Delete**(l, i): A node l is removed from the tree. Specifically, if the ith child from the parent node n, which originally had k children $\{n_1 \dots n_k\}$, is to be deleted, then the sequence of l's m children $\{l_1 \dots l_m\}$ are inserted at position i under node n. n now has $(k+m)-1$ children in the form $\{n_1 \dots n_{i-1}, l_1 \dots l_m, n_{i+1} \dots n_k\}$. We write $\gamma(l \rightarrow)$ to show the cost of a deletion.

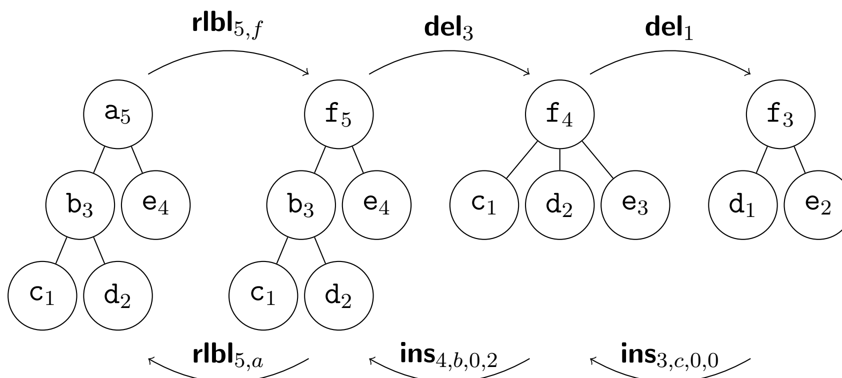


Figure 2.12: Examples of tree edits between trees. Each node in each tree is labelled with its index according to a post-order traversal. Relabels are shortened to "rlbl", and have two additional arguments pertaining to the index of the node to be relabelled and the new label for the node. Deletions are shortened to "del" and give the index of the node to delete. Insertions are shortened to "ins" and have four additional arguments. The first is the index of the node that will become the inserted node's parent. We designate this node e. The second argument is the node to be inserted, the third is the index of e's children that the inserted node will become, and the fourth is the number of e's children the inserted node will take as its children.

For any two trees, there are many possible edit sequences that can transform one tree to the other. A trivial method would be removing all the nodes from the initial tree t_1, then inserting all the required nodes to create the tree t_2. The goal of the MTED problem is to find the edit sequence with the minimum cost according to γ. In Figure 2.12 we show some examples of tree edits, to familiarise the reader with the concept.

MTED algorithms have found uses in several areas, including in the analysis of GP [134] and bioinformatics [2], and within database design [7]. The reader should note that the edits laid out above are not the only way of describing a tree edit and, by extension, the MTED problem. Other tree edit definitions exist; for example, those described by Lu [114], where a different set of moves are used to edit trees. In this thesis, we only consider the "classic" MTED problem for ordered trees, and not any extensions of it.

$$\delta(\theta, \theta) = 0 \tag{2.7}$$

$$\delta(F_1, \theta) = \delta(F_1 - v, \theta) + \gamma(v \rightarrow) \tag{2.8}$$

$$\delta(\theta, F_2) = \delta(\theta, F_2 - w) + \gamma(\rightarrow w) \tag{2.9}$$

$$\delta(F_1, F_2) = \min \begin{cases} \delta(F_1 - v, F_2) + \gamma(v \rightarrow) & (2.10) \\ \delta(F_1, F_2 - w) + \gamma(\rightarrow w) & (2.11) \\ \text{if } F_1 \wedge F_2 \text{ are trees} & \\ \quad \delta(F_1 - v, F_2 - w) + \gamma(v \rightarrow w) & (2.12) \\ \text{otherwise} & \\ \quad \delta(F_1(v), F_2(w)) + \delta(F_1 - T_1(v), F_2 - T_2(w)) & (2.13) \end{cases}$$

Figure 2.13: A recursive solution to the MTED problem, designed to operate on forests. We write $F - x$ to denote deleting the node x in forest F. v and w are the rightmost roots (if any) in F_1 and F_2 respectively. We use $F(x)$ as notation for obtaining the rightmost tree x from F. We write $F - T(x)$ to mean removing the entire rightmost tree x from F. θ is used as a synonym for the empty forest.

2.6.2 Algorithm

The MTED problem for (ordered) trees can be solved in polynomial time. For trees with m and n nodes, the best known MTED algorithm [51] has a time complexity of $\mathcal{O}(n^2 m(1 + \log \frac{m}{n}))$. However, the details of that algorithm are far beyond the scope of this thesis, and instead in this subsection we present a dynamic programming (DP) solution that has $\mathcal{O}(m^2 n^2)$ time complexity [181]. The algorithms presented in this subsection follow from the recursive equations shown in Figure 2.13.

The reader should note that in this section, all edits (insertions, relabels and deletions) have a cost of 1, except the edit which relabels a node with itself, which has a cost of 0. For the MTED to remain consistent, the cost function must have several properties. We refer the reader to work by Spears [139], which showed that the cost function must adhere to the non-negativity property, the self-equality property, the discernibility property, the symmetry property and the triangular inequality property. The authors also noted that if the cost function has these properties, then the tree edit distance metric defined from the cost function also has these properties.

The recursive solution outlined in Figure 2.13 is designed to work on forests rather

than trees. A forest is a collection of trees, and a tree can be thought of as a forest containing a single element. Therefore, this solution is consistent with the definition of the MTED problem outlined in Section 2.6.1.

The recursive solution in Figure 2.13 works by deconstructing the forest into subforests and single nodes, and then computing the distance between these deconstructed parts. When single nodes are encountered, the cost function γ is used to compute the distance. This process continues until the base-case is reached in Equation (2.7).

The reader should note that three distinct deconstructions can be identified in the recursive solution shown in Figure 2.13; one that is constructed from the input forest with the rightmost root node removed (used in Equations (2.8) to (2.12), and denoted by $F - x$) and two that are created by splitting a forest containing $k + 1$ trees into a forest containing the first k trees, and a singleton containing the last tree, (used in Equation (2.13) and denoted by $F(x)$ and $F - T(x)$ respectively).

To provide some intuition as to how these subforests are created, consider the two trees in Figure 2.14, which are labelled with indices according to a post-order traversal. When a forest is labelled in this manner, the subforest constructed by removing the rightmost root node can be created by removing the node with the highest index. For the two subforests that are required by Equation (2.13), these can be created by splitting the input forest into the last tree in the forest, and the remaining elements. Using a post-order indexing of a forest containing $k + 1$ trees, this can be abstractly represented as, for a forest with n nodes, there being an index $m < n$ such that the nodes with indices $1 \dots m$ are in the subforest representing the first k trees, and the nodes $m + 1 \dots n$ are in the forest representing the singleton tree.

One important point concerning these equations, and specifically about the splitting of the forest in Equation (2.13), is that the rightmost rooted singleton tree will always be a subtree of the original input tree given to the problem.

Though these equations could be followed through from the original input trees to find the MTED between them, the time complexity of such an algorithm would be exponential. Instead, a DP solution exists that allows the MTED to be computed in $\mathcal{O}(m^2n^2)$ time complexity. The pseudocode for this DP solution is shown in Algorithms 2.20 and 2.21.

(a) Tree et_1. (b) Tree et_2.

Figure 2.14: Two trees that are used in Section 2.6.2 to illustrate how the MTED algorithm works. In each node in each tree we mark the index according to a post-order traversal.

Algorithm 2.20 MTED Algorithm

Input: t_1 Input tree.
t_2 Output tree.
γ Cost function.
Output: Number representing minimum edit distance between t_1 and t_2.

algorithm MTED(t_1, t_2, γ)
  k_1 = KEY-ROOTS(t_1)
  k_2 = KEY-ROOTS(t_2)
  $td = [t_1$.SIZE(), t_2.SIZE()] ▷ Initialise tree distance table.
  for ($k_l \in k_1$) **do**
    for ($k_r \in k_2$) **do**
      TREE-DIST(k_l, k_r, γ)
  return $td[t_1$.SIZE() − 1, t_2.SIZE() − 1]

Algorithm 2.21 TREE-DIST Algorithm

Input: t_1 Input subtree.
t_2 Output subtree.
γ Cost function.
Output: None.

```
algorithm TREE-DIST(t_1, t_2, γ)
    fd = [t_1.SIZE() + 1, t_2.SIZE() + 1]                ▷ Local forest distance table.
    fd[0][0] = 0
    for (i ∈ {1 ... t_1.SIZE()}) do
        fd[i][0] = fd[i − 1][0] + γ(t_1[i] →)
    for (j ∈ {1 ... t_2.SIZE()}) do
        fd[0][j − 1] = fd[0][j − 1] + γ(→ t_2[j])
    for (i ∈ {1 ... t_1.SIZE()}) do
        f_1 = CREATE-FOREST(t_1, i)          ▷ Create forest from post-order index i
        for (j ∈ {1 ... t_2.SIZE()}) do        onward.
            f_2 = CREATE-FOREST(t_2, j)
            if (f_1.IS-TREE() ∧ f_2.IS-TREE()) then
                fd[i][j] = min(
                    fd[i − 1][j] + γ(f_1[i] →),
                    fd[i][j − 1] + γ(→ f_2[j]),
                    fd[i − 1][j − 1] + γ(f_1[i] → f_2[j]))
                td[f_1.GET-TREE-INDEX()][f_2.GET-TREE-INDEX()] = fd[i][j]
            else
                (fs_1, ft_1) = SPLIT-FOREST(f_1)
                (fs_2, ft_2) = SPLIT-FOREST(f_2)
                fd[i][j] = min(
                    fd[i − 1][j] + γ(f_1[i] →),
                    fd[i][j − 1] + γ(→ f_2[j]),
                    fd[fs_1.GET-FOREST-INDEX()][fs_2.GET-FOREST-INDEX()]+
                      td[ft_1.GET-TREE-INDEX()][ft_2.GET-TREE-INDEX()])
```

The MTED algorithm works by initially identifying each key-root in both the input and output trees. A node n in a tree is a key-root if its leftmost descendant (a leaf's leftmost descendant is itself) is unique among all leftmost descendants found from the set of trees that are the ancestors of n. For the trees in Figure 2.14, the key-roots for tree et_1 are at post-order indices $3, 5$ and 6 and et_2's are at post-order indices $2, 5$ and 6.

We then find each pair of subtrees rooted at each key-root k_1 and k_2, where $k_1 \in \text{KEY-ROOTS}(t_1)$ and $k_2 \in \text{KEY-ROOTS}(t_2)$. For each pair k_1 and k_2, we let the size of $k_1 = m$ and the size of $k_2 = n$. We then label k_1 and k_2 with an index according to a post-order traversal. These are distinct from the post-order indexes of the original trees. Pairs of forests are then extracted of size $i \in \{0 \ldots n\}$ and $j \in \{0 \ldots m\}$, such that the pair of forests of size i and j contains the nodes labelled $(\{1 \ldots i\}, \{1 \ldots j\})$ according to each subtree's post-order traversal. An i or j of 0 represents the empty forest, $\emptyset$. For each of these forests, the MTED is then computed between them.

While this method may appear computationally expensive, in practice, computing the MTED for each pair of forests can be performed in constant time if the solutions to the required subproblems have already been calculated. As an example, we will show the calculations required to compute the MTED between the key-roots of et_1 and et_2 in Figure 2.14. For each pair of subtrees located at the key-roots, a DP table is shown in Table 2.4. Each cell (i, j) in each table refers to the minimum edit distance between the forests as described in the previous paragraph. As an example, in Table 2.4c, at cell $(6, 2)$, the MTED is 4. This means that the forest described by the nodes $\{1 \ldots 2\}$ in k_1 (the tree rooted at post-order index 2 in et_1) has a minimum edit distance to the forest described by nodes $\{1 \ldots 6\}$ in k_2 (the tree rooted at post-order index 6 in et_2) of 4.

By visualising the problem in this manner, we can quickly fill in each table. The cell at $(\emptyset, \emptyset)$ is always 0 (according to Equation (2.7)). The cells at $(\emptyset, j)$ use the value at $(\emptyset, j-1)$ and add the cost of deleting the node j using the γ cost function. Similarly, the cells at $(i, \emptyset)$ use the value at $(i-1, \emptyset)$ and add the cost of inserting the node i using the γ cost function. These correspond to the Equations (2.8) and (2.9).

All other cells are calculated according to Equations (2.10) to (2.13). For a cell at (i, j), we take the minimum of three values; first, the cell at $(i-1, j)$ added to the cost of inserting node i. Second, the cell at $(i, j-1)$ added to the cost of deleting

Table 2.4: MTED subproblems calculated when computing the tree edit distance between the trees et_1 and et_2 shown in Figure 2.14. Each table represents a subproblem computing the MTED between a key-root k_1 in et_1 and a key-root k_2 in et_2. Each cell at (i, j) in each table shows the MTED between the forests containing nodes $\{1 \dots i\}$ and $\{1 \dots j\}$ as found through a post-order traversal of the key-roots k_1 and k_2 respectively. We use $\emptyset$ to refer to the empty forest. Certain cells are highlighted, which show when that value has been used to update the tree distance table in Table 2.5.

(a) MTED table between subtrees at index 2 in et_1 and 3 in et_2.

	∅	1
∅	0	1
1	1	0
2	2	1

(b) MTED table between subtrees at index 2 in et_1 and 5 in et_2.

	∅	1
∅	0	1
1	1	1
2	2	2

(c) MTED table between subtrees at index 2 in et_1 and 6 in et_2.

	∅	1	2	3	4	5	6
∅	0	1	2	3	4	5	6
1	1	1	1	2	3	4	5
2	2	2	2	2	2	3	4

(d) MTED table between subtrees at index 5 in et_1 and 3 in et_2.

	∅	1
∅	0	1
1	1	1

(e) MTED table between subtrees at index 5 in et_1 and 5 in et_2.

	∅	1
∅	0	1
1	1	0

(f) MTED table between subtrees at index 5 in et_1 and 6 in et_2.

	∅	1	2	3	4	5	6
∅	0	1	2	3	4	5	6
1	1	1	2	3	4	4	5

(g) MTED table between subtrees at index 6 in et_1 and 3 in et_2.

	∅	1
∅	0	1
1	1	1
2	2	1
3	3	2
4	4	3
5	5	4
6	6	5

(h) MTED table between subtrees at index 6 in et_1 and 5 in et_2.

	∅	1
∅	0	1
1	1	1
2	2	2
3	3	3
4	4	4
5	5	4
6	6	5

(i) MTED table between subtrees at index 6 in et_1 and 6 in et_2.

	∅	1	2	3	4	5	6
∅	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	0	1	2	3	4
3	3	2	1	2	3	4	5
4	4	3	2	1	2	3	4
5	5	4	3	2	3	2	3
6	6	5	4	3	3	3	2

Table 2.5: Tree distance table used when calculating the MTED between the trees et_1 and et_2 shown in Figure 2.14. Each cell (i, j) shows the MTED between the subtrees rooted at post-order indices i and j in et_1 and et_2 respectively. When computing the distance between the key-roots, as shown in Table 2.4, when it is known that the two forests being compared to each other are also trees, we can update a value in this table. The colour of each cell shows where that cell's value came from.

	1	2	3	4	5	6
1	0	1	2	3	1	5
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4	3	3	1	2	4	4
5	1	1	3	4	0	5
6	5	5	3	3	5	2

node j, and one additional value. If the two forests being compared are also trees, then this additional value is computed as the value at $(i - 1, j - 1)$ plus the cost of relabelling node i with the label in node j. If they are not both trees, then we break both forests into a pair (fs, ft), where fs represents the subforest and ft the singleton tree. We compute the edit distance between fs_i and fs_j, and ft_i and ft_j, and add the costs of these two subproblems. If we proceed by filling in this table row by row, we can guarantee that the MTED of fs_i to fs_j will have already been computed - though deducing which cell contains this value may not be obvious. For the cost of the singleton trees, an additional DP table contains these values, which we call the tree distance table. For the trees in Figure 2.14, this is shown in Table 2.5.

However, as of yet we have not discussed how to fill in the tree distance table. This table is read differently to the other tables. Specifically, the cell (i, j) refers to the MTED between the subtree rooted at post-order index i in et_1 and the subtree rooted at post-order index j in et_2.

We do not need to perform any additional computation to fill in these values, as they are already computed when we are finding the minimum edit distance for the other tables. If we hit the case in Equation (2.12) where two trees are being compared, we know that a corresponding cell in the tree distance table can be filled in. In the DP tables in Table 2.4, there are several coloured cells that correspond to identically coloured cells in Table 2.5, which show that a value has been copied to

the tree distance table.

We can guarantee that the required values in the tree distance table will already have been computed when they are required by the key-root computations, as shown in Table 2.4, by ensuring that the MTED between key-roots are computed in a specific order. If we iterate over each key-root in t_1 ordered by their post-order index, and all the key-roots in k_2, then for any pair of key-roots at post-order indices (i, j), any required subtree computations will have already been computed. This is because, by using a post-order traversal, we will be working from a bottom up approach, and any tree distance calculations required will be rooted at post-order index (i', j') such that $i' < i$ and $j' < j$.

Finally, to conclude the algorithm, the value of the MTED between the input trees t_1 and t_2 can be found in the bottom rightmost cell in the tree distance table. In Algorithms 2.20 and 2.21 we have shown the pseudocode for these processes, which allow the MTED to be computed for two trees. It is this algorithm that we use, when required, in this thesis.

2.6.3 Summary & Discussion

The MTED problem is a broad subject, and in truth this section only serves as an introduction to the topic at large. We point the reader to several resources which provide more detailed information. Bille [19] gives a survey of the MTED problem and provides an overview of several variants of the problem. Paaßen [139] presents an in-depth overview of the problem, showing various algorithms to calculate it, and explains in detail how they work. Finally, Zhang and Shasha [181] present the original description of the algorithm contained in this section.

2.7 Conclusions

The goal of this chapter was to provide the context for the work contained in this thesis. We have discussed literature relating to SAT, LS-SAT heuristics, the automated creation of heuristics, program synthesis and the MTED problem.

Research in automated heuristic creation has predominantly focused on using population-based approaches of program synthesis, such as GP, to achieve its goal. However, this is not the only form of program synthesis that has been developed in wider computer science. It can be reasoned other techniques could have a role to play

in heuristic creation.

The domain that we create heuristics for in this thesis, LS-SAT, is a domain that previous researchers have attempted to automatically create heuristics for. This previous work gives us some context regarding how to represent a heuristic in this domain, how to evaluate the created heuristics, and previous examples of heuristics that have been automatically created. In turn, this previous work forms the cornerstone of our research; how can these automated heuristic creation strategies be improved, what alternate techniques can be employed, and is it possible to create more effective heuristics.

LS-SAT heuristic development has progressed since this previous work was undertaken. There have been many advances in this area, which provides us with the opportunity to consider heuristic components that previous authors were unaware of, and to consider whether they are applicable for automated heuristic creation. It also raises the question of how do we represent some of these constructs in a general language for describing heuristics, as well as how to develop a system that can have these different techniques cooperating together.

In Section 2.6 we reviewed the literature concerning the MTED problem. Though its relevance may not be clear in the context of the rest of the work in this chapter, this section is vitally important for understanding the research in Chapters 5 to 7. In the next chapter, we provide details showing how we represent and evaluate heuristics in this thesis, which provides the foundation for the work undertaken in all subsequent chapters.

Chapter 3

Heuristic Representation & Evaluation

3.1 Introduction

In this chapter we describe the way in which we represent and evaluate LS-SAT heuristics in this thesis. The core aims of this chapter are to give a formal description of our representation of heuristics, and to show how a heuristic is evaluated through a fitness function and a testing set. In later chapters we use program synthesis techniques on the described representation to automatically create heuristics, which are evaluated using the given fitness function.

In Section 3.2 we introduce the reader to the heuristic representation used in this thesis, as well as providing descriptions of how each heuristic component is evaluated. This section references the literature in Sections 2.2, 2.3 and 2.5.2, and assumes a familiarity with LS-SAT heuristic design and simple type systems. Section 3.3 provides the reader with an account of how the heuristics are used as part of an overarching local search algorithm to solve a SAT problem instance. This section also shows how the local search algorithm is designed, and the methods used to ensure that the heuristics are evaluated in an efficient manner. In Section 3.4 we detail how we evaluate the performance of the heuristics. This is done through a fitness function and a testing set. We also evaluate the performance of several hand-crafted heuristics. In later chapters, these results allow us to compare the performance of automatically created heuristics to these hand-crafted ones. Finally in Section 3.5 we present the discussions and conclusions from the research described in this chapter.

3.2 Heuristic Representation

In this section we detail the representation we use to formulate LS-SAT heuristics in this thesis. Like the heuristic representations described in Section 2.4.1, our representation consists of two components; a DSL - conceptually a set of terms - and a data structure used to compose elements of the DSL. We use a tree data structure to construct heuristics from the DSL. Each node in the tree must be labelled with a term from the DSL, and can have an unbounded number of children.

In Section 2.5 when discussing program synthesis techniques, we referred to the created candidate solutions as program trees. The structures that are created from the representation described in this section are both program trees and LS-SAT heuristics; in the context of program synthesis, they are arbitrary program trees. In the context of the target domain, they are LS-SAT heuristics. While the two terms can be used interchangeably, we make an attempt to use the correct one when appropriate.

The design of our heuristic representation takes inspiration from previous work in the design of systems to automate the creation of LS-SAT heuristics [60, 62, 61, 9]. We use a type system to prohibit the combination of certain terms in the DSL. This allows us to ensure that language terms which conceptually mean very different things are not combined in ways that we do not intend. The type system we use is identical to that laid out in Section 2.5.2. A requirement of this type system is that each term in the language must have an associated type signature.

Some work within program synthesis research describes the associated language as a CFG, which is used to prohibit the combination of certain terms in the same way a type system does. The DSL that we present in this section could be formulated as a CFG, and the same set of program trees would be able to be expressed. However, we choose to use a type system for several reasons. Firstly, we have found it to be easily extendable, as it allows us to add new terms and types to the language quickly, as well as perform experiments using subsets of the DSL without having to construct a new CFG. Secondly, it allows the work we undertake to remain consistent with the literature on program synthesis in Section 2.5. Finally, though the type system we use is simple, it could be easily extended with more complicated mechanics such as polymorphic typing. Achieving the same effect with a CFG would be more difficult. In previous research [30], the DSL we used was presented as a CFG. To be clear to the reader, there is no difference between the set of expressible program trees that

can be created from the CFG presented in that work, and the equivalent set of terms in the DSL presented in this section.

The DSL is presented in two tables. In Table 3.1 we show the functions in the DSL with their type signature and an explanation of how they are evaluated. In Table 3.2 we show the terminals in the language. From these two tables, the set of principle types can be constructed, which is shown in Figure 3.1. The reader should note that throughout this thesis different subsets of the DSL are used as the focus of particular experiments. We note to the reader at the time what terms are under consideration in that experiment.

We do not include an explanation of the terms in Table 3.2, and the meaning of some may not be immediately obvious. In Section 3.2.1 we describe explicitly what is meant by each of those terms. In Section 3.2.2 we discuss the inspiration behind the terms in the language, and provide some examples of previously known, hand-crafted heuristics written using the DSL.

Table 3.1: The set of functions in the DSL used in this thesis to create LS-SAT heuristics.

`PickRandomVar`(vs)	`VarSet` $\rightarrow$ `Var`
Given the non-empty set of variables vs, this function returns a randomly chosen variable from vs.	
`GetBestVar`(vs, g)	`VarSet` $\rightarrow$ `GainType` $\rightarrow$ `Var`
Given the non-empty set of variables vs and the variable metric g, this function returns the variable v_1. To compute v_1, we do the following. Compute $g(v)$ for each $v \in vs$, and let v_1 = the variable with the best value in vs according to the ordering imposed by g. If there are multiple variables with the same g value as v_1, pick from these randomly.	
`GetBestVarSnd`(vs, g)	`VarSet` $\rightarrow$ `GainType` $\rightarrow$ `Var`
Given the non-empty set of variables vs and the variable metric g, this function returns the variable v_2. To compute v_2, we do the following. Compute $g(v)$ for each $v \in vs$, and let v_2 = the variable with the second best value in vs according to the ordering imposed by g. If there are multiple variables with the same g value as v_2, pick from these randomly.	

Continued on next page

Table 3.1: The set of functions in the DSL used in this thesis to create LS-SAT heuristics. (Continued)

`GetBestVarAge`(vs, g)	`VarSet` → `GainType` → `Var`
Given the non-empty set of variables vs and the variable metric g, this function returns the variable v_1. To compute v_1, we do the following. Compute $g(v)$ for each $v \in vs$, and let $v_1 =$ the variable with the best value in vs according to the ordering imposed by g. If there are multiple variables with the same g value as v_1, pick the variable with the maximum AGE, breaking ties randomly.	
`GetBestVar2`(vs, g_1, g_2)	`VarSet` → `GainType` → `GainType` → `Var`
Given the non-empty set of variables vs and the variable metrics g_1 and g_2, this function returns the variable v_1. To compute v_1, we do the following. Compute $g_1(v)$ and $g_2(v)$ for each $v \in vs$, and let $v_1 =$ the variable with the best value according to the ordering imposed first by g_1, then g_2 to resolve tie-breaks. If there are multiple variables with the same g values as v_1, pick from these randomly.	
`PickOldest`(vs)	`VarSet` → `Var`
Given the non-empty set of variables vs, this function returns the variable v_1. To compute v_1, we do the following. Find the AGE value for each $v \in vs$, and let $v_1 =$ the variable with the maximum AGE. If there are multiple variables with the same AGE value as v_1, pick from these randomly.	
`WeightedVarPick`(vs, gen, ls)	`VarSet` → `VarProb` → `List VarProb` → `Var`
Given the non-empty set of variables vs, the `VarProb` element gen which represents a function that creates numeric values from a variable, and the potentially empty list of additional `VarProb` elements ls of size n, this function returns the variable v_w. To compute v_w, we do the following. For each of the variables $v \in vs$, let $d_{v,0} = gen(v)$. For each additional $l_i \in ls$, let $d_{v,i} = d_{v,i-1} \times l_i(v)$. Then, using each $d_{v,n}$ value as the weight of v, let v_w be the result of performing a weighted pick on the variables in vs. If all weights are 0, pick from vs randomly.	
`PickRandomM`(vs)	`Maybe VarSet` → `Maybe Var`
Given the potentially empty set of variables vs, pick a variable from vs randomly. If vs is empty, return $nullopt$.	

Continued on next page

Table 3.1: The set of functions in the DSL used in this thesis to create LS-SAT heuristics. (Continued)

<table>
<tr><td><code>GetBestVarM</code>(vs, g)</td><td><code>Maybe VarSet</code> → <code>GainType</code> → <code>Maybe Var</code></td></tr>
<tr><td colspan="2">Given the potentially empty set of variables vs and the variable metric g, this function returns the variable v_1. To compute v_1, we do the following. Compute $g(v)$ for each $v \in vs$, and let $v_1 =$ the variable with the best value according to the ordering imposed by g. If there are multiple variables with the same g value as v_1, pick from these randomly. If vs is empty, return nullopt.</td></tr>
<tr><td><code>GetBestVarAgeM</code>(vs, g)</td><td><code>Maybe VarSet</code> → <code>GainType</code> → <code>Maybe Var</code></td></tr>
<tr><td colspan="2">Given the potentially empty set of variables vs and the variable metric g, this function returns the variable v_1. To compute v_1, we do the following. Compute $g(v)$ for each $v \in vs$, and let $v_1 =$ the variable with the best value according to the ordering imposed by g. If there are multiple variables with the same g value as v_1, pick the variable with the maximum Age, breaking ties randomly. If vs is empty, return nullopt.</td></tr>
<tr><td><code>Filter</code>(cmp, g, i, vs)</td><td><code>Comparator</code> → <code>GainType</code> → <code>Integer</code> →
<code>VarSet</code> → <code>Maybe VarSet</code></td></tr>
<tr><td colspan="2">Given the comparison operator cmp, the variable metric g, the integer i and the non-empty set of variables vs, this function filters the variables in vs, creating the return value ls. To compute ls, we do the following. Compute $g(v)$ for each $v \in vs$. Then compare $g_{(}v)$ to i using the comparison operator cmp. If the result of this expression is $True$, add v to ls. If, after evaluating all vs, ls is empty, return nullopt, else return ls.</td></tr>
<tr><td><code>GetOldestVar</code>(v_1, v_2)</td><td><code>Var</code> → <code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the variables v_1 and v_2, pick the variable with the maximum Age. If both variables have the same Age value, pick one randomly.</td></tr>
<tr><td><code>IfIsNull</code>(v_1, v_2)</td><td><code>Maybe Var</code> → <code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the potentially null variable v_1 and the variable v_2, if v_1 is nullopt, return v_2, else return v_1.</td></tr>
</table>

Continued on next page

Table 3.1: The set of functions in the DSL used in this thesis to create LS-SAT heuristics. (Continued)

<table>
<tr><td><code>IfNotMinAge</code>(vs, v_1, v_2)</td><td><code>VarSet</code> → <code>Var</code> → <code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the non-empty set of variables vs, and the variables v_1 and v_2, if v_1's AGE is not equal to the minimum AGE among the variables in vs, return v_2, else return v_1.</td></tr>
<tr><td><code>IfRandLt</code>(p, v_1, v_2)</td><td><code>Probability</code> → <code>Var</code> → <code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the probability p and the variables v_1 and v_2, with probability p pick v_1, and with probability $1 - p$ pick v_2.</td></tr>
<tr><td><code>IfTabu</code>(a, v_1, v_2)</td><td><code>Age</code> → <code>Var</code> → <code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the AGE value a and the variables v_1 and v_2, if the AGE of variable v_1 is less than a, return v_2 else return v_1.</td></tr>
<tr><td><code>IfVarCompare</code>(cmp, g, v_1, v_2)</td><td><code>Comparator</code> → <code>GainType</code> → <code>Var</code> → <code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the comparison operator cmp, the variable metric g and the variables v_1 and v_2, let $g_{v,1} = g(v_1)$ and $g_{v,2} = g(v_2)$. Compare $g_{v,1}$ to $g_{v,2}$ using the comparison operator cmp. If the resulting expression is $True$, then return v_1, else return v_2.</td></tr>
<tr><td><code>IfVarCond</code>(cmp, g, i, v_1, v_2)</td><td><code>Comparator</code> → <code>GainType</code> → <code>Integer</code> → <code>Var</code> → <code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the comparison operator cmp, the variable metric g, the integer i and the variables v_1 and v_2, let $g_{v,1} = g(v_1)$. Compare i to $g_{v,1}$ using the comparison operator cmp. If the resulting expression is $True$ return v_1 else return v_2.</td></tr>
<tr><td><code>ExponentFunction</code>(fp, g)</td><td><code>FloatingPoint</code> → <code>GainType</code> → <code>VarProb</code></td></tr>
<tr><td colspan="2">Given the number fp, and the variable metric g, a <code>VarProb</code> element is created that represents the function f which takes a variable v as its input.

$$f(v) = \begin{cases} fp^{-g(v)} & \text{if } \textsc{Base-Gain-Type}(g) \wedge \\ & \quad g.b = True \text{ (see Definition 21)} \\ fp^{g(v)} & \text{otherwise} \end{cases} \tag{3.1}$$</td></tr>
</table>

Continued on next page

{ `VarSet`, `Var`, `GainType`, `Comparator`,
`Integer`, `Probability`, `Age`, `FloatingPoint`,
`List VarProb`, `VarProb`, `Maybe Var`, `Maybe VarSet` }

Figure 3.1: The set of principle types in the DSL used in this thesis to create LS-SAT heuristics.

Table 3.1: The set of functions in the DSL used in this thesis to create LS-SAT heuristics. (Continued)

<table>
<tr><td><code>Polynomial</code>(fp, g)</td><td><code>FloatingPoint</code> → <code>GainType</code> → <code>VarProb</code></td></tr>
<tr><td colspan="2">Given the number fp, and the variable metric g, a <code>VarProb</code> element is created that represents the function f which takes a variable v as its input.
$$f(v) = \begin{cases} (\frac{1}{|g(v)|+1})^{fp} & \text{if } g(v) < 0 \\ g(v)^{fp} & \text{otherwise} \end{cases} \tag{3.2}$$</td></tr>
<tr><td><code>PolynomialNegative</code>(fp, g)</td><td><code>FloatingPoint</code> → <code>GainType</code> → <code>VarProb</code></td></tr>
<tr><td colspan="2">Given the number fp, and the variable metric g, a <code>VarProb</code> element is created that represents the function f which takes a variable v as its input.
$$f(v) = \begin{cases} (\frac{1}{g(v)+1})^{-fp} & \text{if } g(v) > 0 \\ (1 + |g(v)|)^{-fp} & \text{otherwise} \end{cases} \tag{3.3}$$</td></tr>
<tr><td><code>NextElement</code>(x, xs)</td><td><code>VarProb</code> → <code>List VarProb</code> → <code>List VarProb</code></td></tr>
<tr><td colspan="2">Given the <code>VarProb</code> element x and the list of <code>VarProb</code> elements xs, this function creates a single <code>VarProb</code> list containing x followed by xs.</td></tr>
<tr><td><code>UpdatePAWS</code>(v)</td><td><code>Var</code> → <code>Var</code></td></tr>
<tr><td colspan="2">Given the variable v, returns v. However, the use of this function anywhere in a heuristic has an effect on the overall heuristic. If this function is used, then the dynamic clause weights are only updated at the end of each local search loop if this node is evaluated. If it is not used in a heuristic, then the dynamic clause weights are updated at the end of every loop of the overarching local search.</td></tr>
</table>

Table 3.2: The set of terminals in the DSL used in this thesis. The meaning of these terminals can be found in Section 3.2.1.

Type	Possible Values
Integer	$\mathbb{Z}$
Probability	$\{0.0 \ldots 1.0\} + \{$Adapt$\}$
VarSet	{RBC-N, RBC_WA-N, CONF, WFF}
Age	$\mathbb{N}$
Comparator	$\{<, \leq, =, \geq, >\}$
FloatingPoint	$\mathbb{R} - \{0.0\}$
GainType	{PosGain, NegGain, NetGain, PosGain_WA, NegGain_WA, NetGain_WA, SubPosGain, SubNegGain, SubNetGain, SubPosGain_WA, SubNegGain_WA, SubNetGain_WA }
List VarProb	{EndList}
Maybe VarSet	{DecrVars, DecrVars_WA, SubDecrVars, SubDecrVars_WA}

3.2.1 Language Details

In Table 3.2 we presented the set of terminals in the DSL used in this thesis. The meaning of some of those terminals may be obvious to the reader, as they directly reference concepts introduced in Section 2.3, while others may not. In this subsection we describe explicitly what is meant by each of the terminals in Table 3.2.

For brevity, some of the terminals are simple to understand, and their meaning can be given in a single sentence. These are as follows:

- Terminals with an `Integer` type are whole numbers that can either be positive or negative.
- Excluding `Adapt`, terminals with a `Probability` type are real numbers between 0.0 and 1.0.
- Terminals with an `Age` type are whole numbers that can only be positive.
- Terminals with a `Comparator` type represent functions that compare two numbers and return a boolean result. They are used as arguments to the functions `Filter`, `IfVarCompare` and `IfVarCond`.
- Terminals with a `FloatingPoint` type are real numbers and can be positive or negative. They are used exclusively as arguments to the functions `ExponentFunction`, `Polynomial` and `PolynomialNegative`. We explicitly prohibit the number 0 being used, as the function `ExponentFunction` is defined in such a way that using 0 with it could create weights that were set at ∞.
- The terminal `EndList` has a `List VarProb` type and represents the empty list. It is used in conjunction with `NextElement` and `WeightedVarPick` to end a list of `VarProb` functions.

In the remaining parts of this subsection, we detail the meaning behind the rest of the terminals in Table 3.2.

Clause Weighting

There are several pairs of terminals in the DSL that have nearly identical names. For example `PosGain` and `PosGain_WA`. In each pair, one has a `_WA` suffix and one does not. Each terminal in each pair represents some concept or idea that uses a clause's

weight in its description. Those terminals without the `_WA` suffix use static clause weighting (referred to as BaseWeight) and those with the `_WA` suffix use dynamic clause weighting (referred to as PAWSWeight). In Definition 12 we referred to a static weighting scheme as 1. This shorthand is interchangeable with BaseWeight. To be clear to the reader, when we refer to static clause weighting, we refer to clause weights that do not change. On initialisation, each clause's weight is set at 1 and remains constant as the local search algorithm runs.

Dynamic clause weighting refers to clause weights that change as the overarching local search algorithm progresses. As discussed in Section 2.3.4, there are many different dynamic clause weighting schemes that have been proposed. We use a single dynamic clause weighting scheme in our DSL, which is described as follows; on initialisation, all weights are set at 1. The weight update function used is that shown in Algorithm 2.13. We use specific rules regarding when the weight update function is invoked, given as follows:

- If the terminal `UpdatePAWS` is contained within the heuristic, then the weights are only updated at the end of each iteration of local search *if* the `UpdatePAWS` terminal has been evaluated. Otherwise, they are not updated.
- If the heuristic does not contain the `UpdatePAWS` terminal, then the clause weights are updated at the end of each iteration of the local search loop.

These rules allow us to design heuristics whose weights update either on every epoch of the local search, or only when certain criteria has been met.

Throughout the remainder of this subsection, we explain what is meant by each of the terminals that use static or dynamically weighted clauses in their formulation.

Gain Type

"Gain type", a term first used by Fukunaga [60], is the name we attribute to the type signature of terminals that represent some metric associated with a variable. For a variable v and gain type g, we write $g(v)$ to obtain v's g value. Each gain type value $g(v)$ represents some change that flipping v will have on the overall solution. For example, the gain type NetGain computes the overall change in the number of satisfied clauses should the variable v be flipped. All of the gain types in Table 3.2 are either directly represented by, or directly inspired by, previously described variable

metrics in Section 2.3. We refer the reader to Definitions 13 to 15 and 18 to 20 for details. In our DSL we have a static and dynamic weighted variant of each gain type.

We use two generic definitions which, when instantiated with the correct arguments, can be used to formally define what is meant by each of the twelve gain types in the DSL. These definitions are also used in Section 3.3.3, when showing how a generic gain type is updated efficiently. The definitions are as follows:

Definition 21 (Base Gain Type Metric)

A base gain type g is a triple of (W, b, i), where W is a clause weighting scheme, b is a boolean value, and i is an integer.

For a SAT problem F, variable $v \in \textsc{Vars}(F)$ and complete assignment A, this is a metric associated with a variable that represents the number of clauses that would transition between specific states if v is flipped. If b is $True$, it represents the number of clauses whose number of $True$ variables will transition from i to some other number if v is flipped. If b is $False$, it represents the number of clauses whose number of $True$ variables will transition from some other number to i if v is flipped. It can be computed as:

$$\sum_{c \,\in\, (b \;?\; \textsc{TrueLitSet}(F,\; A,\; v) \;:\; \textsc{FalseLitSet}(F,\; A,\; v))} \begin{cases} W_c & \textsc{TrueLits}(A,\; c) = i \\ 0 & \textit{otherwise} \end{cases} \tag{3.4}$$

To refer to this value, we write $g_W(A,\, F,\, v)$. If the assignment, SAT formula and weighting scheme are obvious from the context, we write $g(v)$. If a set of variables are ordered according to g, they are ordered from smallest to largest if b is $False$ and largest to smallest if b is $True$. All base gain type values are positive integers.

Definition 22 (Compound Gain Type Metric)

A compound gain type g consists of a pair of base gain types g_1 and g_2.

For a SAT problem F, variable $v \in \textsc{Vars}(F)$ and complete assignment A, this is a metric associated with a variable that represents the difference between the number of clauses that would transition between the states described by g_1 and g_2 if v is flipped. It can be computed as:

$$g(v) = g_1(v) - g_2(v) \tag{3.5}$$

To refer to this value, we write $g_W(A,\, F,\, v)$. If the assignment, SAT formula and weighting scheme are obvious from the context, we write $g(v)$. If a set of variables are ordered according to g, they are ordered from smallest to largest. Compound gain type values can be positive or negative integers.

Table 3.3: All of the gain type metrics defined in the DSL, described in terms of Definitions 21 and 22. Each compound gain type has two entries. The upper entry is g_1 and the lower is g_2. For each we show its W, b and i values.

Name	Type	W	b	i
`PosGain`	Base	BaseWeight	*False*	0
`NegGain`	Base	BaseWeight	*True*	1
`NetGain`	Compound	BaseWeight BaseWeight	*False* *True*	0 1
`PosGain_WA`	Base	PAWSWeight	*False*	0
`NegGain_WA`	Base	PAWSWeight	*True*	1
`NetGain_WA`	Compound	PAWSWeight PAWSWeight	*False* *True*	0 1
`SubPosGain`	Base	BaseWeight	*False*	1
`SubNegGain`	Base	BaseWeight	*True*	2
`SubNetGain`	Compound	BaseWeight BaseWeight	*True* *False*	2 1
`SubPosGain_WA`	Base	PAWSWeight	*False*	1
`SubNegGain_WA`	Base	PAWSWeight	*True*	2
`SubNetGain_WA`	Compound	PAWSWeight PAWSWeight	*True* *False*	2 1

In Table 3.3 we show the parameters required to instantiate the twelve gain types in the DSL using these definitions.

For the majority of the functions in Table 3.1 that require a gain type terminal, their implementation details are relatively simple to understand, as the way in which they are defined follows from the research presented in Chapter 2. The exceptions are the functions `ExponentFunction`, `Polynomial` and `PolynomialNegative`. These are functions inspired by the *exp*, *exp-break-only*, *poly* and *poly-break-only* functions described by Balint and Schöning [12] (see Section 2.3.5). In that original work, the authors designed the functions to be used exclusively with variable metrics represented as positive integers. The functions were used to assign weights to variables, and therefore the output had to also be a positive number.

In our language we were unable to create exact copies of the functions described

by Balint and Schöning. This is because some of the variable metrics in the DSL return negative numbers. Instead, we designed three functions `ExponentFunction`, `Polynomial` and `PolynomialNegative` that could be used to create functions analogous to the *exp*, *exp-break-only*, *poly* and *poly-break-only* functions. Our functions were extended so that they could take negative integers as input. When given a negative number, a very small positive number is given as output. We believe that this is the most intuitive design choice we could make, and is in keeping with the intentions of Balint and Schöning.

Finally, the reader may question why we do not use a variable metric representing the Age of a variable. This is because we felt that the Age metric would be difficult to resolve with some of the used functions - for example, `IfVarCond` and `IfVarCompare`. Instead, we included specialised functions that can either pick a variable with the maximum Age (`PickOldest`), use the Age to determine tie-breaks (`GetBestVarAge`, `GetBestVarAgeM`), or use the Age to pick from two variables (`IfTabu`, `IfNotMinAge`, `GetOldestVar`).

Broken Clauses

`RBC-N` and `RBC_WA-N` are two terminals that represent a currently unsatisfied clause in the SAT problem. The "`N`" in each is a placeholder, that in a language is replaced with a positive integer. This allows multiple broken clauses to be used in a single heuristic.

`RBC-N` returns a currently unsatisfied clause, chosen from all unsatisfied clauses with equal probability. `RBC_WA-N` also returns a currently unsatisfied clause, but picks the clause in a different way. Each clause's dynamic weight is used as part of a weighted pick to choose the unsatisfied clause. This is intended to have the effect of making highly weighted clauses more likely to be picked. This specific form of clause selection is not one that we have seen described in the literature previously.

To be clear to the reader, if we have the terminal `RBC-0` appearing twice in a heuristic, then every time that the heuristic is invoked, a random broken clause c is chosen to represent `RBC-0`. Every occurrence of `RBC-0` is then substituted for c. In addition to this, if we have two broken clauses (for example `RBC-0` and `RBC-1`), it is possible that on a single invocation of the heuristic, they could both choose the same clause.

Other Variable Sets

`DecrVars`, its `Sub` and dynamically weighted variants are dynamic sets of variables that can potentially be empty. They all have a type signature of `Maybe VarSet`. By "dynamic sets of variables", we mean that the variables in these sets can change as the local search algorithm progresses. Each of the variable sets are described as follows:

- `DecrVars` is defined exactly as is described by the dynamic set DecrVars in Section 2.3.2.
- `DecrVars_WA` is defined exactly as is described by the dynamic set DecrVars in Section 2.3.2, except it uses $\text{NetGain}_{\text{PAWSWeight}}$ instead of $\text{NetGain}_{\text{BaseWeight}}$ to fill its set.
- `SubDecrVars` is defined exactly as is described by the dynamic set DecrVars in Section 2.3.2, except it uses $\text{SubNetGain}_{\text{BaseWeight}}$ instead of $\text{NetGain}_{\text{BaseWeight}}$ to fill its set.
- `SubDecrVars_WA` is defined exactly as is described by the dynamic set DecrVars in Section 2.3.2, except it uses $\text{SubNetGain}_{\text{PAWSWeight}}$ instead of $\text{NetGain}_{\text{BaseWeight}}$ to fill its set.

The two remaining variable sets in the DSL can never be empty, and have a type signature of `VarSet`. They are described as follows:

- `CONF` is the set of variables whose NVCC (see Section 2.3.6) value is set to $True$.
- `WFF` is the complete set of variables in the SAT problem.

Adaptive Probability

In Table 3.2 there is a single terminal listed that has a type of `Probability` called `Adapt`. This terminal corresponds to an adaptive probability variable, such as those in Section 2.3.3. It works as follows; on initialisation, the variable is set at 0.5. The criteria for a change in probability and the update function that are used are those described in Section 2.3.3. We also use the example constants provided in that subsection.

3.2.2 Example Heuristics

The DSL we have described can be viewed as an extension of the DSLs used by Fukunaga [60, 62, 61]. Our language uses several constructs from more modern LS-SAT solvers; for example, the `DecrVars` and `CONF` terminals reference the dynamic set of variables DecrVars and the configuration checking metric NVCC, which we described in Sections 2.3.2 and 2.3.6 respectively. These two constructs are used to create the G^2WSAT and SW_{CC} heuristics. We have also included a method of choosing a variable using a weighted pick function, a mechanism used in the Sparrow and ProbSAT heuristics.

In our DSL, we have chosen to include some novel and (as far as we are aware) previously unused mechanisms, for which we have no prior evidence that they will be effective components in the creation of heuristics. Specifically, these are the `SubDecrVars`, `SubDecrVars_WA` and `RBC_WA-N` terminals. We chose to include them as, due to the generic nature of the software created, we found it simple to add the functionality to evaluate these terms. The inspiration behind their inclusion was from our belief that they could potentially be effective components. The `Sub` variants of `DecrVars` represent sets of variables that have a positive SubNetGain value. Picking from this set could prove useful as, for example, a diversification strategy. The `RBC_WA-N` term represents picking a broken clause according to the dynamic clause weights, which could prove useful in satisfying a clause that has previously spent much of the local search algorithm unsatisfied.

In Figure 3.2 we show examples of some previously described heuristics created using the DSL. In Figure 3.3 we show WalkSAT in its tree-form. We choose to describe the heuristics in this thesis using the form in Figure 3.2, as these are more succinct than showing the complete trees, and we believe easier to understand.

Our DSL can express all of the heuristics described in Fukunaga's work [60, 62, 61]. One of these heuristics, called Depth-2-2, is shown in Figure 3.2h. The DSL we have created does have some limitations, as some previously described LS-SAT heuristics cannot be represented using it. Any of the heuristics described in Section 2.3.6 that utilise the CSCC method of configuration checking cannot be described. The Sparrow heuristic, introduced in Section 2.3.5, cannot be described as its diversification strategy cannot be formulated in our DSL. Many of the heuristics in Section 2.3.4 that use dynamic clause weighting schemes, other than the PAWS-like scheme we have included in our language, cannot be described.

In addition to this, it is possible to write many heuristics in an imperative style using the functions laid out in Table 2.2 that cannot be represented by our DSL. In designing our DSL, we did not aim for it to be a comprehensive format that could describe all possible heuristics, rather the aim was to design a language that could be used to represent many currently existing heuristics in a general enough format that allowed other, potentially more effective, heuristics to be represented.

3.3 Running a Heuristic on a Problem Instance

The heuristic representation described in the previous section is designed for use with program synthesis techniques. The representation is not designed in such a manner as to describe heuristics that can be easily used as part of an overarching LS-SAT solver. When used in an LS-SAT solver, a heuristic's purpose is to direct the internal search, and it may need to be evaluated millions of times on a single problem instance. It is our desire for the heuristics that we automatically create, when used as part of an LS-SAT solver, to run as efficiently as possible.

Heuristics designed by hand are usually written in a low-level programming language such as C, and the overarching LS-SAT solver compiled with a high degree of optimisation. The compiled program that is created will be in low-level machine code. While it would be possible to convert our heuristic representation to a representation that a compiler could comprehend, and then automatically build the heuristics, such a process would be computationally expensive as it can take several seconds to compile one program. Instead, we evaluate the heuristics at a software level.

It would be impractical to provide the same types of optimisations in our software solution that a compiler such as C can. Yet, we do make an attempt to ensure the heuristics run as fast as is reasonably possible. We perform some post-processing on the created heuristics, and utilise them as part of an efficiently designed LS-SAT solver. By doing this, while we may not achieve the same performance a compiled heuristic would provide, we do reduce the overall time it takes to run the created heuristics on problem instances.

In this section we describe the techniques we use to ensure the heuristic is evaluated efficiently. Key to this are three steps:

- Deduce which auxiliary data structures are required to be maintained for a specific heuristic to operate correctly.

```
GetBestVar WFF NetGain
```

(a) GSAT heuristic.

```
WeightedVarPick
 RBC-0
 PolyNomialNegative 2.4 NegGain
  EndList
```

(b) PROBSAT heuristic using the *poly-break-only* function with a constant of 2.4.

```
IfVarCond = NegGain 0
 GetBestVar RBC-0 NegGain
  IfRandLt 0.5
   GetBestVar RBC-0 NegGain
   PickRandomVar RBC-0
```

(c) WALKSAT heuristic with a noise parameter value of 0.5.

```
IfNotMinAge RBC-0
 GetBestVar RBC-0 NetGain
 IfRandLt 0.5
  GetBestVarSnd RBC-0 NetGain
  GetBestVar RBC-0 NetGain
```

(d) NOVELTY heuristic with a noise parameter value of 0.5.

```
IfIsNull
 GetBestVarAgeM DecrVars NetGain
  IfRandLt 0.01
   PickOldest RBC-0
   IfNotMinAge RBC-0
    GetBestVar RBC-0 NetGain
    IfRandLt Adapt
     GetBestVarSnd RBC-0
       NetGain
     GetBestVar RBC-0 NetGain
```

(e) G^2WSAT heuristic, using ADAPTNOVELTY++ as its internal diversification strategy.

```
IfRandLt 0.01
  PickRandomVar RBC-0
  IfIsNull
    GetBestVarAgeM DecrVars_WA
      NetGain_WA
    UpdatePAWS
      IfNotMinAge RBC-0
        GetBestVar RBC-0
          NetGain_WA
        IfRandLt Adapt
          GetBestVarSnd
            RBC-0 NetGain_WA
          GetBestVar RBC-0
            NetGain_WA
```

(f) GNOVELTY+ heuristic.

Figure 3.2: Eight examples of previously described, hand-crafted heuristics formulated using the DSL described in Tables 3.1 and 3.2.

```
IfIsNull
 GetBestVarAgeM
  Filter > NetGain_WA 0
    CONF
  NetGain_WA
 UpdatePAWS
  PickOldest RBC-0
```

(g) $\mathrm{SW_{CC}}$ heuristic.

```
GetOldestVar
 GetOldestVar
  GetBestVar RBC-1 PosGain
   GetBestVar RBC-0 NetGain
   GetOldestVar
    GetBestVar RBC-0 NegGain
    GetBestVar RBC-1 NegGain
```

(h) Depth-2-2 heuristic.

Figure 3.2: Eight examples of previously described, hand-crafted heuristics formulated using the DSL described in Tables 3.1 and 3.2. (Continued)

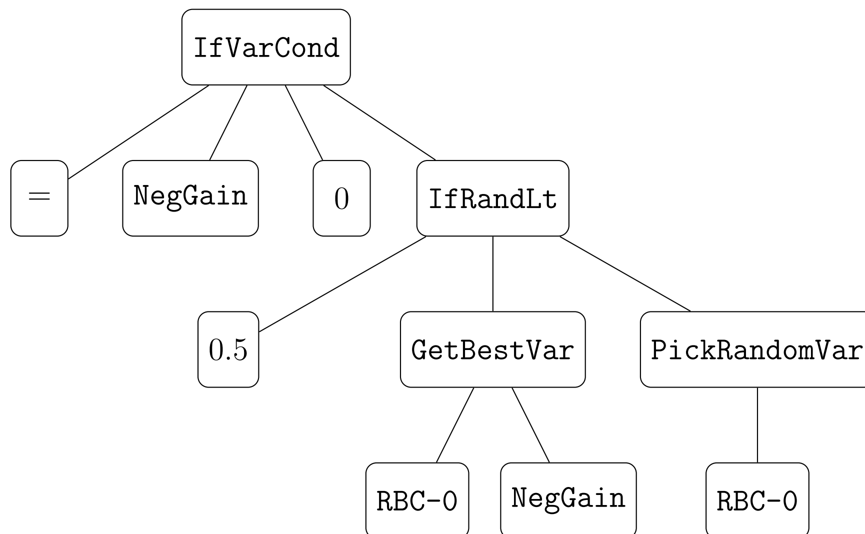


Figure 3.3: The WalkSAT heuristic, shown in Figure 3.2c, visualised as a program tree.

- Convert the heuristic to a format that is suitable for the (potentially) millions of evaluations performed when solving a SAT problem.
- Design the overarching local search algorithm in such a way as to update the required auxiliary data structures correctly.

In the next three subsections, we show how these steps are achieved.

3.3.1 Deduction of a Heuristic's Requirements

The aim of this step is to analyse the heuristic and deduce which auxiliary data structures are required, and therefore need to be maintained as the overarching LS-SAT solver is running. By auxiliary data structure, we specifically mean those mechanisms that underpin the choosing of randomly broken clauses, the maintenance of dynamic sets of variables, the storage of gain type values and dynamic clause weights, and the maintenance of adaptive probability data. We do this by identifying terms in the heuristic which correspond to clear requirements about which specific auxiliary data structures are required.

To record which of these are needed, and the relationship between them, a data structure called DATA-REQUIRED is used. The components of the DATA-REQUIRED structure are described in Table 3.4.

Table 3.4: The set of member variables in the DATA-REQUIRED structure, together with an explanation of their meaning. The information stored concerns which auxiliary data structures are required by a heuristic, as well as identifying the relationships between the components.

clauses	`set<Clause>`
This is the set of unique identifiers that correspond to randomly chosen clauses required by the heuristic. Their inclusion in this set means that the mechanisms to pick and store them must be in place.	

Continued on next page

Table 3.4: The set of member variables in the DATA-REQUIRED structure, together with an explanation of their meaning. The information stored concerns which auxiliary data structures are required by a heuristic, as well as identifying the relationships between the components. (Continued)

variable_sets	`set<VarSet>`
This is the set of variable sets that are required by the heuristic. By "variable set", we refer to any set of variables that are not broken clauses. Most of these are dynamic sets, whose members can change after each iteration of the overarching local search algorithm. Their inclusion in this set means that the mechanisms to store and maintain them must be in place.	
weights	`set<Weight>`
This is the set of clause weights that are required by the heuristic. In our implementation, there are only two types - BASEWEIGHT and PAWSWEIGHT. Their inclusion in this set means that the mechanisms to store and maintain them must be in place. For each weight, we also store at which point in the algorithm they are updated (see Section 3.2.1 for more information regarding this).	
gain_types	`set<GainType>`
This is the set of gain types that are required by the heuristic. Their inclusion in this set means that the mechanisms to store and maintain them must be in place.	
adapt_probability	`Bool`
This is a boolean variable denoting whether an adaptive probability variable is required by the heuristic. If this is set to *True*, it means that the mechanisms to store and maintain an adaptive probability variable must be in place.	

Continued on next page

Table 3.4: The set of member variables in the DATA-REQUIRED structure, together with an explanation of their meaning. The information stored concerns which auxiliary data structures are required by a heuristic, as well as identifying the relationships between the components. (Continued)

gt_vs_requirement	map<(GainType,VarSet),Requirement>
This is a mapping from a pair of gain type and variable set to a requirement. This is for variable sets that are either partially sorted or filtered according to a gain type. By storing this relationship, we can communicate that when a gain type is updated, so are these orderings. Their inclusion in this set means that the mechanisms to store and maintain the variable sets with the given requirements must be in place.	
gt_clause_requirement	map<(GainType,Clause),Requirement>
This is a mapping from a pair of gain type and clause to a requirement. We do not keep information pertaining to any orderings or filtering of clauses as the algorithm runs, as there may be many clauses, and this would be computationally expensive to maintain. Instead, we store the information about how many times a gain type ordering or filtering of a clause may be needed by a heuristic. By doing this, we can memoize this data if it's needed more than once.	
gt_vs_update	map<GainType,vector<VarSet>>
This is a mapping of gain types to vectors of variable sets. Some dynamic variable sets (such as DecrVars) are defined in terms of a gain type, and when this gain type is updated, so is the dynamic set. By storing this relationship, we can inform the algorithm that, when a gain type changes, a dynamic set should also be updated.	
weight_gt_update	map<Weight,vector<GainType>>
This is a mapping of weights to vectors of gain types. Some gain types (such as NegGain_WA) are defined in terms of a clause weighting. By storing the relationship between gain types and weights, we can inform the algorithm that, when a specific clause weight changes, certain gain type values should also be updated.	

We do not show the algorithm to compute the Data-Required data structure. However, we show an example of how it is instantiated in Table 3.5. The instantiation is for the gNovelty+ heuristic, shown in Figure 3.2f. This instantiation follows from the following observations:

- The heuristic requires a single unsatisfied clause to be chosen.
- The heuristic requires the dynamic variable set `DecrVars_WA` be maintained.
- The heuristic requires the dynamic clause weighting scheme PAWSWeight be maintained. This can be seen from the use of the terminals `NetGain_WA` and `DecrVars_WA`.
- The heuristic requires the `NetGain_WA` gain type be maintained, as it is used in both the unsatisfied clause and the `DecrVars_WA` dynamic set.
- An adaptive probability is required for this heuristic.
- The heuristic requires that the `DecrVars_WA` dynamic set be partially ordered, as the best variable according to the `NetGain_WA` is required.
- The two best variables in the chosen broken clause under the ordering imposed by `NetGain_WA` may be required. If this is computed, this data should be memoized as it is used more than once.
- Due to the use of `DecrVars_WA`, when the `NetGain_WA` is updated, this dynamic set may also need to be updated.
- Due to the use of the dynamic clause weighting in `NetGain_WA`, when clause weights are updated, this gain type will also need to be updated.

This part of the heuristic evaluation process identifies the data structures required for a heuristic to operate correctly, as well as the relationships describing how the data is updated. We have designed an LS-SAT solver that can understand this information. It sets up both the correct ordering of the update functions and the memory locations for the required data. In the next subsection, we show how a heuristic is converted to a format that can be used with this LS-SAT solver.

Table 3.5: The instantiation of the DATA-REQUIRED structure for the heuristic GNOVELTY+.

Member Variable	Value
clauses	{`RBC-0`}
variable_sets	{`DecrVars_WA`}
weights	{PAWSWEIGHT($defaultUpdate = False$)}
gain_types	{`NetGain_WA`}
adapt_probability	$True$
gt_vs_requirement	{((`DecrVars_WA`, `NetGain_WA`) → ($sortFirstN = 1$, $calls = 1$))}
gt_clause_requirement	{((`RBC-0`, `NetGain_WA`) → ($sortFirstN = 2$, $calls = 3$))}
gt_vs_update	{(`NetGain_WA` → [`DecrVars_WA`]}
weight_gt_update	{(PAWSWEIGHT → [`NetGain_WA`]}

3.3.2 Evaluating the Heuristic Function

After the heuristic has been analysed, the required data identified, and the relationships regarding when data is to be updated calculated, the heuristic is then post-processed to a machine-code-like format designed to be efficient to run.

We call this format the *internal-form* of the heuristic. Conceptually it is represented as a list of bespoke instructions. Each instruction performs some operation that is part of the overarching variable selection function. Evaluation begins at the start of the list, and proceeds to the end of the list. There are some instructions that change the control flow - by jumping to a later instruction. When all the instructions have been evaluated, the variable to be flipped will have been inserted into a pre-determined place in memory - always `general_data[0]`.

While it would be possible to use the original tree-based variant of the heuristic to compute the variable to flip, by using this format we reduce the overall computational overhead considerably. Traversing the tree-form of the heuristic is itself computationally expensive (in comparison to traversing an array) and, when evaluating a heuristic millions of times, this overhead reduces overall efficiency of the local search algorithm.

While we do not provide details of the internal-form representation, great care

Algorithm 3.1 Detailed LOCAL-SEARCH for SAT

Input:	F	SAT problem instance in CNF.
	$maxFlips$	Maximum number of iterations to run for.
	h	Heuristic to be used.
Output:		Pair of boolean and integer. Boolean is $True$ if solution is found, $False$ otherwise. The integer denotes the number of flips local search undertook.

algorithm LOCAL-SEARCH(F, $maxFlips$, h)
  $assignment$ = INITIALISE(F)
  if (SATISFIED($assignment$, F)) **then** **return** $\{True, 0\}$
  for ($iteration \in \{1 \ldots maxFlips\}$) **do**
    $varToFlip$ = RUN(h)
    UPDATE-DATA($varToFlip$)
    $assignment[varToFlip] = \neg assignment[varToFlip]$
    if (SATISFIED($assignment$, F)) **then return** $\{True, age\}$
    UPDATE-WEIGHTS()
  return $\{False, maxFlips\}$

was taken to ensure that it produces the correct values and is consistent with the descriptions of the DSL given in Section 3.2. An example of the internal-form of a heuristic is shown in Figure 3.4. It shows the internal-form of the gNOVELTY+ heuristic. It has been simplified to make it human-readable.

3.3.3 Overarching Local Search Algorithm

After it has been determined what data is required of a heuristic h, and h converted into a format that can be quickly evaluated, these two parts are brought together to create the overarching local search algorithm utilising h that can be ran on SAT problem instances. In this subsection we explain how the local search algorithm works.

In Algorithm 3.1 we present the pseudocode of the local search algorithm. It can be considered a more detailed version of the pseudocode shown in Algorithm 2.2. It contains five functions, which are described as follows:

Index	Instruction	Memory Locations	Numeric Data
1	PICK_CLAUSE_NO_WEIGHT	{clauses[0]}	–
2	SMALL_SORT_MEMOIZED	{smallSorts[0], gaintype_data[0], clauses[0]}	–
3	IF_RAND_LT	{instr[6]}	0.01
4	PURE_1_RND_RND_CLG	{clauses[0], general_data[0]}	–
5	JUMP	{instr[15]}	–
6	MAYBE_1_GBV_AGE_SET	{large_data[0], general_data[0], gaintype_data[0]}	–
7	IF_IS_NULL	{general_data[0], instr[15]}	–
8	FORCE_WEIGHT_UPDATE	{weights[0]}	–
9	PURE_1_GBV_RND_CLM	{small_sorts[0], general_data[0]}	–
10	IF_NOT_MIN_AGE_CLAUSE	{clauses[0], general_data[0] instr[15]}	–
11	IF_RND_LT_ADAPT	{adapt[0], instr[14]}	–
12	PURE_2_GBV_RND_CLM	{small_sorts[0], general_data[0]}	–
13	JUMP	{instr[15]}	–
14	PURE_1_GBV_RND_CLM	{small_sorts[0], general_data[0]}	–

Figure 3.4: The internal-form of the heuristic gNovelty+, shown in Figure 3.2f. The instructions are processed in the order they are presented. Each instruction has a set of arguments referring to pre-determined locations in memory where results are either stored or are to be stored. Some instructions require additional numerical data, such as IF_RAND_LT.

- INITIALISE: This function initialises each variable in the assignment with a randomly chosen boolean value.
- SATISFIED: This function checks whether the assignment satisfies the SAT problem.
- RUN: This function evaluates the internal-form of the heuristic h. It returns the variable to be flipped.
- UPDATE-DATA: This function updates the internal data that will change when the variable $varToFlip$ is flipped.
- UPDATE-WEIGHTS: This function updates the weights of the clauses if it is required. This can change many other auxiliary data structures in the algorithm such as gain types that rely on weights. The relationships that describe what data needs to be updated are detailed in the DATA-REQUIRED structure, described in Table 3.4.

Below we detail how two components of the function UPDATE-DATA are formulated, and then provide an outline of the UPDATE-DATA function. These two components are the clause data update function and the gain type data update function. The clause data update function keeps track of which clauses are currently satisfied, and the gain type data update function updates arbitrary gain type data when a variable is flipped.

Clause Data Update Function

Conceptually, the goal of an LS-SAT algorithm is to satisfy all clauses - thus proving the formula to be $True$. The naive method to identify when this occurs would be to check all clauses after each iteration of the local search loop, yet this would obviously be computationally expensive to continually perform. Instead, we maintain two data structures that allow us to do this efficiently. The first is called N-TRUE-VARS. This is a matrix containing integers. Each index of the matrix represents the numerical identifier of a clause c, and the contained element shows the total number of literals within c that evaluate to $True$. The second is called N-TRUE-SETS, which is a matrix of sets of clauses. Each index of the matrix corresponds to the total number of $True$ literals that all clauses contained within that element have. By probing the size of the 0th element in N-TRUE-SETS, we can quickly see whether the formula

is satisfied - as if there are no clauses with zero $True$ literals, the assignment must satisfy the formula as all clauses will have at least one satisfied literal. This is how the SATISFIED function checks to see whether the formula is satisfied.

When a variable v is flipped, we update these data structures in the following way. As we know that all the clauses in the TRUELITSET(v) set will have their N-TRUE-VARS value reduced by 1, and those in the FALSELITSET(v) set will have their value increased by 1, we can update N-TRUE-SETS using these sets, and quickly deduce whether an assignment satisfies the formula. An example of this is shown in Figure 3.5.

Gain Type Update Function

For any given heuristic, its DATA-REQUIRED structure describes which gain type values will need to be maintained. By "maintained" we mean that, the correct gain type values for each variable in a given problem may be required by the heuristic. In our implementation the gain type data is kept in a matrix. Whenever a variable is flipped or a clause's dynamic weight changes, this can change the gain type values for some variables in the problem, and therefore they will need to be updated.

Rather than re-computing these values from scratch every time they change, it is more efficient to only compute the changes in the values. When a dynamic weight changes, this is relatively easy to do. For a gain type g represented by the triple (W, b, i), when the weight W_c of a clause c changes by d, then for each $l \in$ LITS(c) the gain type of g(VAR(l)) changes by d.

When a variable is flipped, the changes in gain type value are harder to compute. Other researchers [15, 62] have considered update functions for a single (what we call) gain type metric, such as NEGGAIN or POSGAIN. We use a generalised technique which follows from our generalised definitions of gain types that allow us to maintain and update all the different gain types described in our DSL.

As an example, let us consider SUBNEGGAIN$_1$ (represented by the terminal `SubNegGain`). When a clause c has 2 $True$ literals, all of the $True$ literals "fire" in that clause c. That is to say, the SUBNEGGAIN$_1$ value for the variables in those literals includes the weight of c. For a variable v in the SAT problem, its associated SUBNEGGAIN$_1$ value is the sum of the weight of every clause that has exactly 2 $True$ literals in, which also contains v in a literal that evaluates to $True$. When a clause c's N-TRUE-VARS value changes from 2 (to either 3 or 1), the $True$ literals

Clause Index	1	2	3	4	5	6	7	8	9	10
# True Literals	3	0	1	1	2	1	0	2	1	2

# True Variables	Clauses
0	{2,7}
1	{3,4,6,9}
2	{5,8,10}
3	{1}

(a) An example of the N-True-Vars (upper) and N-True-Sets (lower) data structures at a point in an LS-SAT algorithm. The formula is unsatisfied by the current assignment, as there are currently clauses with zero $True$ literals.

Clause Index	1	2	3	4	5	6	7	8	9	10
# True Literals	2	1	1	2	2	1	0	1	0	3

# True Variables	Clauses
0	{7,9}
1	{2,3,6,8}
2	{1,4,5}
3	{10}

(b) An example of the N-True-Vars (upper) and N-True-Sets (lower) data structures at a point in an LS-SAT algorithm. These examples show the changes made to the two data structures compared to those shown in Figure 3.5a after a variable has been flipped. The variable flipped had a TrueLitSet containing clauses $\{1, 8, 9\}$ and a FalseLitSet containing clauses $\{2, 4, 10\}$. In N-True-Vars green indicates clauses whose number of satisfied literals has increased, while red denotes clauses whose number of satisfied literals has decreased.

Figure 3.5: Examples of two arbitrary N-True-Vars and N-True-Sets data structures at a point in an LS-SAT algorithm. The data structures in Figure 3.5b are obtained from the data structures in Figure 3.5a after a variable has been flipped.

within c have to subtract c's weight from their respective variable's SubNegGain$_1$ value - that is to say, the gain type metric needs to be updated. When a clause c whose N-True-Vars value becomes 2 after a variable is flipped, all the $True$ literal's variables in that clause must now add the weight of c to their SubNegGain$_1$ value.

To generalise this for any gain type, we use that gain type's threshold value - from Definition 21, this is i. When a clause c's number of $True$ literals equals the threshold value, all of the $True$ (or $False$) literal's variables in c must now fire. If the gain type's b value is $True$, all the $True$ literals in c's variables fire. If b is $False$, all the $False$ literals in c's variables fire. This corresponds to increasing the gain type values for the literal's variables according to the weight of c. When c's number of $True$ literals moves away from the threshold value, all the literal's whose variables are firing must now stop - that is, their gain type value must decrease. For compound gain types, there are two threshold values. We use two separate update functions for them, but point them at the same memory location when changing the gain type values.

To update the gain type value of all of the relevant literal's variables in a clause c, we need a structure that can return all of the current variables which are part of a $True$ literal in c, and all of the current variables which are part of a $False$ literal in c. This is done using a data structure called Var-Pos. Var-Pos is a matrix, where each index represents a clause c's numerical identifier. The elements of Var-Pos are conceptually pairs of sets; the first element in the pair is the set of variables which are part of a current $True$ literal in c, and the second element is the set of variables which are part of a current $False$ literal in c. In reality the two sets are represented as a single fixed-sized array. The N-True-Vars[c] value informs us of the boundary between the two sets. Two examples of the Var-Pos data structure are shown in Figure 3.6.

As the local search algorithm progresses, the Var-Pos structure will need to be updated. This occurs when a literal in a clause c changes from being satisfied to unsatisfied (or vice versa). When this happens, c's Var-Pos sets need to be updated. In our LS-SAT solver we perform this update operation in $\Theta(1)$ time. When a variable v is flipped, we change Var-Pos[c]'s flat array by moving v from wherever it was to the new boundary value (denoted by N-True-Vars[c]). However, since we do not know where v was originally in Var-Pos[c], we have to locate its index in Var-Pos[c]. A naive search in a clause of size l would take $\mathcal{O}(l)$ time.

To perform this update operation in $\Theta(1)$ time, we maintain an additional data structure called VAR-POS-POS. Conceptually VAR-POS-POS is a matrix of matrices; each outer index refers to a clause c, and each inner index to a variable v. VAR-POS-POS's $[c][v]$ value informs the algorithm of where v is in VAR-POS$[c]$. When v's position changes in VAR-POS$[c]$, we can update VAR-POS-POS$[c][v]$'s value with its new position.

Usually, when given a SAT problem F, the size of F's clauses are small in comparison to the number of variables in F. Under these circumstances, VAR-POS-POS will be a sparse matrix. We take advantage of this to reduce VAR-POS-POS's memory footprint significantly by storing multiple clause's VAR-POS-POS elements in a single outer array index - as long as a set of clauses do not share a single variable, they can be stored in the same VAR-POS-POS element. Two examples of the VAR-POS-POS data structure are shown in Figure 3.6.

The remaining functions we use to update a SAT problem's data structures are relatively simple to understand. Therefore, we do not include an explanation of their construction. However, an outline of the UPDATE-DATA algorithm is shown in Algorithm 3.2. The structure of UPDATE-DATA is inspired by update algorithms described by Fukunaga [62] and Balint et al. [15].

3.4 Evaluating a Heuristic's Performance

In this section we provide details regarding the fitness function and testing set used throughout this thesis. The format of this section is as follows; in Section 3.4.1 we present the fitness function used to evaluate heuristics. In Section 3.4.2 we present the results from running some well-known, hand-crafted heuristics on the fitness function. Finally in Section 3.4.3 we present the testing set used to evaluate a heuristic's performance on a wider range of problems. We also present the results from running some well-known heuristics on the testing set.

3.4.1 Fitness Function

To compute the fitness of a heuristic h, it is used with the LS-SAT solver shown in Algorithm 3.1 to solve a set of SAT problem instances. The set of problem instances is broken up into subsets based on each problem's number of clauses and variables.

Position Index	1	2	3	4	5
Variable	10	15	20	21	22

Variable	10	...	15	...	20	21	22
Position	1	...	2	...	3	4	5

(a) Example of the VAR-POS (upper) and VAR-POS-POS (lower) matrices for a single clause.

Position Index	1	2	3	4	5
Variable	10	15	22	21	20

Variable	10	...	15	...	20	21	22
Position	1	...	2	...	5	4	3

(b) Example of the VAR-POS (upper) and VAR-POS-POS (lower) matrices for a single clause. These are obtained from the data structures in Figure 3.6a after the variable 22 has been flipped. In this clause, this meant its corresponding literal now evaluates to $True$.

Figure 3.6: Examples of two arbitrary VAR-POS and VAR-POS-POS data structures at a point in an LS-SAT algorithm. The green variables in the VAR-POS data structures signify literals containing that variable which currently evaluate to $True$. The red denote those that currently evaluate to $False$. The data structures in Figure 3.6b are obtained from the data structures in Figure 3.6a after the variable 22 has been flipped.

Algorithm 3.2 LS-SAT UPDATE-DATA Function

Input: v Variable being flipped.
Output: None.

```
algorithm UPDATE-DATA(v)
    for (c ∈ TRUELITSET(v)) do          ▷ Literals that were True become False.
        INTERNAL-UPDATE(v, c, −1)
    for (c ∈ FALSELITSET(v)) do         ▷ Literals that were False become True.
        INTERNAL-UPDATE(v, c, +1)

algorithm INTERNAL-UPDATE(v, c, change)
    n = N-TRUE-VARS[c]
    for (g ∈ GAIN-TYPES) do
        if (g.i = n) then
            UNFIRE-GAIN-TYPE(g, v, c)
    UPDATE-VAR-POS-MATRIX(v, c)
    N-TRUE-VARS[c] = N-TRUE-VARS[c] + change
    UPDATE-N-TRUE-SETS(c)
    for (g ∈ GAIN-TYPES) do
        if (g.i = n + change) then
            FIRE-GAIN-TYPE(g, v, c)
```

Table 3.6: The set of problems used in the fitness function, which are broken up into five subsets of problem instances. All of the problems are 3-SAT problem instances around the phase transition region. We show the relevant data for each subset. "Max Flips" refers to the maximum number of flips allowed before termination. "Pass Criteria" describes the criteria used to decide whether to run the heuristic on the next subset of problems. A heuristic terminates early if it cannot solve a single SAT problem in the first, then the second subset of problem instances.

Problem Subset Name	Size of Subset	Variables	Clauses	Max Flips	Pass Criteria
`uf50`	10	50	218	500	# solved > 0
`uf100`	15	100	430	750	# solved > 0
`uf150`	20	150	645	1,000	None
`uf200`	25	200	860	2,000	None
`uf250`	30	250	1,065	3,000	None

In Table 3.6 we show details of the problem sets which make up the fitness function. All of the SAT problem instances have been taken from the SAT benchmark suite[1].

When a heuristic is ran on a problem, the LS-SAT algorithm creates an initial problem assignment randomly. In Table 3.6 each subset has an entry titled "Max Flips", which tells the LS-SAT algorithm the maximum number of flips every problem within that subset can run for before the algorithm will terminate.

In the fitness function we use a mechanism that allows the fitness evaluation of a heuristic to terminate early. It works by analysing the number of instances solved in the current set and, if it's deemed insufficient, the evaluation terminates early. In Table 3.6 there is a column titled "Pass Criteria". This column shows the criteria used to judge whether to terminate early. If a heuristic is unable to solve any of the SAT problem instances in the first set, it terminates early. This also occurs in the second set of SAT problem instances. This early termination mechanism is inspired by a similar technique used by Fukunaga [60, 63, 61]. We use this mechanism to ensure that we do not spend unnecessary time evaluating ineffective heuristics.

[1]Located at `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`. Specifically the sets `uf50`, `uf100`, `uf150`, `uf200` and `uf250`. The problems used are the first N from these sets.

Table 3.7: The fitness values of the heuristics shown in Figure 3.2 according to Equations (3.6) and (3.7).

Heuristic	Fitness
GSAT	6.800543
WALKSAT	21.500070
NOVELTY	26.800062
G^2WSAT	31.000062
GNOVELTY+	33.000054
PROBSAT	22.400061
SW_{CC}	46.100031
DEPTH-2-2	30.300056

Once h has been ran on all the required problem instances, the results are used to compute $F(h)$ according to Equation (3.6).

$$F(h) = \#\ problems\ solved + \frac{1}{\#\ flips\ on\ satisfying\ runs} \tag{3.6}$$

Five separate $F(h)$ values are calculated. The fitness f of a heuristic h is calculated according to Equation (3.7).

$$f(h) = \frac{1}{5}\sum_{i=1}^{5} F_i(h) \tag{3.7}$$

f's range is be between 0 and 101. The reader should note that, since the heuristics are stochastic functions, a heuristic will not necessarily report the same fitness every time it is ran through the fitness function.

3.4.2 Fitness of Known Heuristics

In this subsection we use previously known, effective heuristics to "benchmark" our fitness function. When designing the fitness function, we wanted previously known heuristics to report a reasonable score, while leaving room for new heuristics to improve on this. The results in this subsection will allow us to compare the fitness values of heuristics found through program synthesis to hand-crafted ones. We tested the fitness function against the set of known SAT heuristics shown in Figure 3.2, the results of which are presented in Table 3.7.

We can see from these results that the reported fitnesses for these known heuristics are relatively low; from a maximum score of 101 no heuristic is able to achieve a score even half of this. The best performing heuristic according to the fitness function is SW_{CC}. The earlier described heuristics - GSAT, WalkSAT and Novelty - reported the lowest fitness values, while the more modern heuristics - G^2WSAT and gNovelty+ - reported slightly higher fitness values. The automatically created heuristic Depth-2-2 performs similarly to the more modern heuristics. ProbSAT is described as one of the state-of-the-art LS-SAT heuristics in Section 2.3, yet it performs relatively poorly in comparison to the other heuristics on the fitness function.

In the next subsection we describe the testing set of SAT problem instances, and run the heuristics shown in Figure 3.2 on them.

3.4.3 Testing Set

The fitness function described in Section 3.4.1 will be used to assign a fitness to the automatically created heuristics. However, the SAT problem instances in the fitness function are small and relatively easy to solve. We use a testing set of problem instances to perform a more rigorous evaluation of some of the created heuristics. Through the testing set, we hope to gain insight into how the created heuristics perform on larger problem instances, and how their performance compares to hand-crafted heuristics.

The testing set of SAT problems is outlined in Table 3.8. Like the fitness function, it is comprised of several subsets of problem instances. The first five subsets are taken from the SATLIB[2], and the remaining subsets are from the Random track at the 2009 SAT Competition[3]. All of the problems are 3-SAT instances in and around the phase transition region.

The termination criteria for the testing set is based on time rather than the maximum number of flips. Each subset in Table 3.8 has an entry titled "Max Time", which is how long each heuristic is ran on each problem instance before it terminates. The testing set uses this termination criteria as, in a real-world setting, flips are not necessarily a good indicator of a heuristic's performance. Some heuristics use many additional data structures in their formulation, which have to be updated as

[2]The sets of SAT problem instances can be found at `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.

[3]The sets of SAT problem instances can be found at `http://www.satcompetition.org/2009/`.

Table 3.8: The set of problems used in the testing set, which are broken up into eleven subsets of problem instances. All of the problems are 3-SAT problem instances around the phase transition region. We show data for each subset. “Max Time” is the maximum time in seconds allocated to each of the problem instances in that set.

Problem Subset Name	Size of Subset	Variables	Clauses	Max Time
`uf50`	1,000	50	218	1
`uf100`	100	100	430	10
`uf150`	100	150	645	10
`uf200`	100	200	860	10
`uf250`	100	250	1,065	10
`ufv4000`	10	4,000	16,800	100
`ufv7000`	10	7,000	29,400	100
`ufv10000`	10	10,000	42,000	100
`ufv13000`	10	13,000	54,600	100
`ufv16000`	10	16,000	67,200	100
`ufv19000`	10	19,000	79,800	100

the algorithm is running. This can increase the overall computation time of a single iteration of local search. By measuring against time, we gain a better understanding of how effective the heuristics truly are.

In Table 3.9 we show how the heuristics in Figure 3.2 perform on the testing set. From these results, we can see that generally all of the heuristics perform well on the first five subsets of problem instances, with the exception of GSAT. These subsets contain smaller SAT problem instances, which are usually easier to solve than larger ones. If we look at the timing information, we can see that SW_{CC} could consistently find satisfying solutions more quickly than the other heuristics, though it was unable to solve as many instances as the most effective heuristics.

On the first five subsets of problem instances, gNovelty+ was the best performing heuristic. It could consistently solve all problem instances in each subset (apart from on `uf200`, which WalkSAT was able to outperform it on), and do this more quickly than the other heuristics that also solved all of the problem instances.

On the subsets containing larger problem instances, most of the heuristics were

Table 3.9: Results from running the heuristics in Figure 3.2 on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subset of problem instances.

	Heuristic							
Subset Name	GSAT	WalkSAT	Novelty	G^{2}WSAT	gNovelty+	ProbSAT	SW$_{cc}$	Depth-2-2
`uf50`	43.0 0.0004	100.0 0.0008	99.8 0.0011	100.0 0.0008	**100.0** **0.0005**	100.0 0.0008	99.6 0.0006	99.7 0.0005
`uf100`	33.0 0.0008	100.0 0.0046	100.0 0.0051	100.0 0.0049	**100.0** **0.0027**	100.0 0.0039	97.4 0.0024	98.8 0.0022
`uf150`	16.0 0.0045	100.0 0.0168	100.0 0.0286	100.0 0.0244	**100.0** **0.0085**	100.0 0.0132	98.0 0.0081	99.0 0.0198
`uf200`	13.0 0.0465	**99.6** **0.1518**	98.8 0.111	99.0 0.1107	99.2 0.059	99.0 0.0666	92.4 0.0255	95.4 0.0813
`uf250`	13.0 0.0715	99.8 0.0701	99.8 0.1369	99.0 0.0782	**100.0** **0.1069**	99.0 0.04	97.0 0.054	93.2 0.2197
`ufv4000`	0.0 0.0	12.0 3.8668	40.0 24.0208	10.0 0.9327	74.0 21.5379	**80.0** **16.234**	0.0 0.0	0.0 0.0
`ufv7000`	0.0 0.0	6.0 4.6592	0.0 0.0	2.0 0.5891	60.0 19.8728	**90.0** **21.9236**	0.0 0.0	0.0 0.0
`ufv10000`	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	8.0 6.4685	**34.0** **15.2547**	0.0 0.0	0.0 0.0
`ufv13000`	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	**26.0** **22.1408**	0.0 0.0	0.0 0.0
`ufv16000`	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
`ufv16000`	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0

unable to solve many of the problems. The exceptions to this were gNovelty+ and ProbSAT. However both heuristic's performance degraded as larger problems were considered, with gNovelty+ unable to solve any instances in any subset after `ufv10000`, and ProbSAT unable to solve any instances in any subset after `ufv13000`.

If we compare and contrast these results to those shown in Table 3.7, we can state that, in general, the performance of the heuristics on the testing set correlates with their performance on the fitness function with two clear exceptions. SW_{CC} doesn't perform as well as its fitness suggested, and ProbSAT performs much better than its fitness suggested. For the former of these, we believe that SW_{CC} reported a much higher fitness due to the way in which the fitness function is formulated. SW_{CC} requires the updating of auxiliary data structures that are relatively computationally expensive to maintain. When evaluating its performance through time taken, we can see that when it does find a solution, it does so quickly. We believe that these auxiliary data structures slow the running of the heuristic, and since the fitness function relies on flips, this is not apparent in its reported fitness. It is still an effective heuristic, but one that relies on picking the "correct" variable, rather than quickly moving through the search space.

On the other hand ProbSAT does not use data structures that require computationally expensive update operations. It is designed to be able to move through the search space quickly and consider many different solutions in order to find a satisfying one. When used on the fitness function, it is unable to do this, as it is limited by the number of flips it can perform. However on the testing set, we believe it is able to consider many more states than other heuristics and, when combined with its effective design, appears to be highly successful at solving many different sized problem instances.

3.5 Discussions & Conclusions

In this chapter we have provided an overview of how we represent heuristics in this thesis, described the underlying architecture that allows us to run these heuristics, and shown the way in which we evaluate heuristics using a fitness function and a testing set of SAT problem instances.

This chapter is technical in nature, however there are some key observations that can be drawn from the work described. Through our representation, we have

designed a language that can describe many different types of heuristic in a single form. We are able to write representations of many previously existing heuristics, yet the representation allows us to compose these different heuristic methodologies in a unified manner. This was one of our goals in the design of our heuristic evaluation software, as we can now apply program synthesis techniques to this domain, and potentially create new, effective heuristics.

To design a system that can evaluate any potential heuristic that could be represented by this language, we had to develop methods of analysing these heuristics, as well as design an LS-SAT solver that could react to the requirements of a heuristic. Through this work, we identified a general method of defining gain types, and presented mechanisms used to maintain the auxiliary data structures required by each heuristic. Of particular note are the algorithms designed to update an arbitrary gain type metric in constant time. In our research, we are aware of mechanisms to compute the weighted and non-weighted PosGain, NegGain and NetGain in constant time [15], however we have found no descriptions of algorithms that can do this for the Sub variants we use.

Using the results obtained from the evaluation of previously described heuristics on the fitness function and the testing set, we will now be able to compare the performance of automatically created heuristics to these previously known LS-SAT heuristics. These results have also provided us with evidence that a heuristic which performs well on the fitness function may not necessarily be successful at solving many different sized problem instances. SW_{CC} reported a high fitness value, yet did not provide a high-level of performance on the testing set of problem instances. On the other hand, ProbSAT did not report one of the higher fitness values on the fitness function, but outperformed all other heuristics on the larger problem instances in the testing set.

In the next chapter we use the systems we have developed in this chapter with two program synthesis methods, exhaustive enumeration and GP, to automatically create LS-SAT heuristics.

Chapter 4

Exhaustive Enumeration & Genetic Programming

4.1 Introduction

In the previous chapter we provided the reader with specific details on how we represent and evaluate heuristics. In Section 2.5 we described several program synthesis techniques, and in the conclusions to that section discussed which methods we believed to be relevant to our work. Succinctly, we stated that, of the methods described which had not been previously used to automatically create heuristics, exhaustive enumeration appeared to be the most appropriate for the task. In this chapter, we perform exhaustive enumeration on subsets of the DSL described in Section 3.2. The aim of the work in this chapter is to ascertain whether exhaustive enumeration is an appropriate technique to automate the heuristic creation process. The enumeration of heuristics also provides us with the opportunity to perform an analysis on the search space of heuristics described by the subset of the DSL we use.

In this chapter we also perform experiments using GP. GP is an established methodology for automatically creating LS-SAT heuristics. Through the results from the exhaustive enumeration and GP experiments, we are able to compare the two methods of program synthesis for our use-case.

As stated previously, the experiments presented in this chapter are performed on subsets of the DSL described in Section 3.2. We refer to each subset of the DSL as a *language*. The languages used in this chapter are Language A and Language A1. The set of terms in each language are shown in Table 4.1.

Table 4.1: Languages A and A1. All terms shown here are contained in Language A1. Language A does not contain those terms with a star next to them. Specifically, it does not contain WFF. In regards to the GP experiments in Section 4.3, terms with a grey background are in the terminal set, and those with a white background are in the function set.

Type Signature	Terms
VarSet → GainType → Var	{GetBestVar, GetBestVarSnd}
VarSet → Var	{PickRandomVar}
Probability → Var → Var → Var	{IfRandLt}
VarSet → Var → Var → Var	{IfNotMinAge}
Var → Var → Var	{GetOldestVar}
Age → Var → Var → Var	{IfTabu}
Comparator → GainType → Var → Var → Var	{IfVarCompare}
Comparator → GainType → Integer → Var → Var → Var	{IfVarCond}
GainType	{PosGain, NegGain, NetGain}
Age	$\{5, 10, 20, 30, 40, 50\}$
Probability	$\{0.1, 0.3, 0.5, 0.7, 0.9\}$
Integer	$\{-2, -1, 0, 1, 2, 3, 4, 5\}$
Comparator	$\{<, \leq, =\}$
VarSet	{RBC-0, WFF*}

The terms used in Language A were chosen for two primary reasons. Firstly, the set of heuristics that can be created using this language contains two previously known, effective heuristics - WalkSAT and Novelty. When performing exhaustive enumeration, if we enumerate enough of the search space, we are guaranteed to find these previously known heuristics. Secondly, the experiments described by Fukunaga [60, 63, 61] were performed on a similar language to Language A, and several examples of effective heuristics were reported. By using a similar language, we can compare the effectiveness of the heuristics created from our experiments and those created from Fukunaga's work.

Language A1 is almost identical to Language A, except that it includes the additional term `WFF`. Language A can be described as a subset of Language A1 - all terms that appear in Language A appear in Language A1. The addition of the `WFF` term allows us to explore a search space that contains other previously described heuristics such as GSAT. It also allows us to compare the two languages, and determine the effect that this additional term has on the topology of the search space. In addition to this, the inclusion of `WFF` means that Language A1 more closely resembles the language used by Fukunaga [60, 63, 61].

The reader may note that in Section 3.2 we stated that our DSL could describe the same set of heuristics that Fukunaga's language could. Yet in this chapter we only use languages that can represent a proportion of the heuristics described by Fukunaga's language. In Fukunaga's work [60, 63, 61], there was no information given pertaining to the range of values that terms with a type of `Age`, `Integer` or `Probability` could have. In our languages, we include a range of terms with these type signatures, however it is unlikely that the chosen terms are identical to those used by Fukunaga, and therefore we cannot be certain that any language we create is identical to that used by Fukunaga.

The format of this chapter is as follows; in Section 4.2 we present the exhaustive enumeration experiments, and in Section 4.3 the GP experiments. In each of those sections we show the methodology used and the results obtained from that section's experiments. We also highlight some of the best performing heuristics which were created, and run them on the testing set of problem instances described in Section 3.4.3. Finally in Section 4.4 we present our discussions and conclusions from the work described in this chapter.

4.2 Exhaustive Enumeration Experiments

In this section we detail the exhaustive enumeration experiments performed using Languages A and A1.

The format of this section is as follows; in Section 4.2.1 we discuss the number of heuristics that could potentially be created by an exhaustive enumeration of the languages, and from this data determine how much of the search space we enumerate. In Section 4.2.2 we detail the methodology of the exhaustive enumeration experiments. In Sections 4.2.3 to 4.2.6 we present the results of the experiments. In Section 4.2.3 we present the results in the order in which they are generated. In Section 4.2.4 we show results pertaining to the distribution of the fitness values of the evaluated heuristics. In Section 4.2.5 we present data gathered concerning how quickly the created heuristics were evaluated. Finally in Section 4.2.6 we show specific examples of heuristics reported to be effective according to the fitness function, and show how they perform on the testing set.

4.2.1 Search Space Size

The search space of the languages described in Table 4.1 is infinite, and therefore an enumeration of all heuristics is impossible. We must determine what limitations are to be placed on the exhaustive enumeration experiments to ensure that they terminate in a reasonable amount of time. We will use the size of a heuristic to determine whether it is to be evaluated, thereby making the search space finite.

To be clear to the reader, when we refer to the size of a heuristic, we refer to the number of terms in that heuristic's program tree representation.

To aid our understanding of the size of the search space of an arbitrary language, we created an algorithm based on BOTTOM-UP-SEARCH (see Algorithm 2.18) that, when given a language L and size d, can determine the number of program trees of exactly size d in L. In Table 4.2 we show the exact number of heuristics in Languages A and A1 of specific sizes, and in Figure 4.1 we show this data graphically. These results show that, for both languages, the number of heuristics grows exponentially in relation to the size of the heuristic, and tells us that it would be impractical to enumerate all heuristics of a large size.

We can also make an observation about how the size of the search space of Language A and Language A1 differs. Language A1 is virtually identical to Language

Table 4.2: The number of heuristics of a specific size in Languages A and A1. For each size d, we show the exact number of program trees of size d in each language.

d	Language A	Language A1	d	Language A	Language A1
1	0	0	11	12,173	108,368
2	1	2	12	62,238	558,144
3	6	12	13	223,155	2,017,536
4	0	0	14	714,542	7,155,312
5	1	4	15	2,264,475	29,360,224
6	24	100	16	8,040,276	133,042,608
7	189	804	17	32,104,239	585,809,872
8	614	2,608	18	116,861,220	2,331,681,856
9	1,272	5,456	19	414,649,530	9,448,276,608
10	3,996	22,576	20	1,440,234,132	40,105,399,680

A, and only contains one additional term. Yet, the number of heuristics in Language A1 is much larger than in Language A. We assume that, if we were to use the complete DSL in Section 3.2, the search space will grow even more quickly, and at low sizes, enumeration would not be a suitable strategy for program synthesis.

Based on this data, we decided on a maximum size of 17 for exhaustively enumerating Language A, and a maximum size of 15 for Language A1. As Language A1 contains all terms in Language A, by enumerating Language A1 we will enumerate all heuristics in Language A. Therefore in conducting these experiments we will evaluate exactly 79,375,663 unique heuristics. We note that even at this size, several previously described heuristics will be recreated; specifically, WalkSAT (size 13), Novelty (size 14) and, exclusively for Language A1, GSAT (size 3).

4.2.2 Methodology

The methodology of the exhaustive enumeration experiments we describe as follows; given a size d and a language L, we used an algorithm based on the Top-Down-Search algorithm (see Algorithm 2.17) to enumerate all program trees in L containing exactly d terms. Each created program tree was then evaluated against the fitness function described in Section 3.4.1. We repeated the experiment for the range of d values required to obtain the desired set of results for Languages A and A1.

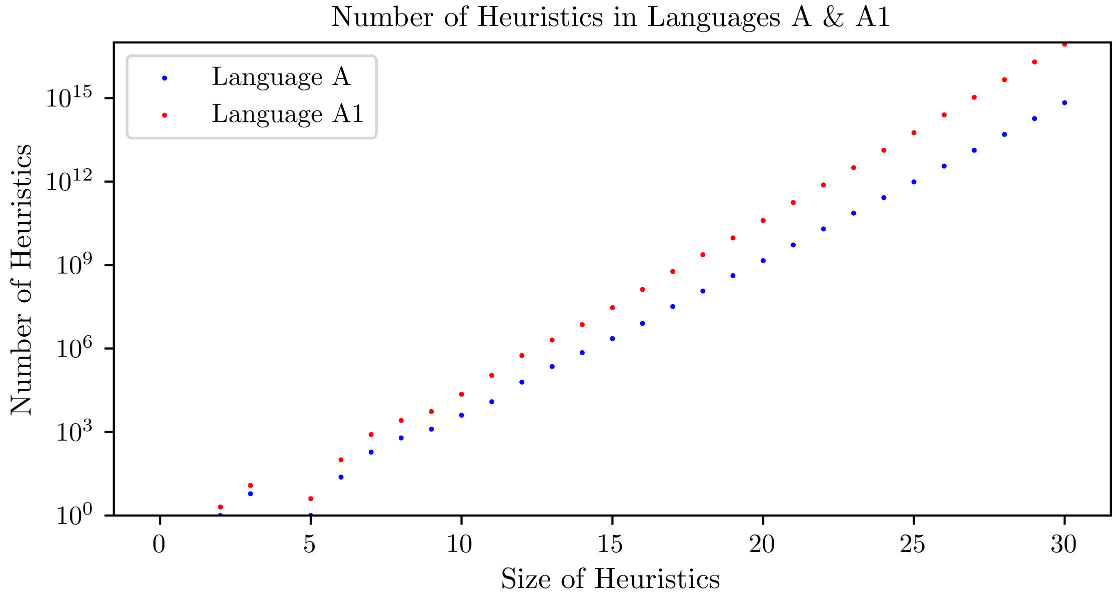


Figure 4.1: The number of heuristics of a specific size in Languages A and A1.

When performing the experiments, we provided an explicit ordering of terms in the language to the enumeration algorithm, which affected the order that the heuristics were enumerated in. We present some of the results using this ordering. The algorithm that enumerated the program trees works by conceptually exploring a search tree like that shown in Figure 2.9. When progressing the search, for any type hole in a partial program, all possible instantiations of that type hole are made, and the new partial program trees put back into the set of unprocessed partial program trees. This means that the heuristics from our experiments were created in a specific order, with all heuristics with the same first term grouped together, then the same first two terms and so on. In effect, if we were to consider the results as a stream, any heuristic in the stream when compared to the previous would usually only differ by a single term.

The ordering of terms that we used to instantiate a type hole can be described as follows; for any terms with the correct return type, the terms are ordered first by type signature size, then the term's type signature's lexicographical ordering and, if both are the same, then the term's lexicographical ordering.

The experiments were conducted on a computer with 2 Intel Xeon E5-2630 processors with 6 cores (12 threads) each, running at 2.6GHz. The system uses a 64-bit operating system and has 32GB of RAM. The software is written in C++, and

is able to utilise all the cores on the machine it is ran on. In total, these experiments took 13 days to run on this machine.

4.2.3 Results in Generated Order

In this subsection we show some of the results obtained from the exhaustive enumeration of Languages A and A1. The heuristics in this section are presented in the order that they are generated in by the enumeration algorithm.

Small Sized Heuristics

In Table 4.2 we showed the number of heuristics of specific sizes in the two languages. For very small sizes (≤ 5), the heuristics can be listed, as there are few in number. In Table 4.3 we show these heuristics.

We can see from these results that, when compared to the fitness values reported in Section 3.4.2 for hand-crafted heuristics, these are not particularly effective heuristics. We can also see that the heuristic GSAT is recreated. However, it is not the best performing of the heuristics shown. In a rather surprising result, two heuristics that choose the second best variable according to $\textsc{NetGain}_1$ and $\textsc{NegGain}_1$ perform slightly better than GSAT. However compared to other heuristics, such as those seen in Section 3.4.2, these heuristics have a relatively low fitness value.

Larger Sized Heuristics

The number of heuristics of a larger size are much greater in number. Listing them all would be impractical, and we therefore show the results in a series of graphs. The full set of graphs can be found in Appendix A. We present some results in this subsection. Specifically, we show the results for Language A at sizes 10, 17, 13 and 14 in Figures 4.2, 4.3, 4.6 and 4.7a respectively. We also present partial results for the heuristics in Language A1 of size 14 in Figure 4.7b. To be clear to the reader, in all these graphs each data point represents the fitness of an individual heuristic.

Table 4.3: The set of heuristics of a small size in Languages A and A1. The index on the left represents the order in which the heuristics were generated in for that size. The first number is the index of the heuristic in Language A1, and the second in Language A.

Index	Size	Heuristic	Fitness
1 (1)	2	PickRandomVar { RBC-0 }	0.4
2 (-)		PickRandomVar { WFF }	0.0
1 (1)	3	GetBestVar { RBC-0, NegGain }	4.0
2 (2)		GetBestVar { RBC-0, NetGain }	4.0
3 (3)		GetBestVar { RBC-0, PosGain }	1.0
4 (-)		GetBestVar { WFF, NegGain }	0.4
5 (-)		GetBestVar { WFF, NetGain }	8.6
6 (-)		GetBestVar { WFF, PosGain }	0.0
7 (4)		GetBestVarSnd { RBC-0, NegGain }	11.8
8 (5)		GetBestVarSnd { RBC-0, NetGain }	10.8
9 (6)		GetBestVarSnd { RBC-0, PosGain }	1.8
10 (-)		GetBestVarSnd { WFF, NegGain }	0.4
11 (-)		GetBestVarSnd { WFF, NetGain }	2.0
12 (-)		GetBestVarSnd { WFF, PosGain }	0.2
1 (1)	5	GetOldestVar { PickRandomVar { RBC-0 }, PickRandomVar { RBC-0 }}	0
2 (-)		GetOldestVar { PickRandomVar { RBC-0 }, PickRandomVar { WFF }}	0
3 (-)		GetOldestVar { PickRandomVar { WFF }, PickRandomVar { RBC-0 }}	0
4 (-)		GetOldestVar { PickRandomVar { WFF }, PickRandomVar { WFF }}	0

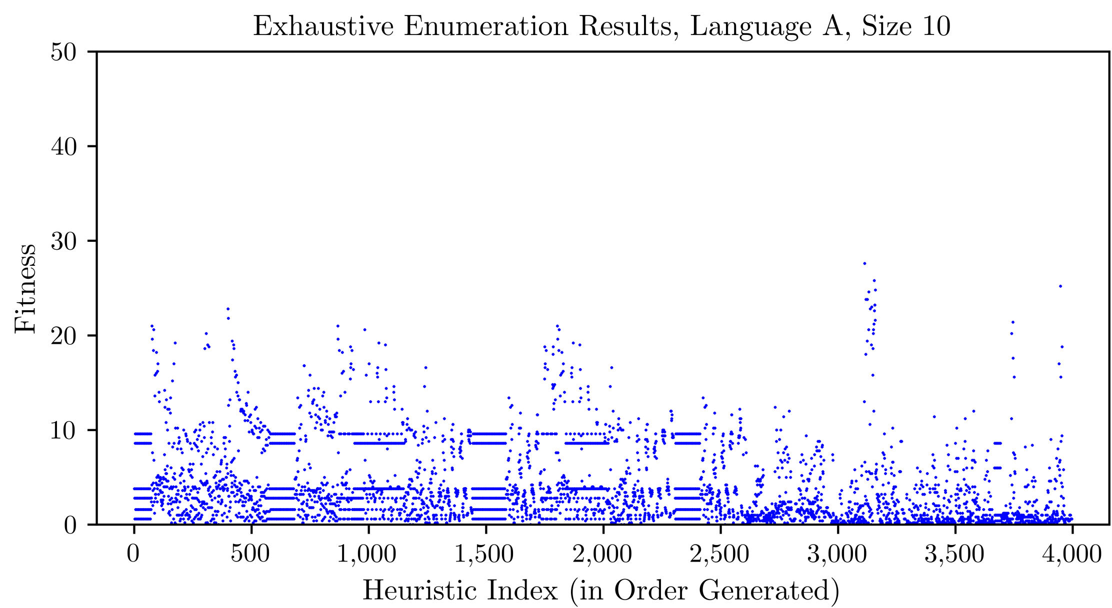


Figure 4.2: Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 10. The heuristics are presented in the order they are generated in.

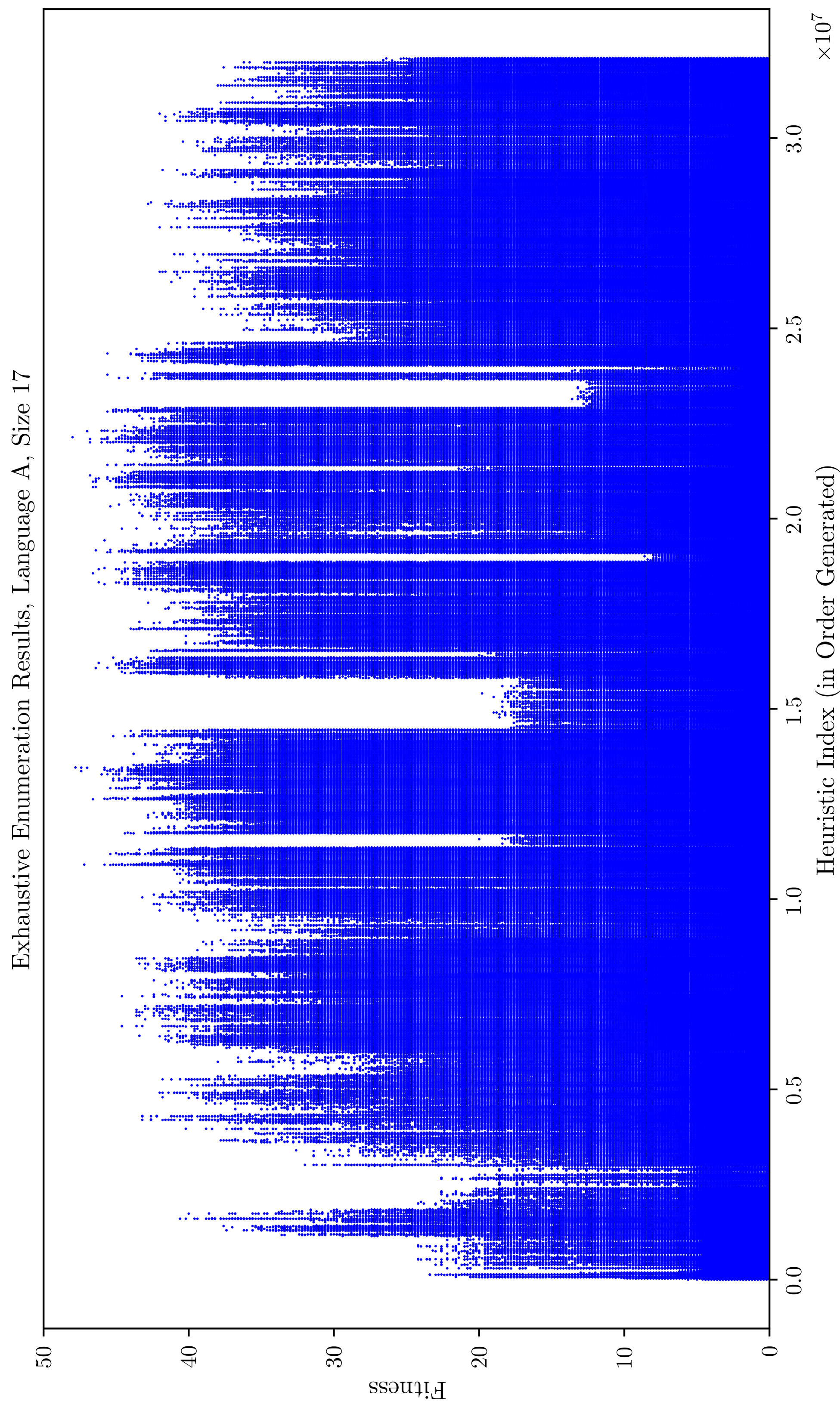


Figure 4.3: Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 17. The heuristics are presented in the order they are generated in.

Analysis of Results

In later subsections we discuss the distribution of fitnesses, as well as specific examples of effective heuristics. However, there are some interesting observations that can be made from the heuristics presented in the order they were generated in.

When examining the results in general, what is perhaps most striking is the density and distribution of heuristics. Comparatively, there are many effective and poorly performing heuristics of all sizes. Effective heuristics exist at many areas in the search space, and are not clustered in a single area. Some graphs have clear peaks - areas where there are many heuristics with a high fitness value. Figure 4.3 contains many examples of these peaks. In Figure 4.4 we show a subset of the heuristics from index $500,000$ to $550,000$ in Language A of size 14. In that graph we can clearly see several distinct areas where there are many effective heuristics, and areas where there are none at all. For example, between index $500,000$ and $510,000$ there are two peaks, where concentrations of effective heuristics are clustered together in "strips".

As the order that the heuristics are generated in controls the topology of the results when visualised in this manner, this ordering is key to understanding these peaks. The relationship between one heuristic and the next can be defined as an "increment" of the first heuristic. By this we mean, the last term is incremented according to the ordering imposed by the language. If it is not possible to do so, then the terminal is removed, the second to last term that was instantiated is incremented, and the last term instantiated with the first valid term according to the ordering of the language.

Terms close to each other in the order generated can be grouped together according to how many of their first n terms are the same. To illustrate this, in Figures 4.5 and 4.6 we present all heuristics in Language A of size 13 in a series of graphs. In Figure 4.5 we present the heuristics coloured according to their first term, and in Figure 4.6 we present the same heuristics, but colour them according to their leading 2 and 3 terms. In the graphs in Figure 4.6 we alternate between red and blue when any one of these first n terms changes compared to the previous heuristic, as showing an individual colour for each unique set of terms would be impractical.

The separation of peaks becomes more pronounced when the heuristics are presented like this. We can clearly see in Figures 4.6a and 4.6b there are peaks that are only one colour, suggesting that they exclusively contain heuristics with the same leading 2 or 3 terms. This is not always the case; there are also examples of peaks

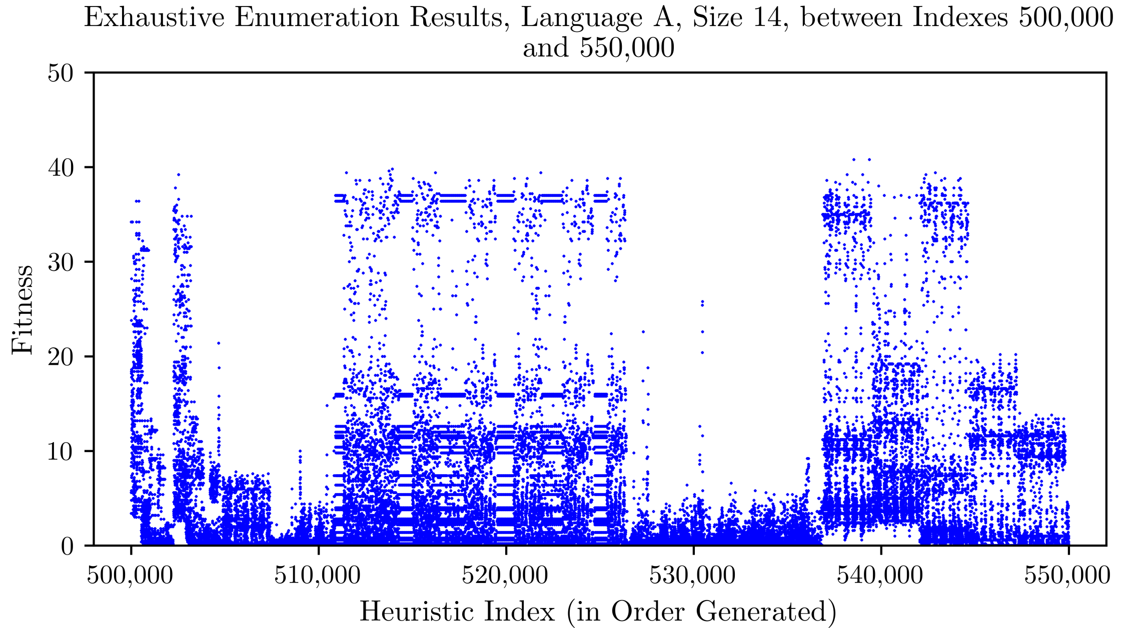


Figure 4.4: Results from the exhaustive enumeration experiments, showing the fitness values for a subset of heuristics in Language A of size 14. We show the heuristics from index $500,000$ to index $550,000$.

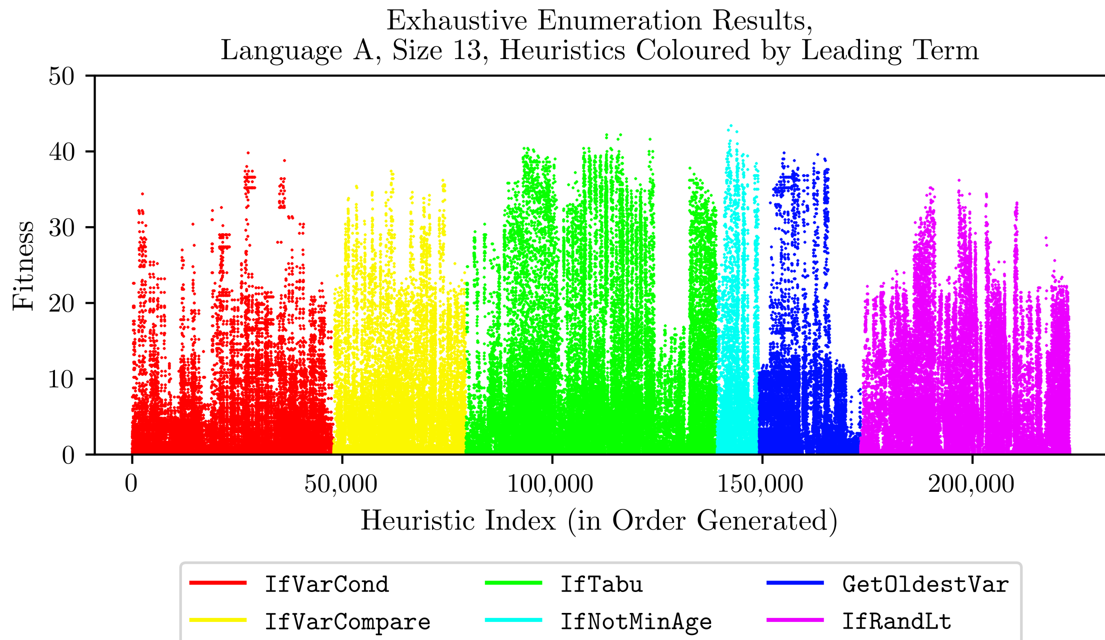
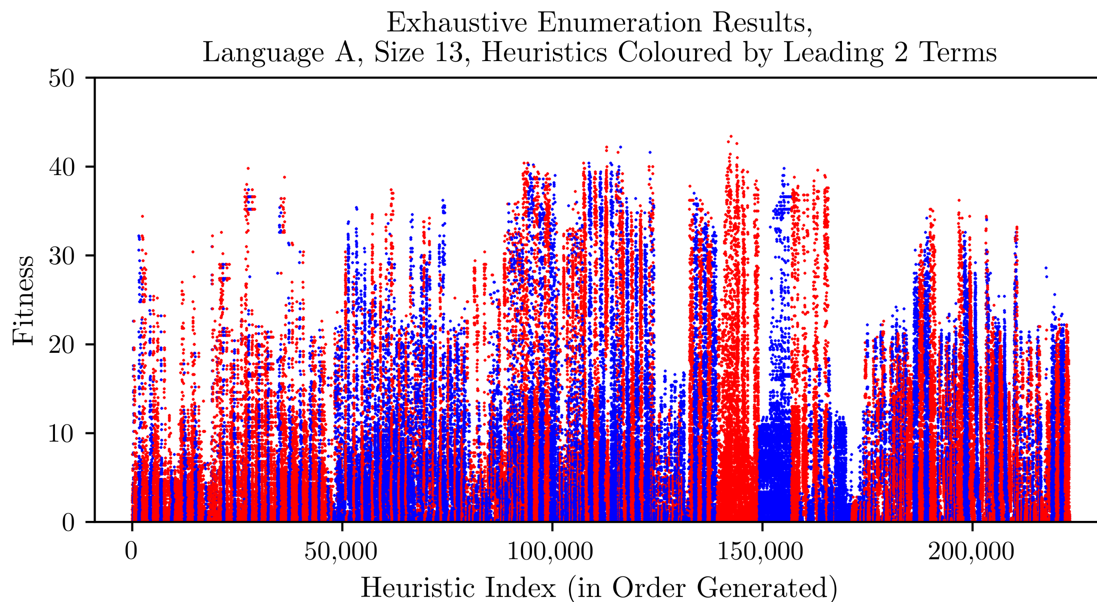
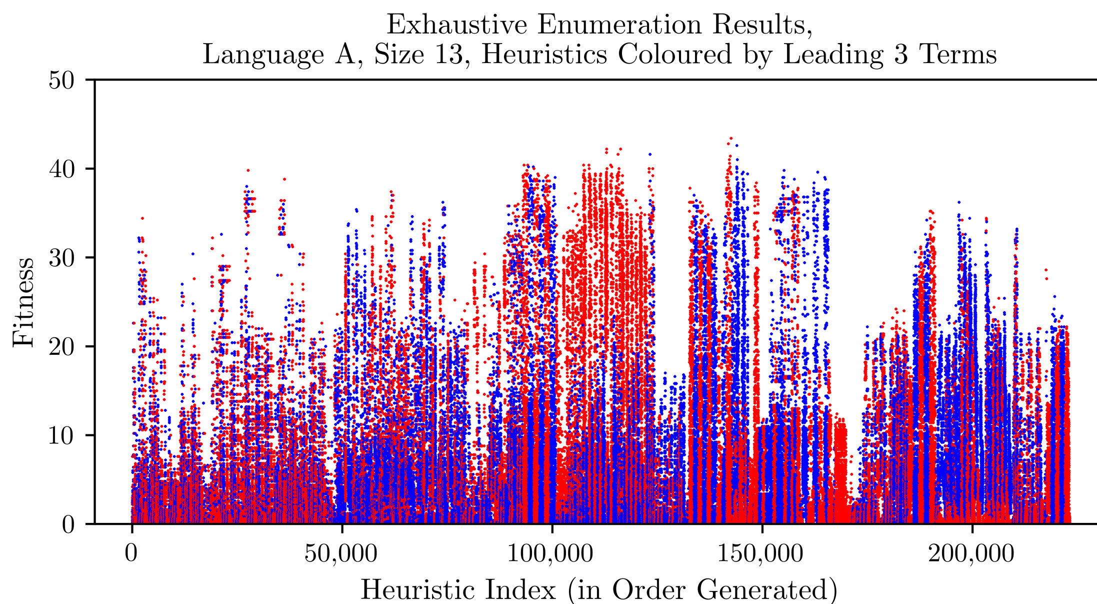


Figure 4.5: Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 13. Each heuristic has been coloured according to its first term.



(a) Results from the exhaustive enumeration experiments. Each heuristic has been coloured according to its first two terms.



(b) Results from the exhaustive enumeration experiments. Each heuristic has been coloured according to its first three terms.

Figure 4.6: Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 13. We colour each heuristic according to its leading n terms. Colours change between red and blue whenever a heuristic's leading n terms change when compared to the previous heuristic's leading n terms.

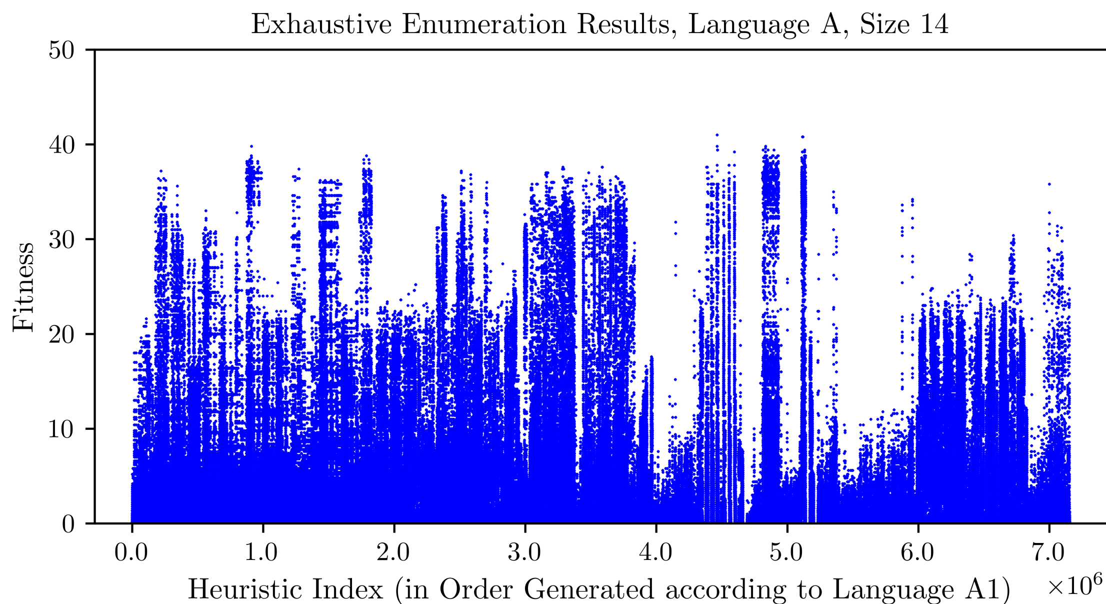
that transcend several different colours. However, we can state that some peaks only contain program trees with a very specific formulation of leading terms, which correspond to effective heuristics.

We want to be clear that this relationship between leading term(s) and heuristic fitness is not exact; there are many examples of poorly performing heuristics with the same leading term(s) as effective heuristics. However, these results do suggest that analysing the search space by examining a heuristic's neighbours could be an effective strategy for finding and navigating these peaks, which could in turn lead to heuristics with a high fitness value. For sizes of heuristic greater than those considered in this chapter, it could be a viable alternative to enumeration of the search space.

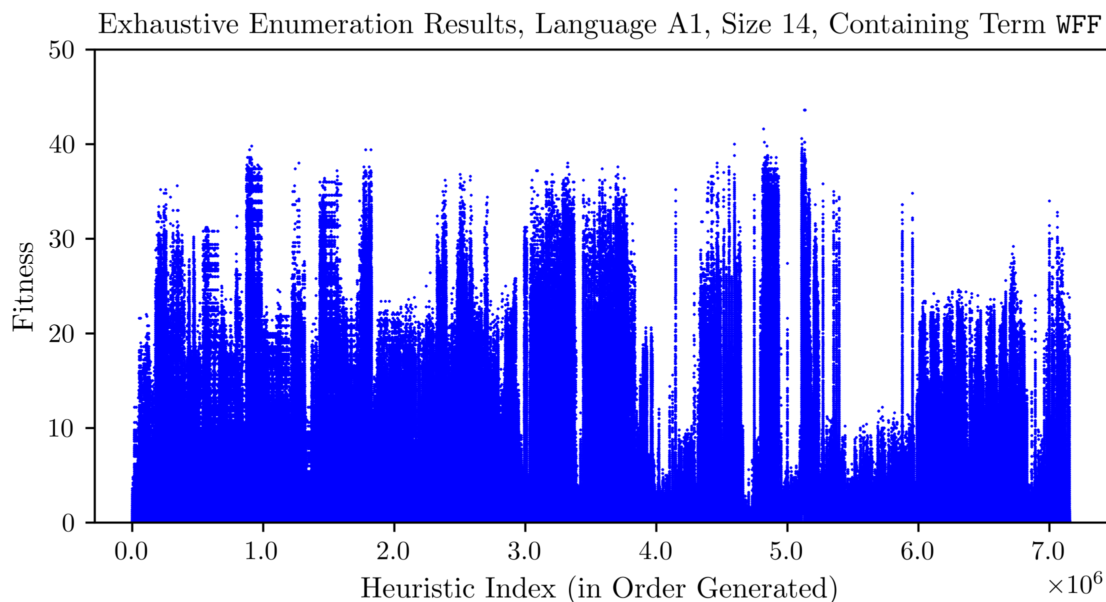
In Figure 4.7 we show the results obtained for heuristics of size 14 from Languages A and A1. Figure 4.7a shows the heuristics in Language A, and Figure 4.7b shows the heuristics in Language A1 which are not in Language A. The heuristics shown in Figure 4.7a are indexed according to their generation ordering from Language A1. This gives us some indication as to the effect that adding the `WFF` term to Language A1 had on the results when compared to Language A.

We can clearly see that the graphs are generally similar to each other. That is to say, the peaks and troughs in both graphs are at the same indexes. For example, at around index $1,000,000$ there is a clear peak in both graphs. However in Figure 4.7b the peak is wider and contains a greater number of data points at lower fitness values. At around index $6,000,000$ in Figure 4.7b there is a clear peak of heuristics with high fitness values, yet in Figure 4.7a this area contains very few high-quality heuristics. The similarity between graphs suggests to us that there are core structures of program tree which correspond to heuristics with high fitness values. Which term is used to instantiate nodes that require a term with a type signature of `VarSet` is not necessarily important; rather it is how the other terminals are combined together that play a greater role in the overall effectiveness of the heuristic. However, areas of the graph relative to each other where there are less effective heuristics reported suggests that this is not always true. The picking of a random broken clause may be imperative to how effective a heuristic is, and this cannot be substituted for a `WFF` terminal without degrading the heuristic's quality.

There is one other interesting phenomenon that we would like to draw the attention of the reader to. Returning to the results in Figure 4.2, we can clearly see there are examples of horizontal "lines" of heuristics with, seemingly, the exact same fitness.



(a) Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A of size 14. Each heuristic's index is derived from its generation ordering according to Language A1.



(b) Results from the exhaustive enumeration experiments, showing the fitness values for all heuristics in Language A1 of size 14 that are not in Language A.

Figure 4.7: Results from the exhaustive enumeration experiments, showing heuristics of size 14 in Languages A and A1.

Index	Heuristic	Fitness

581	`IfVarCond` = `PosGain` −1
	`GetBestVar RBC-0 NegGain`
	`GetBestVarSnd RBC-0 NetGain`
		9.6

587	`IfVarCond` = `PosGain` −1
	`GetBestVar RBC-0 NetGain`
	`GetBestVarSnd RBC-0 NetGain`
		9.6

593	`IfVarCond` = `PosGain` −1
	`GetBestVar RBC-0 PosGain`
	`GetBestVarSnd RBC-0 NetGain`
		9.6

Figure 4.8: Three examples of heuristics of size 10 in Language A that return the same fitness value due to the formulation of the language.

By examining these heuristics with our knowledge about SAT and Language A, the reasoning behind these results becomes clear.

In Figure 4.8 we show three examples of heuristics that exist on one of the horizontal planes seen in Figure 4.2. These heuristics all have the same form; get the best variable according to some gain type metric, and compare that variable's PosGain_1 to an integer. The integers in question are all negative. PosGain_1's possible values an only be ≥ 0, therefore, the `IfVarCond` expression will always evaluate to *False* and return the right subtree. We can see that in Figure 4.8, the right subtrees of all three heuristics are exactly the same. Thus, all heuristics in this form will return the same variable to flip each time. It is, in essence, a phenomenon created by the formulation of the language. We discuss this in further detail in Section 4.4, and highlight potential techniques to stop heuristics like this being created.

4.2.4 Fitness Results

In this subsection we present data regarding the heuristic's fitness values. We look at the distribution and the variance of the fitness values reported.

The absolute best heuristic found from the exhaustive enumeration experiments had a fitness of 48.0. This fitness value is higher than any reported from the hand-crafted heuristics considered in Section 3.4.2. In Figure 4.9 we show the heuristics in

Languages A and A1 of size ≤ 12, 13 and 14, presented in fitness order. The data presented in this manner makes it much easier to see the range of fitnesses compared to the results presented in the previous subsection. We can clearly see that there are many poorly performing heuristics, and it is only a very small number that are actually effective. We can also see that for Language A1, the proportion of poorly performing heuristics is much greater than for Language A.

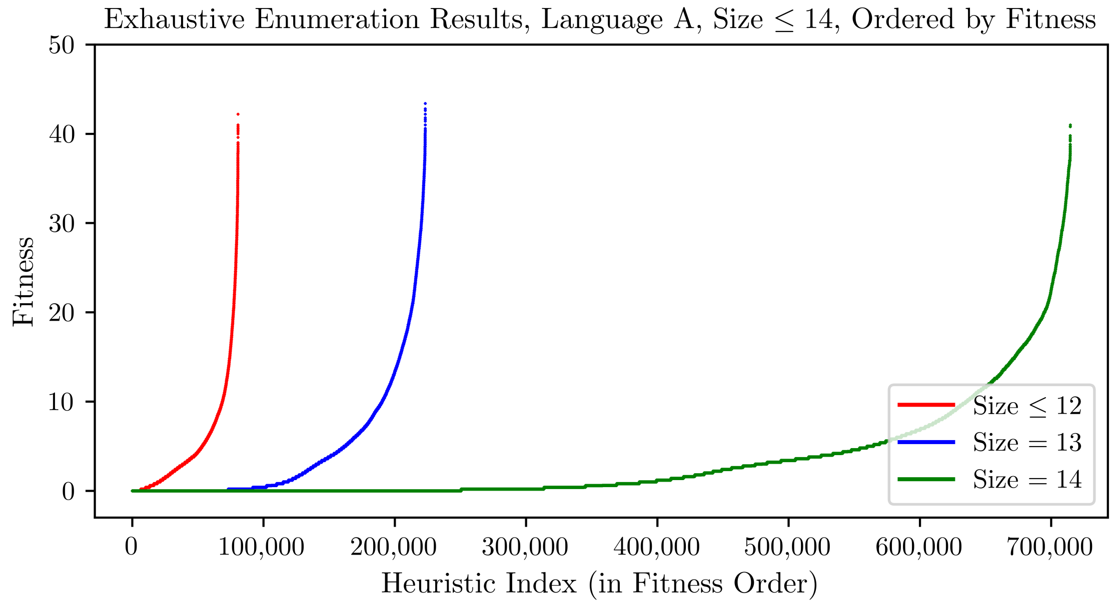
In Table 4.4 (for Language A) and Table 4.5 (for the heuristics in Language A1 not in Language A) we separate the heuristics into groups according to their fitness. From these tables, we can see the distribution of the heuristic's fitness values.

We can see that the percentage of heuristics in both languages that are actually effective according to the fitness function is tiny. Heuristics that have a fitness value > 30 (around the fitness of Novelty according to the results in Section 3.4.2) account for approximately 0.97% of all heuristics in Language A, and approximately 0.16% of all the heuristics in Language A1 that are not in Language A.

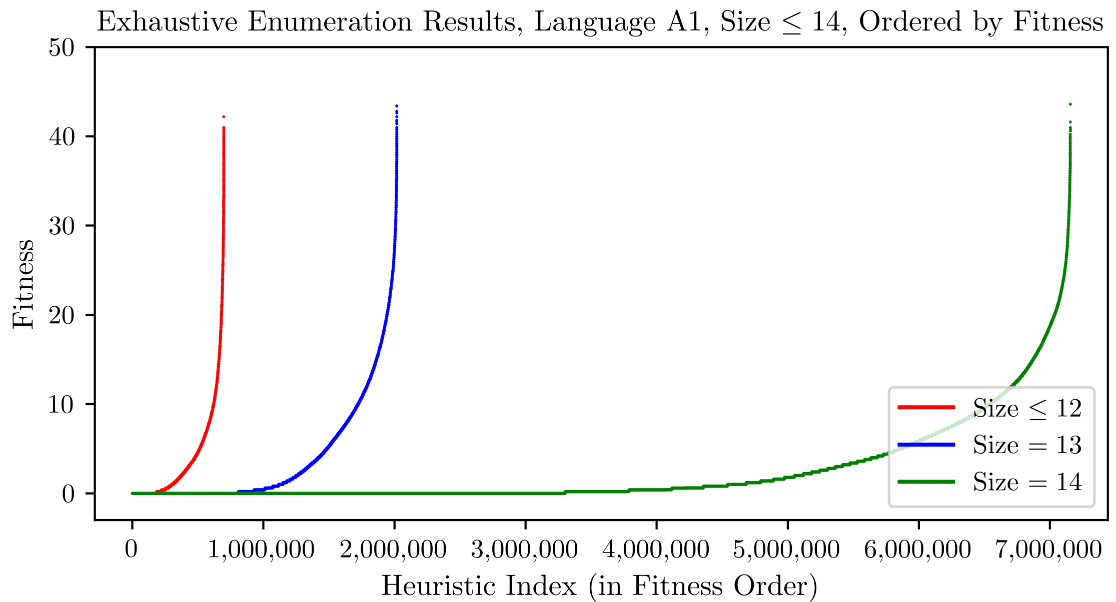
In the results for Language A, the heuristics with a fitness of 0 are not the biggest set, it is those heuristics with a fitness > 0 and ≤ 5. Since these heuristics are based on random walk, we wonder if these are just heuristics that have got "lucky", and been able to solve a single instance. The proportion of the heuristics in Language A1 not in Language A that have a fitness of 0 is much greater than the proportion in Language A. In Section 2.3.2 we showed how even simple heuristics which work by picking a random variable from a broken clause were often more effective than GSAT-like heuristics. This could provide an explanation for these "lucky" heuristics; by continually picking a variable from a broken clause, they were able to solve a small number of the SAT problems in the fitness function simply through chance.

The number of low-quality heuristics in both sets suggests that our decision to use a fitness function that employs an early termination mechanism was a correct one. It undoubtedly saved us many computational hours, as we would have had to evaluate many poorly performing heuristics on all problem instances if we had not used it.

In general, as we consider sets containing heuristics of greater size, we can see by comparing the best heuristics in each subsequent set to the previous, that the number of highly effective heuristics increases. Further to this, we can see that as we consider heuristics of larger sizes, the fitness of the absolute "best" heuristic in those sets increases. Yet, we do wonder whether there is an upper limit for these scores and this fitness function, as we never encountered any heuristics with a fitness above 50.



(a) All heuristics in Language A of size ≤ 14, ordered by their fitness.



(b) All heuristics in Language A1 of size ≤ 14, ordered by their fitness.

Figure 4.9: All heuristics in Languages A and A1 of size ≤ 14. We separate the heuristics in each language into subsets based on their size, and order them by their fitness value.

Table 4.4: The fitness distribution of heuristics in Language A of size ≤ 10 and 11 - 17. We use $f(h)$ to refer to a heuristic's fitness.

(a) The fitness distribution of heuristics in Language A of size ≤ 10, 11 and 12.

	size ≤ 10		size $= 11$		size $= 12$	
Fitness Group	Total	%	Total	%	Total	%
$f(h) = 0$	371	6.08	785	6.45	5,375	8.64
$0 < f(h) \leq 5$	4,095	67.10	7,623	62.62	34,560	55.53
$5 < f(h) \leq 10$	1,101	18.04	2,189	17.98	12,405	19.93
$10 < f(h) \leq 15$	320	5.24	703	5.78	4,620	7.42
$15 < f(h) \leq 20$	146	2.39	345	2.83	2,309	3.71
$20 < f(h) \leq 25$	53	0.87	261	2.14	1,478	2.37
$25 < f(h) \leq 30$	6	0.10	190	1.56	905	1.45
$30 < f(h) \leq 35$	10	0.16	55	0.45	454	0.73
$35 < f(h) \leq 40$	1	0.02	19	0.16	126	0.20
$40 < f(h) \leq 45$	0	0	3	0.02	6	0.01

(b) The fitness distribution of heuristics in Language A of size 13 - 15.

	size $= 13$		size $= 14$		size $= 15$	
Fitness Group	Total	%	Total	%	Total	%
$f(h) = 0$	73,154	32.78	250,573	35.07	792,527	35.00
$0 < f(h) \leq 5$	87,961	39.42	309,010	43.25	1,073,836	47.42
$5 < f(h) \leq 10$	28,840	12.92	75,058	10.50	220,561	9.74
$10 < f(h) \leq 15$	13,586	6.09	37,241	5.21	75,510	3.33
$15 < f(h) \leq 20$	8,730	3.91	22,512	3.15	48,608	2.15
$20 < f(h) \leq 25$	4,505	2.02	8,937	1.25	22,626	1.00
$25 < f(h) \leq 30$	3,391	1.52	5,741	0.80	16,208	0.72
$30 < f(h) \leq 35$	1,993	0.89	3,621	0.51	12,571	0.56
$35 < f(h) \leq 40$	963	0.43	1,846	0.26	1,995	0.09
$40 < f(h) \leq 45$	32	0.01	3	> 0.00	32	> 0.00
$45 < f(h) \leq 50$	0	0.00	0	0	1	> 0.00

Table 4.4: The fitness distribution of heuristics in Language A of size ≤ 10 and 11 - 17. We use $f(h)$ to refer to a heuristic's fitness. (Continued)

(c) The fitness distribution of heuristics in Language A of size 16 and 17.

	size = 16		size = 17	
Fitness Group	Total	%	Total	%
$f(h) = 0$	2,747,171	34.17	10,619,758	33.08
$0 < f(h) \leq 5$	3,816,696	47.47	13,855,443	43.16
$5 < f(h) \leq 10$	813,416	10.12	3,698,471	11.52
$10 < f(h) \leq 15$	274,884	3.42	1,552,900	4.84
$15 < f(h) \leq 20$	162,651	2.02	965,498	3.01
$20 < f(h) \leq 25$	103,201	1.28	624,049	1.94
$25 < f(h) \leq 30$	74,022	0.92	439,287	1.37
$30 < f(h) \leq 35$	35,394	0.44	245,073	0.76
$35 < f(h) \leq 40$	11,531	0.14	91,931	0.29
$40 < f(h) \leq 45$	1,298	0.02	11,712	0.04
$45 < f(h) \leq 50$	12	> 0.00	117	> 0.00

Variance

If we recall the fitness function f described in Section 3.4.1, it is calculated as the average of five repetitions of the F function. The F function computes a numerical value from running the heuristic on the set of problem instances described in Table 3.6. If we were to consider each F value as an individual fitness, then we can study the overall variance of a heuristic's reported fitness value. That is to say, we can determine how reliable each heuristic is at returning a similar F value for each repetition, and whether their reported fitness value has come from a large or small range of F values.

To do this, we took the results from all heuristics in Language A and those in Language A1 not in Language A, and plotted the mean square variance of their F values against their reported fitness values. These results can be seen in Figure 4.10. A high variance reported would suggest that a heuristic is not necessarily reliable. A low variance would suggest a heuristic consistently performs well.

By studying these graphs, we can see that generally the variance of the reported fitness values is low. We can also see that there is no correlation between fitness and variance; if there was a positive correlation, this would suggest that heuristics

Table 4.5: The fitness distribution of heuristics in Language A1 not in Language A of size ≤ 10 and 11 - 15. We use $f(h)$ to refer to a heuristic's fitness.

(a) The fitness distribution of heuristics in Language A1 not in Language A of size ≤ 10, 11 and 12.

	size ≤ 10		size $= 11$		size $= 12$	
Fitness Group	Total	%	Total	%	Total	%
$f(h) = 0$	10,345	40.63	35,484	36.89	133,999	27.02
$0 < f(h) \leq 5$	11,701	45.96	42,696	44.38	223,604	45.09
$5 < f(h) \leq 10$	2,617	10.28	10,556	10.97	79,675	16.07
$10 < f(h) \leq 15$	626	2.46	3,720	3.87	31,994	6.45
$15 < f(h) \leq 20$	132	0.52	1,789	1.86	14,019	2.83
$20 < f(h) \leq 25$	25	0.10	1,059	1.10	7,495	1.51
$25 < f(h) \leq 30$	11	0.04	602	0.63	3,580	0.72
$30 < f(h) \leq 35$	2	0.01	236	0.25	1,331	0.27
$35 < f(h) \leq 40$	2	> 0.00	51	0.05	207	0.04
$40 < f(h) \leq 45$	0	0.00	2	> 0.00	2	> 0.00

(b) The fitness distribution of heuristics in Language A1 not in Language A of size 13 - 15.

	size $= 13$		size $= 14$		size $= 15$	
Fitness Group	Total	%	Total	%	Total	%
$f(h) = 0$	735,977	41.02	3,051,394	47.38	13,803,597	50.94
$0 < f(h) \leq 5$	581,945	32.43	2,232,749	34.67	9,510,047	35.10
$5 < f(h) \leq 10$	211,224	11.77	612,621	9.51	2,153,291	7.95
$10 < f(h) \leq 15$	128,178	7.14	299,145	4.64	877,292	3.24
$15 < f(h) \leq 20$	72,312	4.03	141,729	2.20	433,867	1.60
$20 < f(h) \leq 25$	41,194	2.30	72,274	1.12	199,663	0.74
$25 < f(h) \leq 30$	16,870	0.94	21,644	0.34	81,683	0.30
$30 < f(h) \leq 35$	5,667	0.32	7,553	0.12	30,532	0.11
$35 < f(h) \leq 40$	992	0.06	1,649	0.03	5,632	0.02
$40 < f(h) \leq 45$	22	> 0.00	12	> 0.00	145	> 0.00

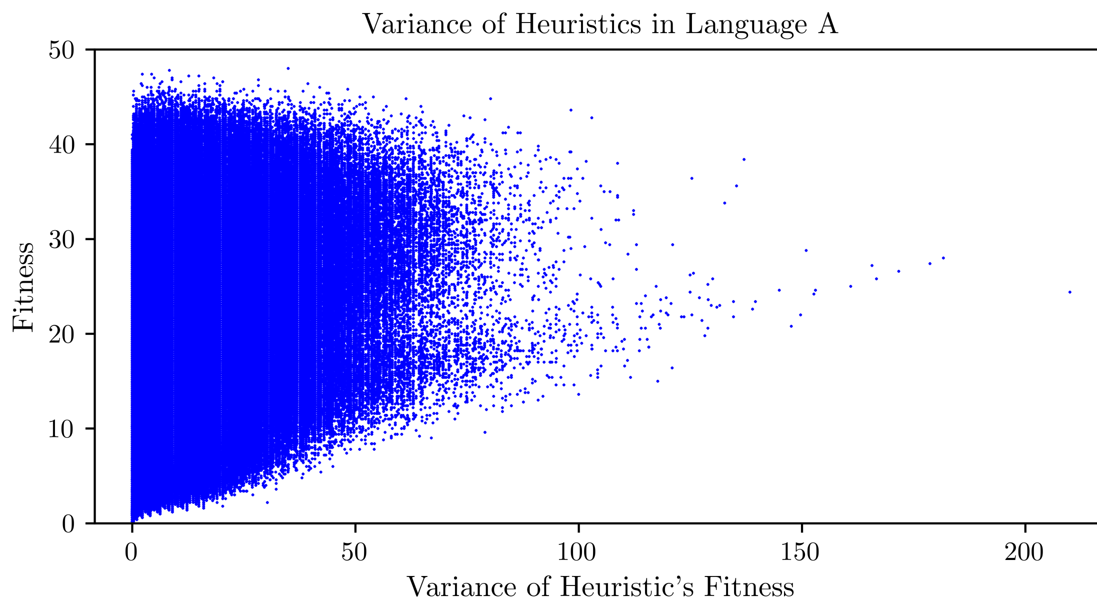
which performed well according to our fitness function were unreliable at consistently finding solutions. There are several unreliable heuristics, however there appears to be no trend between an unreliable heuristic and a high scoring heuristic. These results suggest to us that the five repetitions of the fitness function were perhaps too many; a smaller number could have provided us with a fitness value that is just as reliable.

One additional point of interest concerning these results is that the heuristics that can only be described using Language A1 appear to be more reliable than those from Language A. That is to say, it appears that the `WFF` terminal can be used to create more reliable heuristics. Yet, we also know that the `WFF` terminal cannot create heuristics that are as effective as those created using a randomly chosen broken clause (at least on the search space of heuristics that we have considered). We know from the literature presented in Section 2.3 that modern-day heuristics generally do not use strategies that consider all variables in a SAT problem. While the effectiveness of the heuristics created using the `WFF` terminal suggests that this is a sensible decision, these results also suggest that strategies using the `WFF` terminal can have a beneficial effect on the reliability of a heuristic.

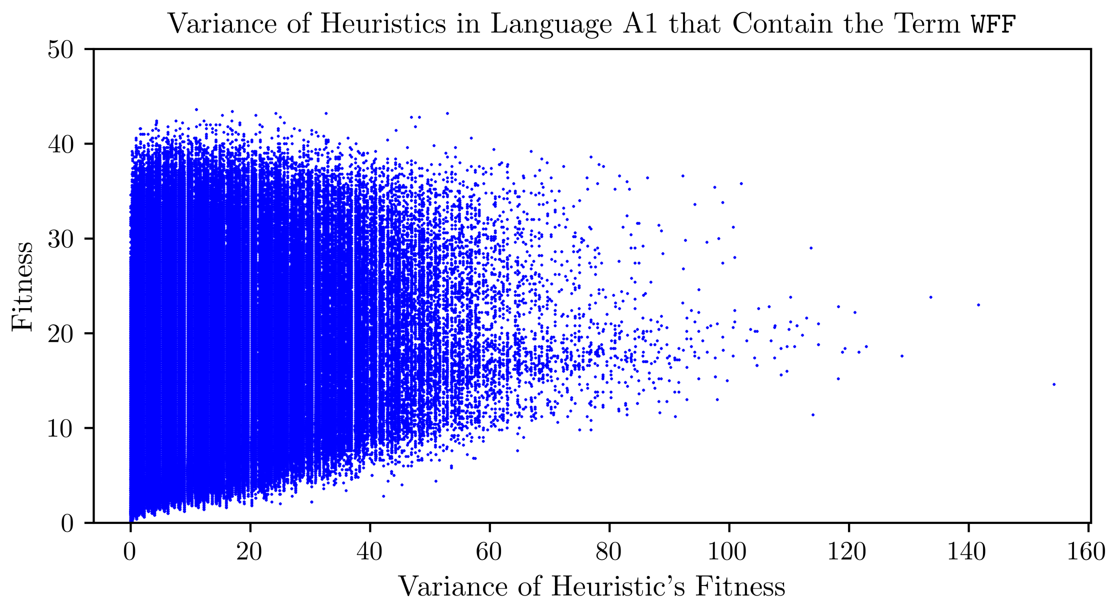
4.2.5 Timing Results

In Section 2.3 we noted how several researchers had succeeded in creating more efficient heuristics by decreasing the computational overhead of the heuristic function itself. While the heuristics evaluated in our software will never be as efficient as a hand-written variant (as we discussed in Section 3.3), we have taken great care to ensure that they are efficient in terms of how they update their required auxiliary data structures. Our fitness function measures how many flips a heuristic performs, paying no attention to the total time taken to solve a problem instance. When performing the exhaustive enumeration experiments, we collected additional data about the average time it took each heuristic to perform a flip in the overarching local search algorithm. In this subsection we analyse that data.

By plotting a heuristic's nanosecond-per-flip data against its reported fitness, we can determine which heuristics are not just effective, but fast. Fast and effective heuristics, we believe, are more preferable to slow and effective heuristics as, in a real-world setting, the faster heuristic could evaluate more assignments in a SAT problem, and potentially find a satisfying solution more quickly. We plotted this information for the enumerated heuristics with a fitness value > 10 in Language A



(a) All heuristics in Language A, showing the fitness of each heuristic plotted against that heuristic's fitness variance.



(b) All heuristics in Language A1 that are not in Language A, showing the fitness of each heuristic plotted against that heuristic's fitness variance.

Figure 4.10: All heuristics in Languages A and A1, showing the fitness of each heuristic plotted against that heuristic's fitness variance.

(total 4,926,816 heuristics) and for those in Language A1 not in Language A (total 2,504,809 heuristics) in Figure 4.11.

In both graphs there appears to be no correlation, general grouping or clustering of data points. However, we can identify a set of heuristics that appear to be distinctly separate from all other heuristics in Figure 4.11a, at around the 750 nanosecond-per-flip value. We were able to ascertain that many of the heuristics in this area used a smaller number of gain type metrics in their construction when compared to the heuristics in other areas. In future work we believe it may be beneficial to take this data into consideration when designing a fitness function, as it may help in identifying effective heuristics.

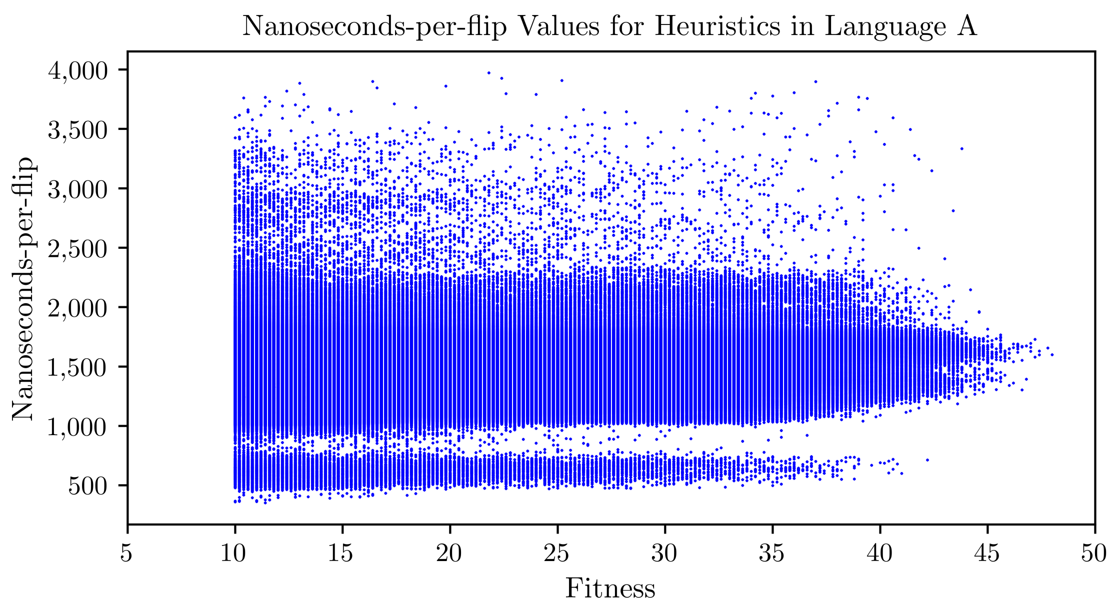
By comparing the two graphs, we can see that generally the heuristics that contain the `WFF` terminal take longer to complete an iteration of local search. We believe this is due to many of these heuristics needing to maintain a partial ordering of all variables according to a gain type, which adds considerable computational overhead to the heuristic function.

4.2.6 Individual Results

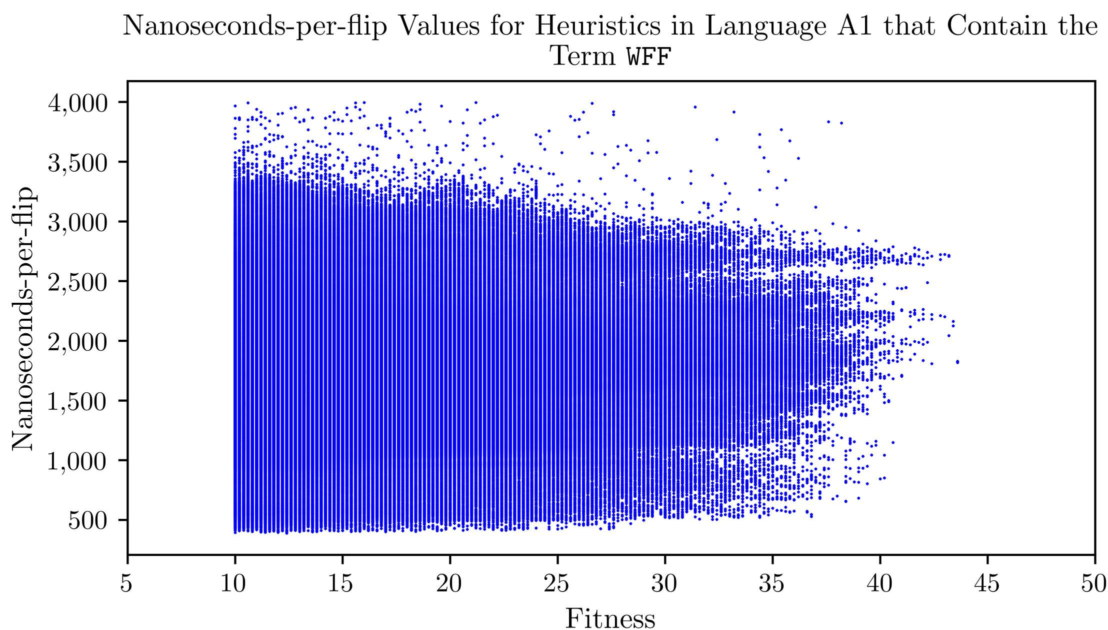
In this subsection we present examples of heuristics created from the exhaustive enumeration experiments which reported a high fitness. We also run these heuristics on the testing set, to ascertain how effective they are at solving different sized problem instances to the ones they were trained on.

We chose six heuristics from the enumeration of Language A and six from the enumeration of Language A1. We specifically chose heuristics from the enumeration of Language A1 that cannot be represented by Language A. The heuristics are shown in Figures 4.12 and 4.13.

There are several similarities that can be seen in the chosen heuristics. Firstly, nearly all the presented heuristics use the `GetOldestVar` function in their construction (the exception being SS-A1-3 in Figure 4.13c). This specific term was first described in work by Fukunaga [60, 63, 61]. From these results, it appears to be an effective component in the creation of LS-SAT heuristics. Several of the heuristics that used `GetOldestVar` extensively (SS-A-1, SS-A-4, SS-A1-1 and SS-A1-6 in Figures 4.12a, 4.12d, 4.13a and 4.13f respectively) are similar in their general construction to one of the heuristics automatically created in the systems designed by Fukunaga. We showed that heuristic, Depth-2-2, in Figure 3.2h. As this pattern has been seen



(a) All heuristics in Language A with fitness > 10, showing the fitness of each heuristic plotted against that heuristic's nanoseconds-per-flip value.



(b) All heuristics in Language A1 not in Language A with fitness > 10, showing the fitness of each heuristic plotted against that heuristic's nanoseconds-per-flip value.

Figure 4.11: All heuristics in Languages A and A1 with fitness > 10, showing the fitness of each heuristic plotted against that heuristic's nanoseconds-per-flip value.

```
GetOldestVar
 GetBestVar RBC-0 PosGain
 GetOldestVar
  GetBestVar RBC-0 NegGain
  GetBestVar RBC-0 NetGain
```

(a) Heuristic SS-A-1. Fitness value of 42.2.

```
GetOldestVar
 GetBestVar RBC-0 NetGain
 IfTabu 10
  GetBestVar RBC-0 NegGain
  IfNotMinAge RBC-0
   GetBestVar RBC-0 PosGain
   GetBestVar RBC-0 NegGain
```

(b) Heuristic SS-A-2. Fitness value of 48.0.

```
IfNotMinAge RBC-0
 IfVarCompare < NegGain
  GetBestVar RBC-0 NegGain
  GetBestVarSnd RBC-0 PosGain
 GetBestVar RBC-0 PosGain
```

(c) Heuristic SS-A-3. Fitness value of 41.0.

```
GetOldestVar
 GetOldestVar
  GetBestVar RBC-0 NegGain
  GetBestVar RBC-0 NegGain
 IfNotMinAge RBC-0
  GetBestVar RBC-0 NetGain
  GetBestVar RBC-0 PosGain
```

(d) Heuristic SS-A-4. Fitness value of 45.8.

```
IfTabu 50
 GetBestVar RBC-0 NetGain
 GetOldestVar
  IfNotMinAge RBC-0
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 PosGain
  GetBestVar RBC-0 NegGain
```

(e) Heuristic SS-A-5. Fitness value of 47.8.

```
IfTabu 20
 GetBestVar RBC-0 NetGain
 GetOldestVar
  GetBestVar RBC-0 NegGain
  IfTabu 40
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 PosGain
```

(f) Heuristic SS-A-6. Fitness value of 47.4.

Figure 4.12: Six heuristics that reported a high fitness value from the enumeration of Language A.

```
GetOldestVar
 GetOldestVar
  GetBestVar RBC-0 NegGain
  GetBestVar WFF PosGain
 GetBestVar WFF NetGain
```

(a) Heuristic SS-A1-1. Fitness value of 40.6.

```
GetOldestVar
 IfNotMinAge RBC-0
  GetBestVar RBC-0 NegGain
  GetBestVar RBC-0 PosGain
 GetBestVar WFF PosGain
```

(b) Heuristic SS-A1-2. Fitness value of 43.0.

```
IfTabu 20
 GetBestVar RBC-0 NegGain
 IfTabu 10
  GetBestVar RBC-0 PosGain
  GetBestVar WFF NetGain
```

(c) Heuristic SS-A1-3. Fitness value of 44.0.

```
GetOldestVar
 GetBestVar RBC-0 PosGain
 IfVarCond <= NegGain 0
  GetBestVarSnd WFF PosGain
  GetBestVar RBC-0 NegGain
```

(d) Heuristic SS-A1-4. Fitness value of 45.0.

```
GetOldestVar
 GetBestVar RBC-0 PosGain
 IfVarCond = PosGain 1
  GetBestVar WFF NegGain
  GetBestVar RBC-0 NegGain
```

(e) Heuristic SS-A1-5. Fitness value of 40.2.

```
GetOldestVar
 GetOldestVar
  GetBestVar WFF NetGain
  GetBestVar WFF PosGain
 GetOldestVar
  GetBestVar RBC-0 NegGain
  GetBestVar RBC-0 NetGain
```

(f) Heuristic SS-A1-6. Fitness value of 43.2.

Figure 4.13: Six heuristics that reported a high fitness value from the enumeration of Language A1.

several times in both our work and that by Fukunaga, this suggests to us that this mechanism could prove to be useful in the design of hand-crafted heuristics. Perhaps it could be used to augment previously existing heuristics, or to design entirely new ones.

We found it quite surprising that many of the heuristics which reported a high fitness value used the `IfTabu` term, as we did not believe that it would prove to be particularly effective. There are few examples of modern LS-SAT heuristics which use tabu mechanisms in their construction, instead relying on alternate techniques to prohibit the choosing of variables which have recently been flipped.

Few of the heuristics we highlighted used any of the functions other than `GetOldestVar`, `IfTabu`, `IfNotMinAge` and `GetBestVar`. There are some exceptions to this such as SS-A-3, SS-A1-4 and SS-A1-5 (shown in Figures 4.12c, 4.13d and 4.13e respectively). We found it quite surprising that none of these heuristics used the functions `PickRandomVar` or `IfRandLt`. The mechanisms which these functions represent have been widely used in the creation of hand-crafted LS-SAT heuristics which can be represented by Languages A and A1. For example, WalkSAT and Novelty use `IfRandLt`, while WalkSAT uses `PickRandomVar`.

In Table 4.6 we show the results from running the chosen heuristics on the testing set presented in Table 3.8.

Let us first consider the results in Table 4.6a. We can see that all six heuristics performed well on the initial five subsets of problem instances, with there being no clear best performing heuristic. We can state that every heuristic was able to solve at least 95% of the problems in these subsets. On the subsets containing larger problem instances, we can see that SS-A-1, SS-A-3 and SS-A-4 had good performance, with SS-A-4 performing the best and able to solve at least 50% of the SAT problems it was ran on. The terms used in the construction of the heuristics that performed well on the larger problem instances is noteworthy, as none of them used the `IfTabu` term discussed previously. As the language contains terms which correspond to relatively low Age values, perhaps these heuristics would perform better on larger problem instances if they had used terms which correspond to larger Age values. The best performing heuristic on all problem instances, SS-A-4, used a combination of several `GetOldestVar` functions, together with picking the best variable from a randomly chosen broken clause according to several gain type metrics.

When considering the results in Table 4.6b, we can see that the six heuristics from

Table 4.6: Results from running the heuristics in Figures 4.12 and 4.13 on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subset of problem instances.

(a) Results from running the heuristics in Figure 4.12 on the testing set.

	Heuristic					
Subset Name	SS-A-1	SS-A-2	SS-A-3	SS-A-4	SS-A-5	SS-A-6
uf50	99.6 0.0006	99.7 0.0005	99.7 0.0005	99.7 0.0006	99.9 0.0006	**99.9** **0.0005**
uf100	99.0 0.0038	98.8 0.0028	99.2 0.0032	99.2 0.002	**100.0** **0.0022**	99.0 0.0031
uf150	99.0 0.0122	**100.0** **0.0088**	100.0 0.0096	**100.0** **0.0088**	99.0 0.009	99.4 0.0091
uf200	96.8 0.0257	96.8 0.0215	95.0 0.0215	97.0 0.0461	**98.6** **0.025**	95.8 0.0311
uf250	99.0 0.0376	98.2 0.0381	97.6 0.033	98.6 0.0298	**99.6** **0.0474**	96.6 0.0406
ufv4000	70.0 9.4255	10.0 4.2608	72.0 10.3856	**72.0** **9.4374**	48.0 12.3256	20.0 7.3924
ufv7000	98.0 23.1846	0.0 0.0	**98.0** **22.1826**	90.0 19.0177	10.0 7.4578	0.0 0.0
ufv10000	78.0 15.9632	0.0 0.0	**88.0** **27.5494**	76.0 16.1754	0.0 0.0	0.0 0.0
ufv13000	44.0 15.8781	0.0 0.0	48.0 21.9421	**50.0** **15.4293**	0.0 0.0	0.0 0.0
ufv16000	32.0 23.7888	0.0 0.0	50.0 33.5542	**50.0** **26.0793**	0.0 0.0	0.0 0.0
ufv16000	42.0 20.3257	0.0 0.0	38.0 25.0816	**54.0** **29.4342**	0.0 0.0	0.0 0.0

Table 4.6: Results from running the heuristics in Figures 4.12 and 4.13 on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subset of problem instances. (Continued)

(b) Results from running the heuristics in Figure 4.13 on the testing set.

	Heuristic					
Subset Name	SS-A1-1	SS-A1-2	SS-A1-3	SS-A1-4	SS-A1-5	SS-A1-6
`uf50`	99.9 0.0009	99.8 0.0006	99.9 0.0009	99.8 0.0008	99.8 0.0007	**100.0** **0.0009**
`uf100`	**100.0** **0.0054**	99.8 0.003	100.0 0.0051	99.2 0.004	99.8 0.0044	100.0 0.0067
`uf150`	100.0 0.0289	99.8 0.017	100.0 0.0184	100.0 0.0224	99.8 0.013	**100.0** **0.018**
`uf200`	99.6 0.1233	97.0 0.0296	99.8 0.1496	95.2 0.0432	96.0 0.0292	**100.0** **0.0971**
`uf250`	99.6 0.1125	98.2 0.0579	99.8 0.066	99.2 0.0594	97.2 0.0612	**100.0** **0.1062**
`ufv4000`	2.0 1.2015	10.0 9.69	0.0 0.0	**80.0** **15.2965**	72.0 22.6692	18.0 6.5506
`ufv7000`	0.0 0.0	0.0 0.0	0.0 0.0	**92.0** **31.0538**	90.0 36.927	0.0 0.0
`ufv10000`	0.0 0.0	0.0 0.0	0.0 0.0	**58.0** **26.7388**	30.0 13.6374	0.0 0.0
`ufv13000`	0.0 0.0	0.0 0.0	0.0 0.0	**20.0** **12.1084**	0.0 0.0	0.0 0.0
`ufv16000`	0.0 0.0	0.0 0.0	0.0 0.0	**10.0** **7.9136**	0.0 0.0	0.0 0.0
`ufv16000`	0.0 0.0	0.0 0.0	0.0 0.0	**10.0** **5.6826**	0.0 0.0	0.0 0.0

the enumeration of Language A1 also performed well on the initial five subsets of problem instances. In comparison to the results in Table 4.6a, these heuristics were more consistent, as a larger number of the heuristics were able to solve all instances in these five subsets. SS-A1-6 performed the best on these, solving 100% of the instances. However, the timing data shows us that the heuristics from Language A1 took longer to solve these problem instances. We believe this can be attributed to the use of subtrees in the form `GetBestVar` {`WFF`, g}. To calculate the variable returned from a subtree such as this, a partial ordering of all variables according to the gain type metric g is required. This is computationally expensive to maintain for problem instances with a larger number of variables.

When we look at the results from running these heuristics on the larger problem instances, we can see that most of the heuristics were unable to solve many problems. Only two heuristics, SS-A1-3 and SS-A-4, were able to solve a notable number of these instances. However, their performance was notably worse on the largest problem instances when compared to those heuristics shown in Table 4.6a.

Finally, we compare these results to the results of evaluating hand-crafted heuristics on the testing set, the results of which were shown in Table 3.9.

On the subsets containing smaller problem instances, the performance of the automatically created heuristics and the best performing of the hand-crafted heuristics is generally similar. The automatically created heuristics from Language A have comparable performance to gNovelty+, however they are less consistent - that is to say, they do not solve as many instances. The automatically created heuristics from Language A1 perform better than gNovelty+ on these subsets - for example SS-A1-6 is more consistent than gNovelty+. We can see that the automatically created heuristics perform better than the other hand-crafted heuristics on the subsets containing smaller problem instances.

In Table 3.9 we saw that ProbSAT had the best performance of the hand-crafted heuristics on the subsets containing larger problem instances. Comparing the performance of the automatically created heuristics to ProbSAT, we can see that several of them appear to offer better performance on these subsets. Specifically the heuristics SS-A-1, SS-A-3, SS-A-4 and SS-A1-4.

Table 4.7: The parameters used in the GP experiments in Section 4.3.

Parameter	Value
Population Size	1,000
Generations	100
Initialisation Method	Grow (Max Depth = 7)
Selection Method	Tournament
Crossover	80%
Mutation	10%
Reproduction	5%
Elite Programs	5%

4.3 Genetic Programming Experiments

In this section we show the methodology used and results obtained from our GP experiments performed using Language A and A1. We also show results detailing how some of the heuristics created from GP perform on the testing set of problem instances.

4.3.1 Methodology

We used a software suite called EpochX [137] to build the GP software that is the focus of the experiments detailed in this section. It is a general-purpose software library that aids in the construction of GP systems. The parameters for our GP experiments are shown in Table 4.7. We performed 5 repetitions of the GP experiment for both Languages A and A1.

In total each repetition of the GP algorithm evaluated $101,000$ heuristics. This additional $1,000$ is attributed to the initial population of heuristics. The terminal set and function set used are highlighted in Table 4.1. The reader may note that we use the grow method of initialising our population. This was done for a very specific reason. Our language contains several functions that are not "balanced" - that is to say, a function such as `IfVarCond` requires several arguments. Some of these arguments can only be instantiated with a terminal, for example, `Comparator`. Other arguments to `IfVarCond` can only be other functions. In preliminary experiments, populations created through the full method did not utilise these "unbalanced" functions in their created program trees. In turn, when using this method (or the ramped half-and-half

method), many heuristics created used functions from a small subset of the language, and the overall populations were not particularly diverse. The grow method provided us with more varied heuristics to initialise our population, and we made the decision to use this initialisation technique.

The GP experiments were performed on the same computer described in Section 4.2.2. Each repetition of each experiment took around 4 hours to run, for a total of 40 hours for all experiments.

4.3.2 Results

In this subsection we present the results from the GP experiments performed using Languages A and A1. We provide general data about each of the 5 repetitions performed using each language, and give more detailed data about the best of these repetitions. By "best" we mean the repetition that reported the fittest overall heuristic. For Language A this was repetition 5, and for Language A1 this was repetition 3. In Table 4.8 we show some general data about the population at various points in each GP repetition. In Figures 4.14 and 4.15 we show detailed information regarding the general fitness distribution and size of the heuristics created from each of the best repetitions from Languages A and A1.

We can see from the results in Table 4.8 that the fittest heuristic reported from each repetition had a higher fitness than the fittest heuristic found through exhaustive enumeration. Compared to the tens of millions of heuristics evaluated in those experiments, these results were obtained from only $101,000$ heuristic evaluations. Indeed, by generation 50 in nearly all repetitions, the heuristics generated were of a higher quality than those produced by exhaustive enumeration.

If we look at the results in Figure 4.14 we can see how repetition 5 for Language A and repetition 3 for Language A1 progressed. At the beginning of the algorithm, the mean fitness of the population grew quickly yet, as time went on, the gains made in the overall mean fitness slowed. It is clear that in the final generation, there are many high-quality heuristics. The general way in which these algorithms progressed is consistent with other work in GP.

The results in Figure 4.15 show us information pertaining to the size of the heuristics created in each generation for the repetitions highlighted previously. We can see that the heuristics were of a very large size, and grew as the algorithm progressed. In the GP community, this phenomenon is known as bloat [172]. We

Table 4.8: Statistical data pertaining to the GP experiments performed using Languages A and A1. For each repetition, we show the best heuristic's fitness, the best heuristic's size, the mean fitness of the population, and the mean size of the population for specific generations. These are the initial generation, the 25th, 50th, 75th and the 100th generation.

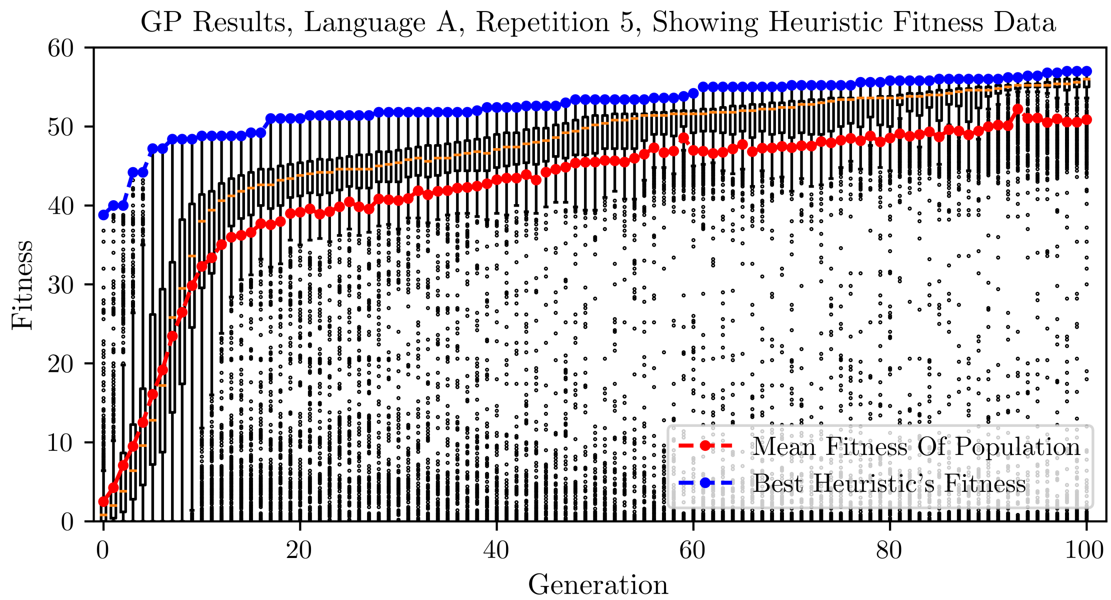
(a) Statistical data pertaining to the GP experiments performed using Language A.

		Repetition				
		1	2	3	4	5
Initial Gen	Best Heuristic's Fitness	37.6	37.2	38.8	34.6	38.8
	Best Heuristic's Size	107	18	79	69	25
	Mean Fitness	2.32	2.57	2.33	2.26	2.49
	Mean Size	45.15	47.60	50.84	50.10	47.88
25th Gen	Best Heuristic's Fitness	50.6	48.2	49.6	52.4	51.4
	Best Heuristic's Size	149	116	316	206	304
	Mean Fitness	39.34	36.94	38.1	39.53	40.46
	Mean Size	120.86	100.42	109.88	107.32	85.86
50th Gen	Best Heuristic's Fitness	52.6	50.8	52.2	53.8	53.4
	Best Heuristic's Size	612	243	226	477	331
	Mean Fitness	42.97	41.02	43.01	43.57	45.49
	Mean Size	228.57	156.06	189.99	213.69	190.35
75th Gen	Best Heuristic's Fitness	54.4	53.0	54.2	54.4	55.2
	Best Heuristic's Size	1,008	617	498	573	302
	Mean Fitness	46.37	43.28	46.2	46.99	48.26
	Mean Size	442.92	226.86	328.01	347.24	379.07
100th Gen	Best Heuristic's Fitness	55.4	54.8	56.0	55.2	57.0
	Best Heuristic's Size	1,059	632	1,152	486	1,004
	Mean Fitness	49.5	48.55	48.65	49.05	50.86
	Mean Size	612.26	487.96	558.02	480.68	639.71

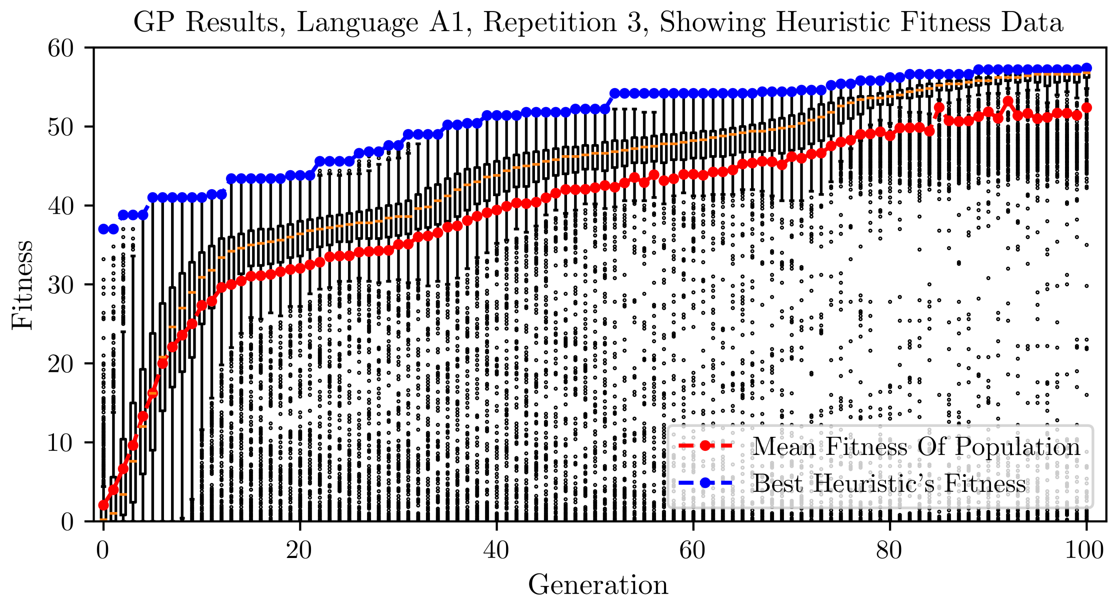
Table 4.8: Statistical data pertaining to the GP experiments performed using Languages A and A1. For each repetition, we show the best heuristic's fitness, the best heuristic's size, the mean fitness of the population, and the mean size of the population for specific generations. These are the initial generation, the 25th, 50th, 75th and the 100th generation. (Continued)

(b) Statistical data pertaining to the GP experiments performed using Language A1.

		Repetition				
		1	2	3	4	5
Initial Gen	Best Heuristic's Fitness	35.8	30.6	37.0	27.4	33.2
	Best Heuristic's Size	102	18	48	27	70
	Mean Fitness	2.17	2.24	2.05	1.9	2.0
	Mean Size	46.73	49.12	51.37	48.94	48.01
25th Gen	Best Heuristic's Fitness	48.0	46.6	45.6	47.2	49.2
	Best Heuristic's Size	77	78	114	38	56
	Mean Fitness	34.97	37.06	33.61	35.62	37.15
	Mean Size	107.31	95.82	127.46	77.86	92.19
50th Gen	Best Heuristic's Fitness	51.2	49.4	52.2	50.8	53.6
	Best Heuristic's Size	194	214	212	252	106
	Mean Fitness	41.18	43.11	42.25	41.48	43.35
	Mean Size	143.59	137.83	171.56	168.44	140.89
75th Gen	Best Heuristic's Fitness	54.2	50.6	55.4	51.6	54.0
	Best Heuristic's Size	319	201	1,121	512	589
	Mean Fitness	44.26	45.01	48.03	45.43	47.6
	Mean Size	261.26	206.08	342.76	276.42	280.28
100th Gen	Best Heuristic's Fitness	55.8	51.8	57.4	52.6	55.8
	Best Heuristic's Size	807	168	1,122	806	903
	Mean Fitness	48.19	45.42	52.39	47.38	48.97
	Mean Size	484.34	233.42	603.97	380.28	411.08

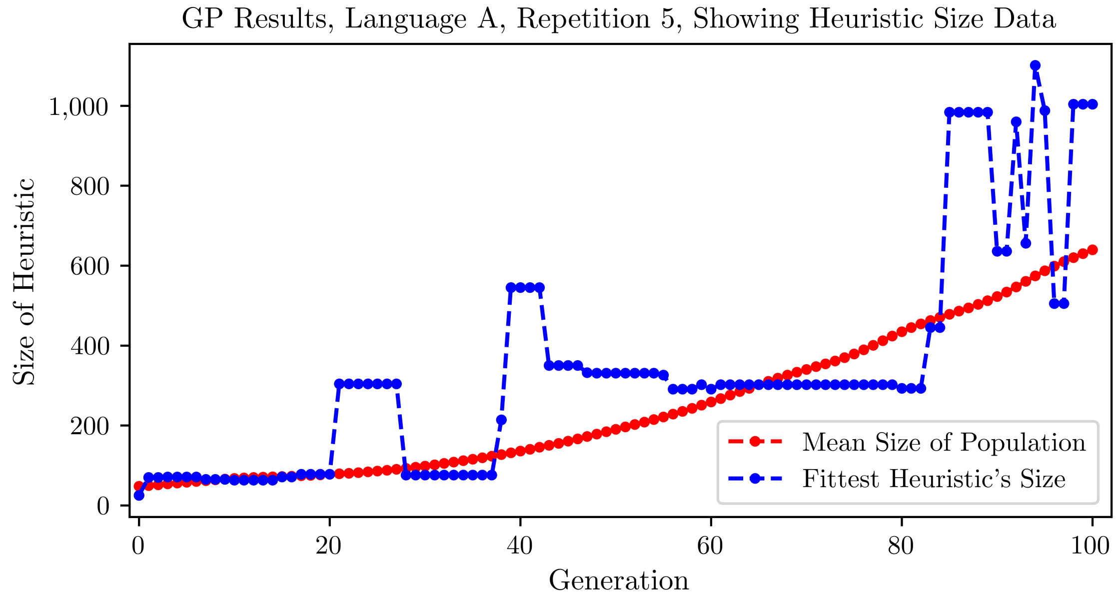


(a) Graph showing the fitness data from the 5th GP repetition performed using Language A.

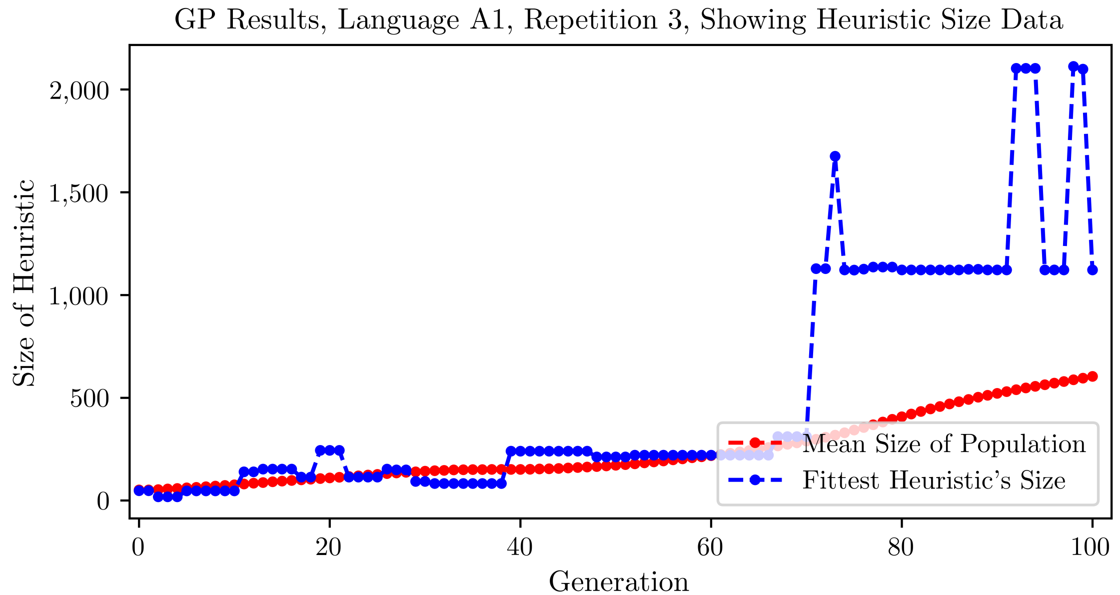


(b) Graph showing the fitness data from the 3rd GP repetition performed using Language A1.

Figure 4.14: Fitness data from the best repetitions of the GP experiments performed using Languages A and A1. For each generation in each repetition, we show a boxplot detailing the distribution of fitness values in that generation, with outliers shown. We also show the mean fitness and the best heuristic's fitness.



(a) Graph showing the size data from the 5th GP repetition performed using Language A.



(b) Graph showing the size data from the 3rd GP repetition performed using Language A1.

Figure 4.15: Size data from the best repetitions of the GP experiments performed using Languages A and A1. For each generation in each repetition, we show the mean size of the population's heuristics, and the size of the fittest heuristic.

Table 4.9: Statistical data concerning the frequency of terms used in the fittest heuristic returned from each repetition of the GP experiments performed using Languages A and A1. We only show terms that are functions, variable sets or gain types.

Term	GP-A-1	GP-A-2	GP-A-3	GP-A-4	GP-A-5	GP-A1-1	GP-A1-2	GP-A1-3	GP-A1-4	GP-A1-5
PickRandomVar	51	38	46	16	71	36	4	54	43	40
GetBestVar	86	37	111	54	72	75	17	95	70	89
GetBestVarSnd	71	51	69	24	61	44	10	73	43	54
GetOldestVar	34	18	23	12	47	30	2	48	24	40
IfNotMinAge	51	28	57	20	25	11	5	18	20	51
IfTabu	23	14	29	21	37	31	3	33	29	31
IfRandLt	29	16	54	14	26	17	6	54	21	17
IfVarCond	36	11	31	20	36	35	7	48	34	27
IfVarCompare	34	38	31	6	32	30	7	20	27	16
RBC-0	259	154	283	114	229	91	20	161	93	162
WFF	-	-	-	-	-	75	16	79	83	72
PosGain	66	39	42	30	70	51	10	64	54	55
NegGain	83	54	99	44	56	53	20	74	58	61
NetGain	78	44	101	30	75	80	11	98	62	70

can see from the results in Table 4.8 that each repetition in both GP experiments appeared to suffer from it. We can say that generally, across all repetitions, the best heuristic's size grew as the algorithm progressed, though not as uniformly as the mean, and it decreased on several occasions.

In Table 4.9 we show the number of times each term is used in the best heuristic reported from each repetition. These heuristics are named, with each name derived from the repetition of the experiment it originated from. We collected this data to determine if there were any terms that were being used more than others. However, it appears that there is no discernible pattern from these results, and no heavily favoured term.

4.3.3 Individual Results

In this subsection we look at how the fittest heuristics created from each GP repetition perform on the testing set described in Section 3.4.3.

We do not present any of these heuristics in this subsection, as they are very large. However, we do present the fittest heuristic created from each of the GP experiments performed using Language A and A1 in Appendix B. These heuristics are GP-A-5 and GP-A1-3. GP-A-5 reported a fitness of 57.0 and contains $1,004$ terms, while GP-A1-3 reported a fitness of 57.4 and contains $1,122$ terms.

In Table 4.10 we show the results obtained from evaluating the best heuristics created from each repetition on the testing set. Like those heuristics considered in Section 4.2.6, the heuristics based on Language A1 generally performed better on the smaller problem instances (the first five subsets) than those from Language A. However, they took longer to solve these instances than those created from Language A. On the six subsets containing larger problem instances, none of the heuristics performed well, with few being able to solve many of those problem instances. Generally we would state that both sets of heuristics performed similarly to each other. Of all heuristics tested, GP-A-4 and GP-A1-2 had the best performance.

By comparing these results to those shown in Section 4.2.6, we can see that the heuristics created using GP perform slightly better on the initial five subsets of problem instances in regards to the number of problems solved, but slightly worse when we look at the time taken to solve them. We had assumed that this would be the case, as larger heuristics are generally more computationally expensive to evaluate, which in turn would mean that they would take longer to complete a single iteration of local search, and potentially take longer to find satisfying solutions.

On the larger problem instances, those heuristics created from exhaustive enumeration outperformed those from GP. From exhaustive enumeration we found several examples of heuristics that performed well on the larger problem instances. From our GP experiments, we only found two heuristics that were able to solve any notable proportion of those problem instances. Both of these heuristics performed much worse than the best performing heuristics from exhaustive enumeration.

Table 4.10: Results from running the heuristics created from GP using Languages A and A1 on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subsets.

(a) Results from running the heuristics created from GP and Language A on the testing set.

	Heuristic				
Subset Name	GP-A-1	GP-A-2	GP-A-3	GP-A-4	GP-A-5
uf50	99.9 0.0011	99.9 0.0011	**100.0** **0.0008**	99.9 0.0009	99.7 0.0009
uf100	98.8 0.0049	99.0 0.0051	**100.0** **0.0036**	99.2 0.0041	99.6 0.0041
uf150	100.0 0.0228	99.4 0.031	**100.0** **0.0124**	100.0 0.022	100.0 0.0189
uf200	97.2 0.0625	96.4 0.0758	**100.0** **0.1339**	97.6 0.0401	97.2 0.0384
uf250	98.6 0.1006	98.8 0.1118	**99.0** **0.062**	97.4 0.0597	97.6 0.0576
ufv4000	12.0 1.9523	32.0 4.1525	34.0 13.092	**50.0** **10.1234**	30.0 16.8937
ufv7000	0.0 0.0	18.0 7.0373	14.0 4.5718	**30.0** **7.9705**	0.0 0.0
ufv10000	0.0 0.0	0.0 0.0	10.0 6.233	**28.0** **17.5356**	0.0 0.0
ufv13000	0.0 0.0	0.0 0.0	0.0 0.0	**2.0** **1.6226**	0.0 0.0
ufv16000	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
ufv16000	0.0 0.0	0.0 0.0	0.0 0.0	**10.0** **4.8211**	0.0 0.0

Table 4.11: Results from running the heuristics created from GP using Languages A and A1 on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subset. (Continued)

(a) Results from running the heuristics created from GP and Language A1 on the testing set.

	Heuristic				
Subset Name	GP-A1-1	GP-A1-2	GP-A1-3	GP-A1-4	GP-A1-5
`uf50`	100.0 0.002	**100.0** **0.001**	100.0 0.0012	100.0 0.0011	100.0 0.001
`uf100`	100.0 0.008	**100.0** **0.0046**	100.0 0.0057	100.0 0.0053	100.0 0.005
`uf150`	100.0 0.0326	**100.0** **0.0211**	100.0 0.0335	100.0 0.0265	100.0 0.0264
`uf200`	98.4 0.1764	98.8 0.1169	99.0 0.1604	99.0 0.1281	**99.0** **0.1103**
`uf250`	98.8 0.2157	99.2 0.076	**99.8** **0.1768**	99.0 0.0694	99.2 0.075
`ufv4000`	20.0 15.4998	62.0 18.5151	**66.0** **23.3641**	2.0 1.0264	40.0 14.7485
`ufv7000`	0.0 0.0	**80.0** **48.2407**	44.0 23.5909	0.0 0.0	10.0 2.6482
`ufv10000`	0.0 0.0	**30.0** **18.8682**	30.0 26.2578	0.0 0.0	0.0 0.0
`ufv13000`	0.0 0.0	**10.0** **8.4459**	0.0 0.0	0.0 0.0	0.0 0.0
`ufv16000`	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
`ufv16000`	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0

4.4 Discussions & Conclusions

In this chapter we have presented experiments that have automated the heuristic creation process through exhaustive enumeration and GP. We have shown that both of these methodologies can be used to create effective LS-SAT heuristics. We have provided analysis on the millions of heuristics produced from exhaustive enumeration, as well as some analysis on the GP experiments conducted.

In our analysis on exhaustive enumeration, our primary finding was that there exist areas in the search space where effective heuristics are concentrated together. In our results, these areas were represented as peaks in the landscape of heuristics when presented in the order they are generated. For any two heuristics that are next to each other in this ordering, they are almost always nearly identical to each other, usually only deviating by a single term.

On the use of Language A1 in comparison to Language A, there was no discernible difference when comparing the highest quality of heuristic created from each experiment. However, when comparing the distribution of the quality of heuristics found through exhaustive enumeration, we could see that those heuristics exclusively in Language A1 reported a lower fitness value on average. There was some difference when comparing the variance of the fitness quality of both sets of heuristics, with those created from Language A1 appearing to be more consistent in regards to their reported fitness. When examining the timing data of both sets, on average the heuristics that were solely in Language A1 took longer to run.

We evaluated a set of heuristics taken from the exhaustive enumeration experiments on the testing set. We found these heuristics performed very well on problem instances of a similar size to those they were trained on. On larger instances, several of these heuristics performed well, and we would suggest that they are worthy of more focused research, as they have the potential to be used as general-purpose heuristics. The tested heuristics that could only be described using Language A1 performed slightly better on problems of similar size to those they were trained on, and heuristics based solely on Language A appeared to perform better on the larger problem instances. We believe the heuristics created from Language A1 reported worse performance on the larger problem instances as they frequently contained subtrees in the form `GetBestVar` $\{$`WFF`, $g\}$. Heuristics containing subtrees in this form would need to maintain a partial ordering of all variables in the problem according to the gain type metric g. For large problem instances, this would be computationally expensive to

do. In turn, these heuristics would perform less iterations of local search in the time allocated, and be unable to perform as well as they do on smaller problem instances.

Heuristics that use the `WFF` terminal are rarely used in modern-day heuristics, with researchers using other methods to represent large sets of variables, such as through configuration checking. The data concerning the large number of heuristics evaluated in this chapter may lend some credence to this choice, as the best heuristics in our data set that used the `WFF` terminal generally performed poorly on large problem instances when compared to those built from random walk components.

In the GP experiments performed, the best reported heuristic from each repetition had a higher fitness value than any of those created through exhaustive enumeration. However, the heuristics created were very large, and compared to those created from exhaustive enumeration, not easy to understand. When evaluating these heuristics on the testing set, we saw that the performance of the heuristics on smaller problem instances was comparable to those created from exhaustive enumeration, however few performed well when ran on the larger problem instances. Without further analysis of the heuristics, it would be difficult to determine why they did not perform well. However it may be the case that the created heuristics are overfitted, hence their performance is poor on the larger problem instances, which are unlike those they were trained on.

In general, we would state that our fitness function is not appropriate for identifying LS-SAT heuristics which are effective on problem instances larger than those they are trained on. Though we found some examples of heuristics that perform well on larger sized problem instances, we have also seen many examples that do not perform well. In both cases, the heuristics have reported a high fitness value. It may also be the case that the fitness function has reported a lower fitness value for a heuristic that is effective at solving larger problem instances. In the rest of this thesis, we do not experiment with alternative fitness functions. However, we believe that a potential future avenue of research could be in investigating the role the fitness function has on the quality of the heuristics created. Such research could help us to design a fitness function that can identify heuristics which are effective on problem instances larger than those they are trained on.

As to whether enumeration is a viable alternate methodology to GP, we would have to state that it would depend on the use-case. The fittest heuristics created from GP reported a higher fitness value than those created from exhaustive enumeration.

Yet, those created from exhaustive enumeration are much smaller and easier to understand. It would be much easier to use them in a hand-crafted LS-SAT solver, or as a bit-part in a larger heuristic. Therefore we believe that, for our use-case, exhaustive enumeration has some advantages over GP. The obvious disadvantage of using exhaustive enumeration is that it takes much longer to run than GP.

There are two core discrepancies with how exhaustive enumeration interacts with the languages we used. Firstly, there is the issue of program trees which have branches of terms that are redundant. An example of this was shown in Figure 4.8. This occurred due to an oversight regarding the range of certain gain type metrics. There are two ways to solve this issue; either create a more verbose language that prohibits certain integers being compared to certain gain types, or use a more verbose typing system to ensure that these type of programs cannot be formed. Both of these solutions, we feel, are a large undertaking for comparatively little reward.

Secondly, there are groups of heuristics that are, semantically, the same heuristic. For example, heuristics in the form `IfRandLt` $\{p, v_1, v_2\}$ are equivalent to `IfRandLt` $\{(1-p), v_2, v_1\}$. There are other examples of equivalent programs using other terms, and groups of terms. To create a generalised solution to alleviate this issue, we would require some way of reasoning about partial and complete heuristics. Like the previous issue, we deem the benefit to be small, though the issue of semantic equivalence in what are stochastic functions is an interesting topic in itself.

Though the exhaustive enumeration technique applied to heuristic creation was computationally expensive to perform, we feel that we have gained invaluable data regarding the structure of well-performing heuristics. We have also shown several examples of heuristics that perform well on large problem instances. While the heuristics created do not outperform those created from GP in regards to the fitness function, they are small and simple to understand. In the next chapter we use a subset of the data we have presented here to explore the relationship between heuristics using the MTED metric.

Chapter 5

Analysing Heuristics Using the Minimum Tree Edit Distance

5.1 Introduction

In the previous chapter we performed a series of experiments using program synthesis to create LS-SAT heuristics, the majority of which were conducted using exhaustive enumeration. In this chapter we perform an analysis on a subset of the heuristics created from those exhaustive enumeration experiments using the MTED metric, which we introduced in Section 2.6. Through this analysis, we propose a new program synthesis technique, and perform simulations evaluating its performance.

To be clear to the reader, all of the experiments contained within this chapter are performed using heuristics and fitness results taken from the exhaustive enumeration experiments described in the previous chapter.

The format of this chapter is as follows. In Section 5.2 we describe our initial observations about the search space of heuristics when analysed using the MTED metric, as well as defining the neighbourhood of a heuristic. In Section 5.3 we perform a more in-depth analysis on a heuristic's neighbourhood, and evaluate its potential as a method of moving through the search space. In Section 5.4 we present a set of experiments which simulate local search on heuristics. Finally in Section 5.5 we present our conclusions from the research presented in this chapter.

5.2 Initial Observations

In Figures 4.5 and 4.6 we presented all the heuristics in Language A of size 14 in the order that they were generated. In those figures the heuristics were coloured according to their leading n terms, for n values of 1, 2 and 3. We discussed then that some of the peaks in the graphs, where there are concentrations of heuristics with a high fitness, appeared to be separate from other groups of heuristics near them according to their leading n terms.

This measurement of closeness by generation ordering does not completely describe how effective heuristics can be grouped together by their structure. Figures 4.12 and 4.13 contained examples of effective heuristics from Languages A and A1. Though we only presented a small number of heuristics, some of them have similarities in both their structure and ordering of terms. For example, SS-A1-4 and SS-A1-5 (shown in Figures 4.13d and 4.13e respectively) have a similar "core structure"; they both have a root of `GetOldestVar`, and the second argument to the root is the `IfVarCond` function, but nearly all other terms used in each heuristic differ from each other.

In Section 2.6 we introduced the MTED metric, which is used to calculate the minimum number of edits required to transform one tree to another. In this chapter we use this distance metric to analyse the similarity between any two heuristic's underlying program tree representations. This allows us to compare arbitrary program trees of different sizes, irrespective of their generation ordering or even language. It is through this metric that we analyse the search space of heuristics.

To begin, let us consider the four heuristics in Figure 5.1. Each heuristic has been hand-picked with the criteria that the chosen heuristic has a distinct quality of fitness different from the others. We then compared each heuristic to every other heuristic in Language A of size ≤ 15 using the MTED metric. That is to say, the MTED was found between each candidate heuristic in Figure 5.1 and every other heuristic in the subset of heuristics considered. All the heuristics were then separated into sets based on their MTED from the candidate heuristic. For a candidate heuristic h and MTED n, we refer to the set containing all heuristics whose MTED to h is n as h's MTED(n) set. In Figure 5.2 we show the distribution of fitness values in all the MTED sets for the heuristics shown in Figure 5.1, and in Table 5.1 we show the size of those MTED sets.

We can see that the size of each MTED(n) set generally correlates with its n value. The distribution of the size of these sets is a little surprising; we had assumed that the

Table 5.1: The size of the MTED(n) sets for the heuristics shown in Figure 5.1.

	Candidate heuristic			
n	EX-1	EX-2	EX-3	EX-4
1	24	13	25	25
2	256	82	275	275
3	1,639	325	1,786	1,786
4	7,093	1,042	7,698	7,698
5	22,144	3,421	23,388	23,388
6	52,520	11,211	52,024	52,024
7	100,548	31,443	87,710	87,710
8	167,143	77,133	119,786	119,764
9	255,609	177,302	158,822	158,253
10	365,161	362,097	259,850	256,067
11	483,001	601,816	469,656	457,837
12	583,090	753,790	692,133	673,611
13	597,483	678,780	704,678	695,392
14	429,301	384,652	444,226	463,099
15	169,757	154,363	180,963	199,293
16	41,879	45,215	64,440	68,967
17	6,023	0	15,225	17,496
18	14	0	0	0

```
IfVarCond < NegGain 3
 IfRandLt 0.9
  GetBestVar RBC-0 NegGain
  GetBestVar RBC-0 PosGain
 GetBestVarSnd RBC-0 NetGain
```

(a) Heuristic EX-1. Fitness value of 0.

```
IfRandLt 0.9
 GetBestVar RBC-0 PosGain
 GetOldestVar
  GetBestVarSnd RBC-0 NetGain
  GetOldestVar
   GetBestVarSnd RBC-0 PosGain
   PickRandomVar RBC-0
```

(b) Heuristic EX-2. Fitness value of 7.2.

```
IfTabu 5
 GetBestVar RBC-0 NegGain
  IfVarCond = NegGain 0
   GetBestVar RBC-0 NegGain
   GetBestVarSnd RBC-0 PosGain
```

(c) Heuristic EX-3. Fitness value of 19.2.

```
IfTabu 30
 GetBestVar RBC-0 PosGain
  IfVarCond < PosGain 0
   GetBestVar RBC-0 PosGain
   GetBestVar RBC-0 NegGain
```

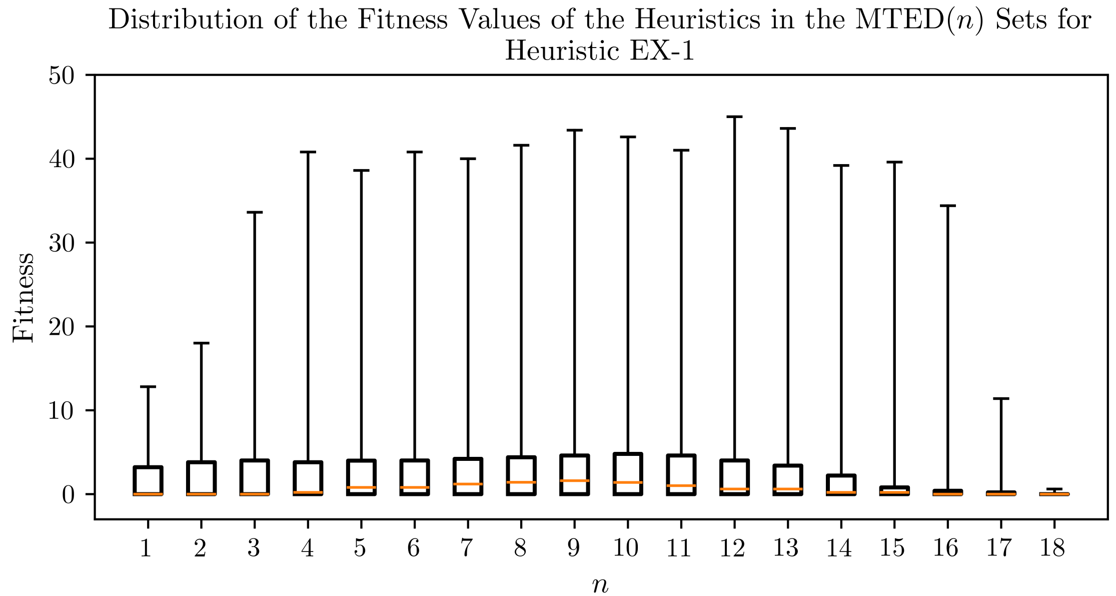
(d) Heuristic EX-4. Fitness value of 35.2.

Figure 5.1: Four heuristics from the enumeration of Language A.

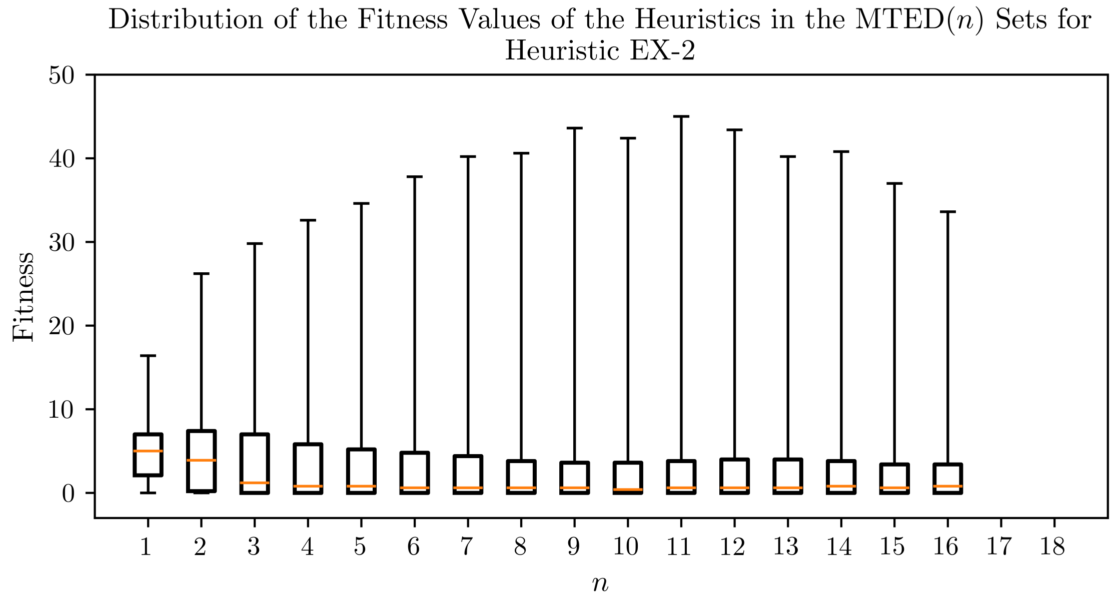
highest n values would correspond to the largest MTED(n) sets. Instead, the highest n values correspond to relatively small MTED(n) sets. We believe this occurred because these sets were constructed from a subset of the heuristics in Language A. We assume that if we were to consider the infinite set of heuristics in the language, then higher n values would always correspond to larger sized MTED(n) sets.

When we consider the distribution of fitness values in each set, generally we can say that the highest fitness value found in each MTED(n) set correlates with n, and that the median, lower quartile and upper quartile fitness values negatively correlate with n. We can also state that each set contains some heuristics with a high-quality fitness, but that the majority of the heuristics are of a low quality.

For each of the candidate heuristics considered we can see that, for $n \leq 4$, many of their MTED(n) sets contain heuristics with a higher fitness than the candidate. This is more pronounced for heuristics with a lower fitness, such as heuristic EX-1. Heuristic EX-1's MTED(1) set contains several heuristics that have a higher fitness than heuristic EX-1. Even the fittest candidate heuristic, heuristic EX-4, has some heuristics in its MTED(1) set which have a higher fitness than it. These sets are also relatively small. When we consider the four candidate heuristics shown in Figure 5.1,

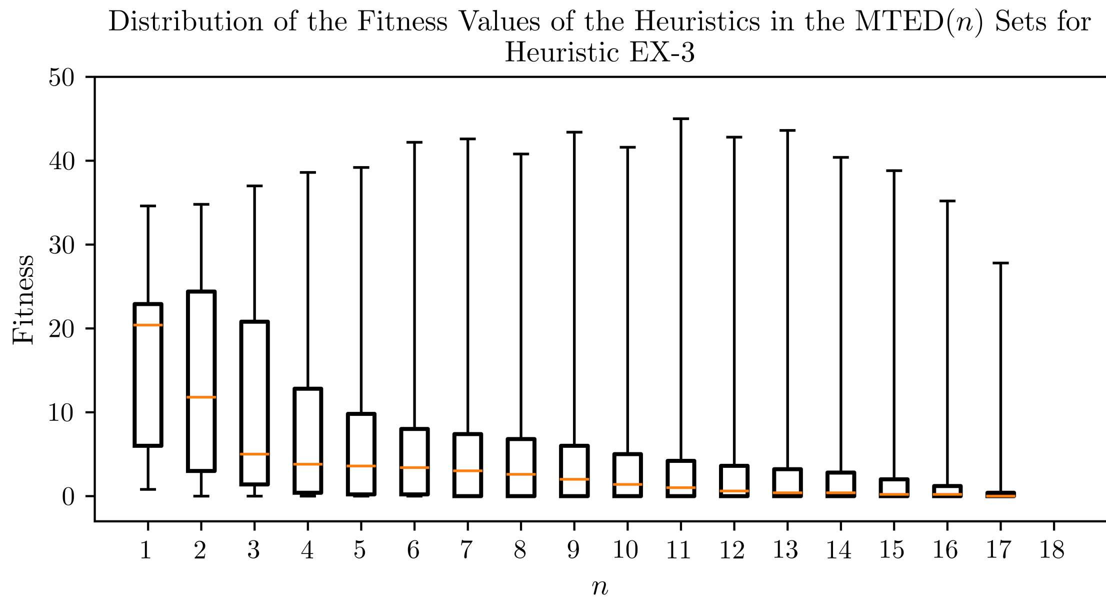


(a) Distribution of the fitness values of the heuristics in EX-1's MTED(n) sets. Heuristic EX-1 is shown in Figure 5.1a and has a fitness of 0.

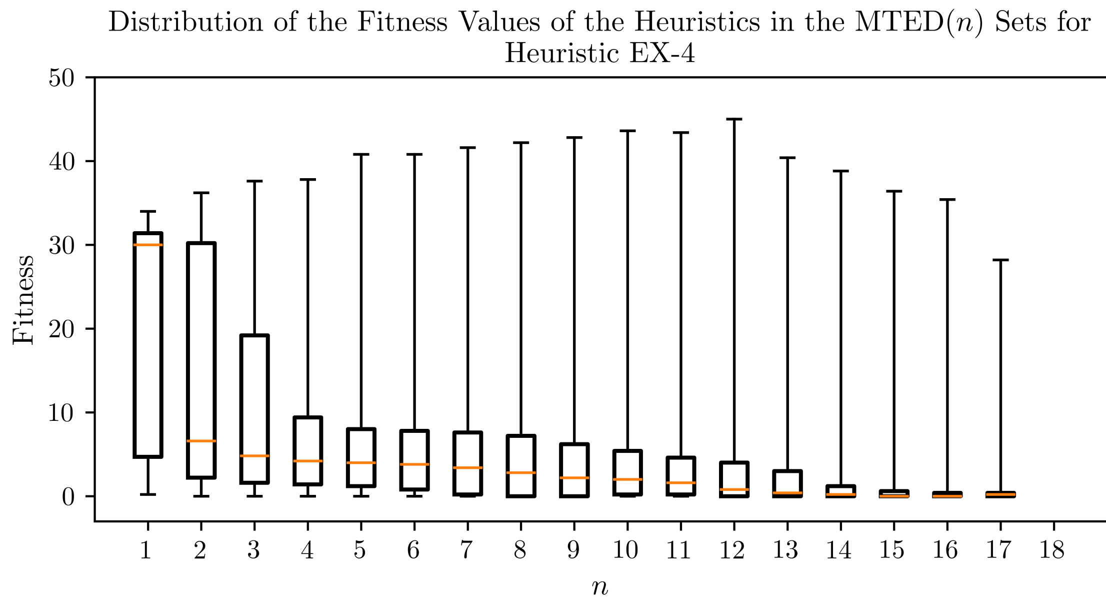


(b) Distribution of the fitness values of the heuristics in EX-2's MTED(n) sets. EX-2 is shown in Figure 5.1b and has a fitness of 7.0.

Figure 5.2: The distribution of the fitness values in four heuristic's MTED(n) sets. The sets are created from all heuristics in Language A of size ≤ 15. For each set, we show the median, minimum, maximum lower and upper quartile fitness values.



(c) Distribution of the fitness values of the heuristics in EX-3's MTED(n) sets. Heuristic EX-3 is shown in Figure 5.1c and has a fitness of 23.8.



(d) Distribution of the fitness values of the heuristics in EX-4's MTED(n) sets. Heuristic EX-4 is shown in Figure 5.1d and has a fitness of 31.4.

Figure 5.2: The distribution of the fitness values in four heuristic's MTED(n) sets. The sets are created from all heuristics in Language A of size ≤ 15. For each set, we show the median, minimum, maximum, lower and upper quartile fitness values. (Continued)

and n values ≤ 4, the largest MTED(n) sets are the MTED(4) sets for the heuristics EX-3 and EX-4, which both contain $7,698$ heuristics.

From the observations made in this subsection, we can see that, for small n values, a heuristic h's MTED(n) set could potentially be used to find heuristics with a higher fitness than h. We now define the *neighbourhood* of a program tree. The neighbourhood of a program tree is a similar concept to a heuristic's MTED(n) set, but is defined in more general terms. Its definition is as follows:

Definition 23 (Neighbourhood of a Program Tree)
Given a language L, a candidate program tree h and an integer representing the upper neighbourhood bound n, the neighbourhood $N(h, n)$ is defined as all valid program trees under L that can be obtained from h through a series of tree edits with a cost of at most n. We assume that each tree edit has a cost of 1. If the program tree is obvious from the context, we write $N(n)$.

In Definition 23 we do not stipulate that the tree edits have to be the ones described in Section 2.6, and it is feasible that this definition could be used with other types of tree edits. However, for the remainder of this thesis, we only use this definition with the edits described in Section 2.6. We have deliberately defined the neighbourhood in such a way so that it mirrors the MTED(n) set described previously, however the neighbourhood is a more general concept. The key difference between the neighbourhood and the MTED(n) set is that the MTED(n) set contains heuristics exactly n edits away from the candidate, while the neighbourhood contains program trees requiring edits with a cost of at most n to obtain from the candidate.

In the remaining sections in this chapter we use the neighbourhood of a heuristic in several experiments. We do not have an algorithm to calculate the neighbourhood of an arbitrary program tree, so we instead use the memoized results from the exhaustive enumeration experiments described in Chapter 4 to create each heuristic's neighbourhoods. Specifically, we use the heuristics in Language A of size ≤ 15. We use Algorithm 5.1 to create all the heuristic's neighbourhoods once, then memoize the results so they can be reused. Constructing the required neighbourhoods took 14 hours using the system described in Section 4.2.2.

The evidence laid out in this section suggests that neighbourhoods with upper bounds of at most 4 could be well suited for finding higher quality heuristics, or neighbours, and we hypothesise that this could be used as a form of program synthesis. In the next section we perform experiments to explore this relationship further, with

Algorithm 5.1 MEMOIZE-NEIGHBOURHOODS

Input:	H	Vector of memoized heuristics.
	n	Maximum neighbourhood bound used.
Output:		A map of pairs of heuristics h and integers i to sets of heuristics. Each map element describes the set of heuristics in h's $N(i)$ neighbourhood.

```
algorithm MEMOIZE-NEIGHBOURHOODS(H, n)
    m = []                                          ▷ Initialise map.
    for (i ∈ {0 ... H.SIZE() − 1}) do
        h_1 = H[i]
        for (j ∈ {i + 1 ... H.SIZE() − 1}) do
            h_2 = H[j]
            dist = MTED(h_1, h_2, γ)      ▷ Cost function γ is identical to that used
            if (dist ≤ n) then                in Section 2.6.
                m.AT({h_1, dist}).APPEND(h_2)  ▷ As the MTED is symmetric, only
                m.AT({h_2, dist}).APPEND(h_1)     one comparison is needed.
    return m
```

the goal of ascertaining whether the exploration of a heuristic's neighbourhood is a feasible basis for program synthesis.

5.3 Analysing a Heuristic's Neighbourhood

In this section we perform a set of experiments designed to ascertain whether exploring the neighbourhood of a heuristic could be used as the basis for a program synthesis method. The experiments described in this section make use of the memoized results from Chapter 4, using all heuristics in Language A of size ≤ 15. To be clear, the candidate heuristics and the neighbourhoods created only contain heuristics of size ≤ 15 - we do not consider any neighbours that are of a larger size.

The outline of this section is as follows; in Section 5.3.1 we examine the percentage of a candidate heuristic's neighbourhood that have a higher fitness than the candidate. In Section 5.3.2 we examine the size of a heuristic's neighbourhood. Finally in Section 5.3.3 we examine candidate heuristic's neighbourhoods to determine the number of evaluations required to find a fitter neighbour.

5.3.1 Percentage of Fitter Neighbours

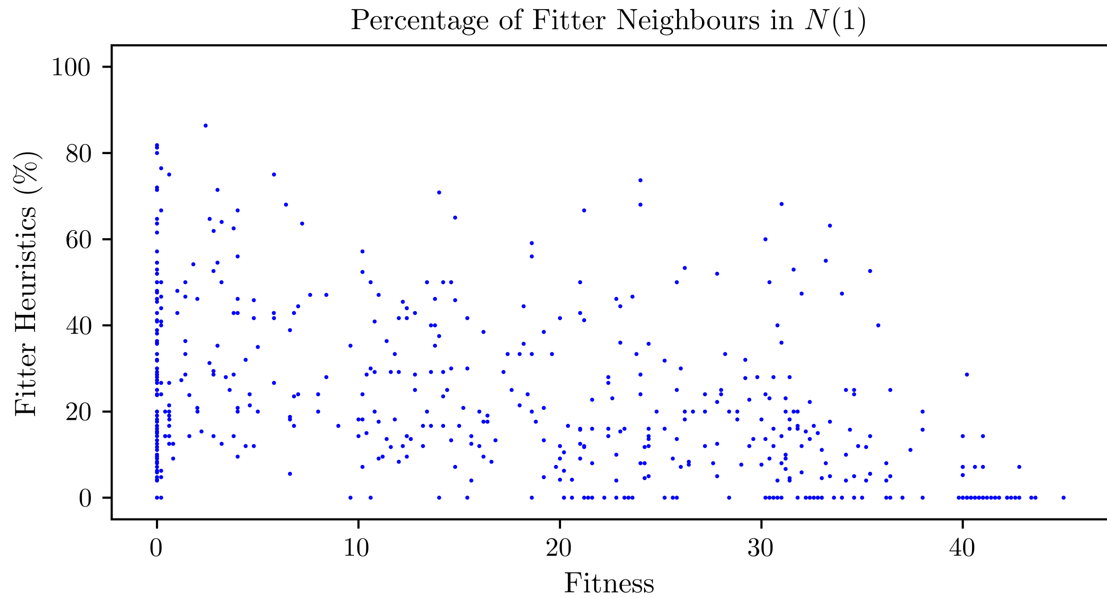
We begin by performing a set of experiments examining a heuristic h's neighbourhood to determine what percentage of h's neighbours have a higher fitness than h. This experiment is performed to determine the chance of finding a fitter neighbour if we were to probe the neighbourhood randomly.

The experiment is designed as follows; we split every heuristic in Language A of size ≤ 15 into sets. Each heuristic h with fitness $f(h)$ is put into one of the following sets: $f(h) = 0$, $0 < f(h) \leq 10$, $10 < f(h) \leq 20, \ldots, 40 < f(h) \leq 50$. From each of these sets we chose 100 heuristics randomly, and found the neighbourhood $N(h, n)$ for $n \in \{1 \ldots 4\}$. From the set of heuristics described by each neighbourhood, we determined the percentage of those heuristics that had a higher fitness than h. In Figure 5.3 we show these results.

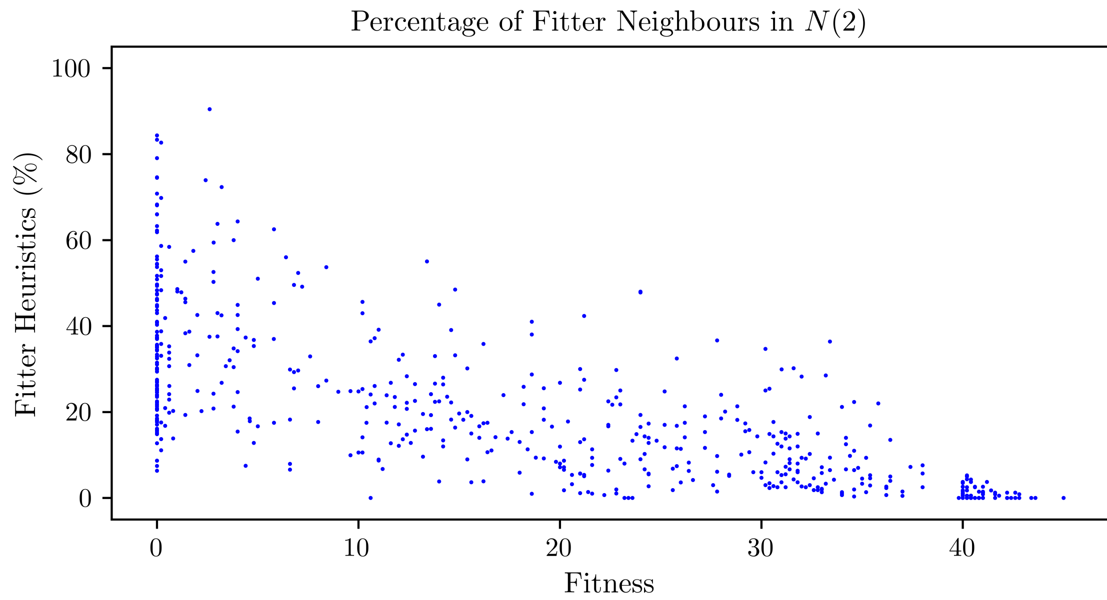
We can see that in each graph, there are many heuristics with a high proportion of neighbours that are fitter - that is to say, with higher fitness than the candidate heuristic. We can also see that there is generally a negative correlation between the proportion of a heuristic h's neighbourhood that are fitter, and h's fitness. This becomes more pronounced as we consider neighbourhoods with a higher n bound. In essence, a higher fitness corresponds to less neighbours that are fitter. This result was somewhat expected; we know from Table 4.4b that this set contains a small number of heuristics with a high fitness. It stands to reason that for heuristics with an already high fitness, there are fewer neighbours that have a higher fitness.

We can also see that for all neighbourhoods considered, there are some heuristics that have no fitter neighbours. We consider these heuristics to be the "optimum" heuristic in its neighbourhood. As we consider larger neighbourhood bounds, the number of optimum heuristics decreases. Yet, even for heuristics with a high fitness (> 40), there are still some heuristics whose $N(4)$ neighbourhoods contain no fitter neighbours. For lower n values, the proportion of candidate heuristics that have a low fitness value and a low proportion of fitter neighbours increases. This suggests to us that the smallest n values may be insufficient for consistently finding fitter neighbours.

Generally we can see that, as we consider higher bounded neighbourhoods, the proportion of a heuristic's neighbourhood that are fitter decreases. Yet for lower bounded neighbourhoods, the number of heuristics with no fitter neighbours increases. This suggests that there is a balance between neighbourhood upper bound and

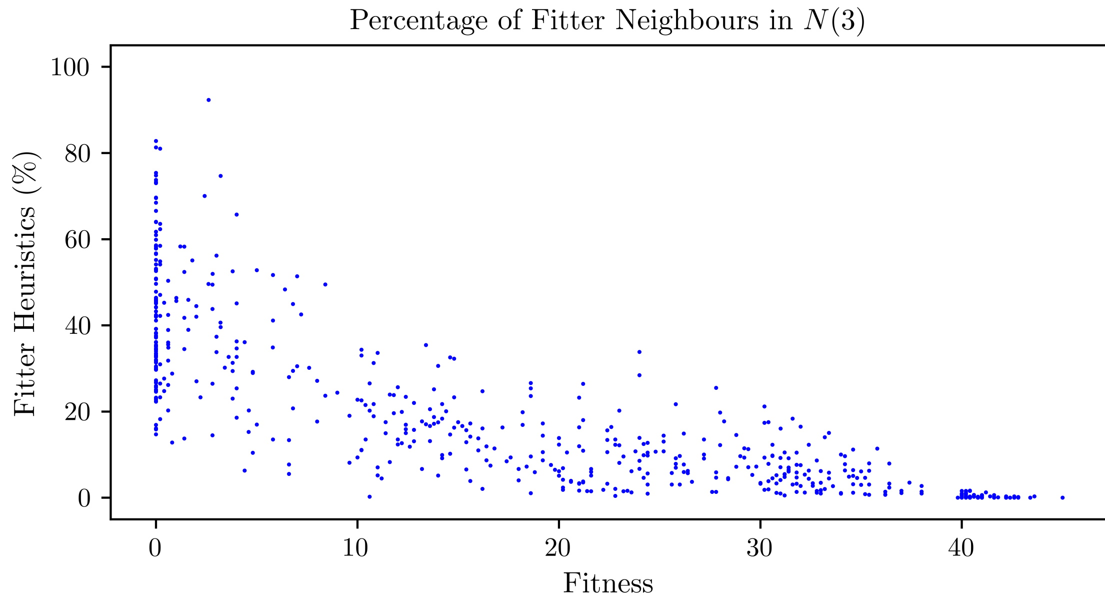


(a) The percentage of heuristics in neighbourhoods described by $N(1)$ that are fitter than the candidate heuristic.

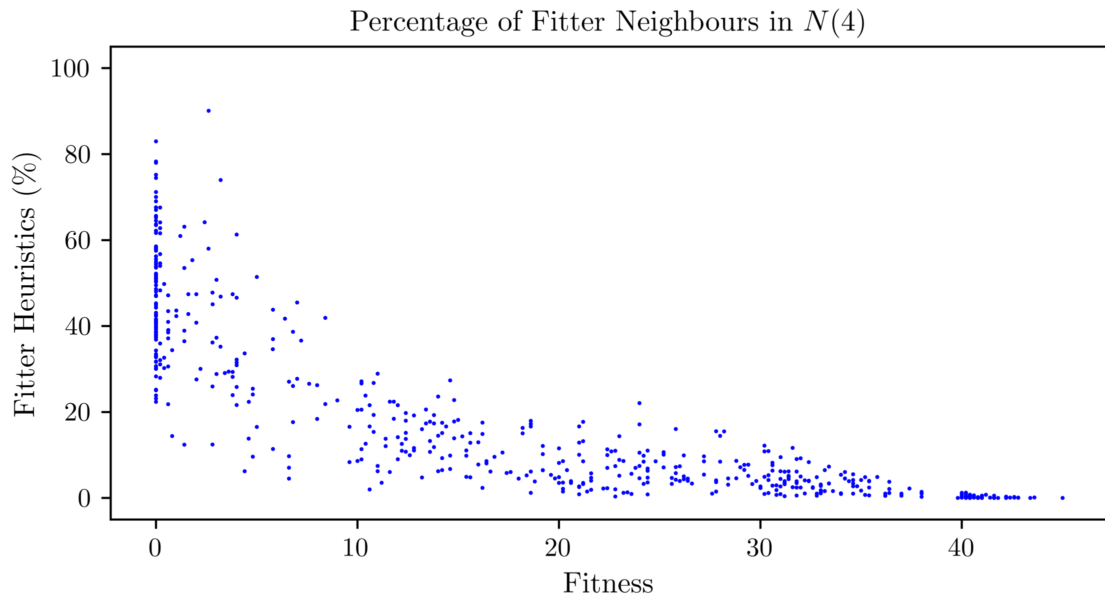


(b) The percentage of heuristics in neighbourhoods described by $N(2)$ that are fitter than the candidate heuristic.

Figure 5.3: The percentage of heuristics in neighbourhoods described by $N(n)$, where $n \in \{1 \ldots 4\}$. The candidate heuristics are in Language A of size ≤ 15. The same candidate heuristics are used in each experiment.



(c) The percentage of heuristics in neighbourhoods described by $N(3)$ that are fitter than the candidate heuristic.



(d) The percentage of heuristics in neighbourhoods described by $N(4)$ that are fitter than the candidate heuristic.

Figure 5.3: The percentage of heuristics in neighbourhoods described by $N(n)$, where $n \in \{1 \ldots 4\}$. The candidate heuristics are in Language A of size ≤ 15. The same candidate heuristics are used in each experiment. (Continued)

neighbourhood fitness quality; too small a neighbourhood bound, and there may be no fitter neighbours. Too large a bound and it may be difficult to find a fitter neighbour. In the next subsection we analyse the size of these neighbourhoods.

5.3.2 Size of Neighbourhood

In this subsection we present data concerning the size of the neighbourhoods of a set of heuristics, with the goal of understanding what the relationship is between the size of a heuristic and the size of its neighbourhood. We performed an experiment described as follows; we split all the heuristics in Language A of size ≤ 15 into sets according to their size. Each heuristic h with size $s(h)$ was put into one of the sets $s(h) \leq 10$, $s(h) = 11$, $s(h) = 12, \ldots, s(h) = 15$. We then chose 100 heuristics randomly from each set, and calculated their neighbourhoods $N(n)$ for $n \in \{1 \ldots 4\}$. In Table 5.2 we present statistical data regarding the size of these neighbourhoods.

From these results, we can draw two clear conclusions; firstly, for any neighbourhood $N(h, n)$, the size of the neighbourhood increases compared to the size of $N(h, n-1)$. This data reinforces our observations from Section 5.2. Secondly, larger heuristics generally have more neighbours. However, this is not always true, and we believe this will depend on the structure of the heuristic and the language used. For example, in Tables 5.2a and 5.2b we show the statistical data for $N(1)$ and $N(2)$. From that data we can see that for the heuristics of size ≤ 10, the neighbourhood size average is greater than that for heuristics of size 11.

We can also see that the difference between the neighbourhood sizes for $N(h, n)$ and $N(h, n-1)$ is not constant, and appears to grow exponentially. In the previous subsection we noted that there is a balance between the neighbourhood bound and the percentage of fitter neighbours. The data presented in this subsection suggests that higher neighbourhood upper bounds will produce much larger neighbourhoods, and so this should be taken into account when deciding what neighbourhood bound to use.

5.3.3 Finding a Fitter Neighbour

In this subsection we use the data about the size and quality of the heuristics in a neighbourhood to determine how quickly a fitter neighbour can be found if probing the neighbourhood randomly. The experiment we perform is described as follows; we split all heuristics in Language A of size ≤ 15 into sets. These sets are based on a

Table 5.2: Statistical data pertaining to the neighbourhood size of different sized heuristics. For each neighbourhood bound $n \in \{1 \ldots 4\}$ we split all the heuristics into sets according to their size, picked 100 heuristics from these sets randomly, generated those heuristic's neighbourhoods, and then calculated statistical data about the size of the neighbourhoods.

(a) Statistical data concerning the size of neighbourhoods described by $N(1)$.

Size of Heuristics in Set	Average	Min	Q_1	Median	Q_3	Max
≤10	12.08	0	8.0	11.0	17.0	17
11	10.57	6	10.0	11.0	12.0	13
12	13.3	4	11.0	14.0	15.0	16
13	16.18	6	14.0	17.0	18.25	19
14	18.8	7	17.75	21.0	22.0	22
15	20.87	9	17.0	22.5	25.0	25

(b) Statistical data concerning the size of neighbourhoods described by $N(2)$.

Size of Heuristics in Set	Average	Min	Q_1	Median	Q_3	Max
≤10	85.39	24	57.0	67.0	131.0	131
11	79.04	51	75.0	81.5	89.0	97
12	108.63	58	86.0	115.0	125.0	136
13	145.64	71	119.0	150.0	171.75	184
14	190.17	73	165.5	217.0	234.0	234
15	220.28	54	149.0	243.5	281.0	300

(c) Statistical data concerning the size of neighbourhoods described by $N(3)$.

Size of Heuristics in Set	Average	Min	Q_1	Median	Q_3	Max
≤10	367.31	83	264.0	343.0	552.0	552
11	429.87	282	416.0	445.0	462.0	500
12	605.77	307	498.0	625.0	677.0	736
13	846.72	441	692.0	897.0	988.0	1,066
14	1,194.57	446	1,024.25	1,357.0	1,477.0	1,478
15	1,367.96	180	760.0	1,472.5	1,919.0	2,087

Table 5.2: Statistical data pertaining to the neighbourhood size of different sized heuristics. For each neighbourhood bound $n \in \{1 \dots 4\}$ we split all the heuristics into sets according to their size, picked 100 heuristics from these sets randomly, generated those heuristic's neighbourhoods, and then calculated statistical data about the size of the neighbourhoods. (Continued)

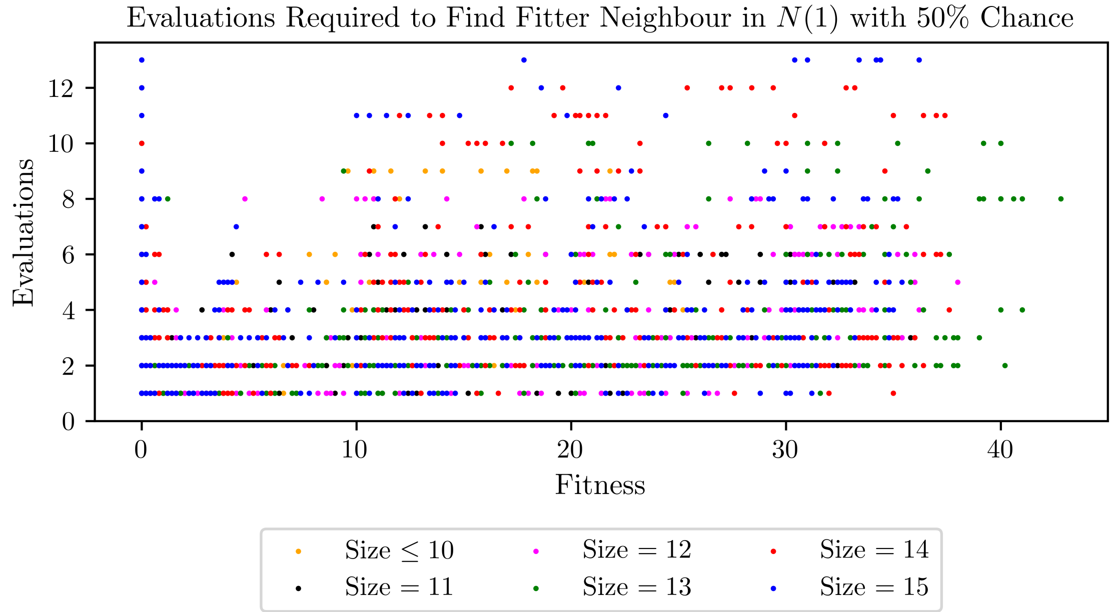
(d) Statistical data concerning the size of neighbourhoods described by $N(4)$.

Size of Heuristics in Set	Average	Min	Q_1	Median	Q_3	Max
≤10	1,345.12	370	1,063.0	1,488.0	1,746.0	1,746
11	2,102.26	1,505	2,015.25	2,146.0	2,260.25	2,433
12	2,700.54	1,584	2,471.5	2,867.0	2,973.5	3,263
13	3,783.28	2,139	3,173.75	4,090.0	4,396.5	4,489
14	5,458.1	1,853	4,865.5	6,127.0	6,640.0	6,673
15	5,864.28	482	2,710.0	6,137.5	9,012.0	9,818

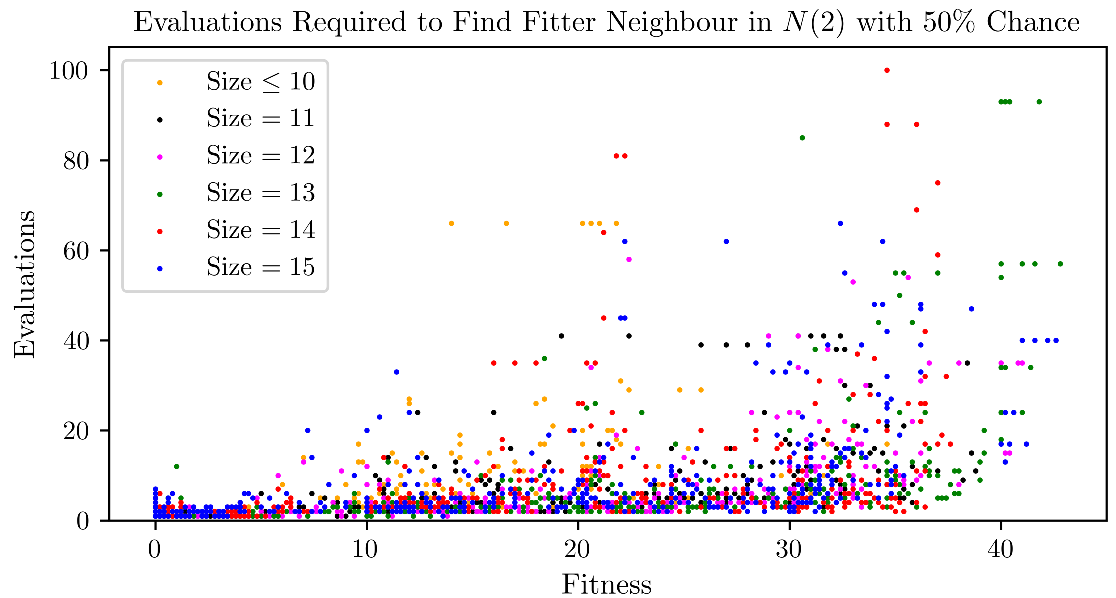
heuristic's fitness value and size. In total there are 36 sets. Each set can be described as a pair $(s,\ f)$, where s denotes the size of the heuristics contained in the set and f the fitness grouping of the heuristics contained within the set. The fitness and size values are those used in Sections 5.3.1 and 5.3.2. From each set (where possible) we chose 100 heuristics randomly. We then calculated the neighbourhood $N(n)$ for each heuristic for $n \in \{1 \dots 4\}$. From the information about the neighbourhood of each heuristic, we calculated the number of evaluations required to have a 50% chance of finding a fitter neighbour. In Figure 5.4 we show the results from this experiment.

In comparison to the other graphs, the results for $N(1)$ in Figure 5.4a are the most striking. There appears to be no heuristic that has a neighbourhood which would require a large number of evaluations to find a fitter neighbour - apart from those neighbourhoods that have no fitter neighbours.

However, for the larger neighbourhood bounds, we can see examples of heuristics that require many more evaluations than others in the same graph. In Figure 5.4b we can see that there is one heuristic that requires nearly 150 evaluations to have a 50% chance of finding a fitter neighbour, yet the majority of the other heuristics require far less evaluations. This trend continues as we consider higher neighbourhood bounds, and in Figure 5.4d for $N(4)$ some heuristics require several thousand evaluations to have a 50% chance of finding a fitter neighbour.

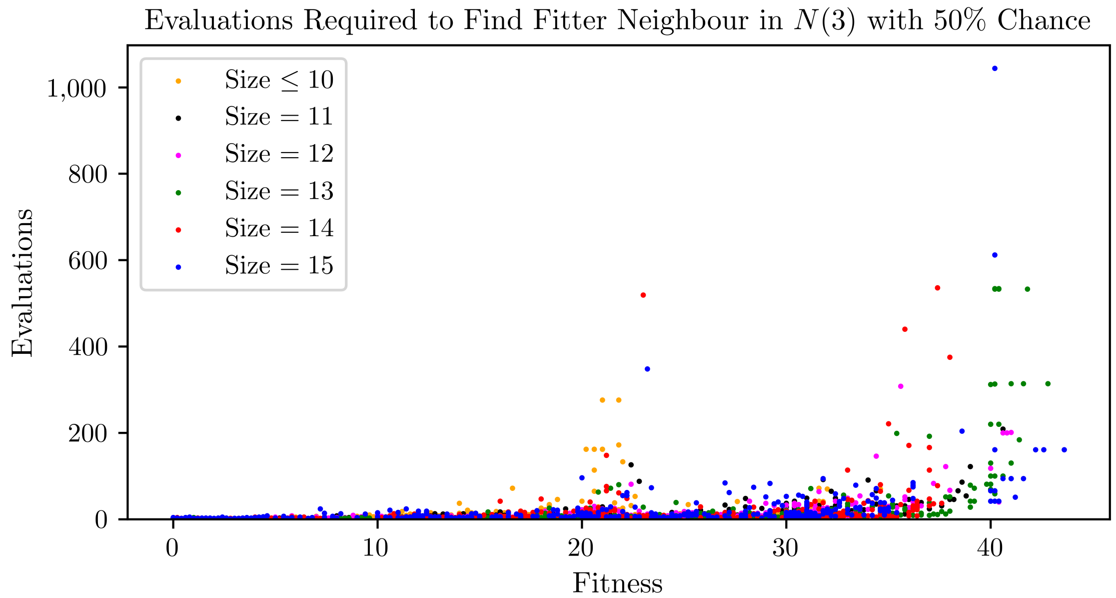


(a) The number of evaluations required to have a 50% chance of finding a neighbour fitter than the candidate heuristic in neighbourhoods described by $N(1)$.

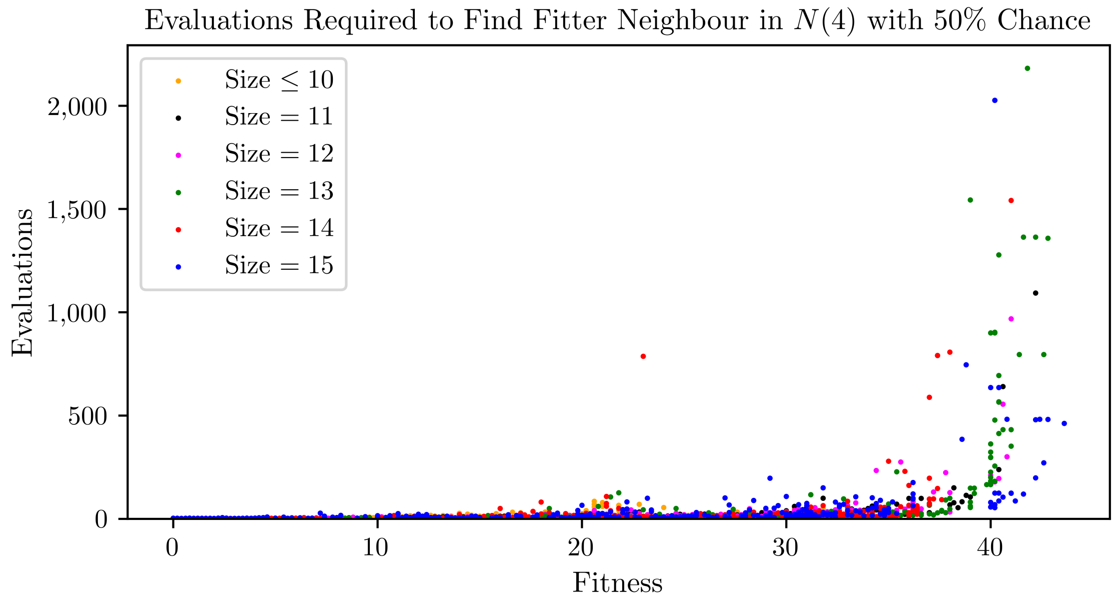


(b) The number of evaluations required to have a 50% chance of finding a neighbour fitter than the candidate heuristic in neighbourhoods described by $N(2)$.

Figure 5.4: The number of evaluations required to have a 50% chance of finding a neighbour fitter than the candidate heuristic in neighbourhoods described by $N(n)$, for $n \in \{1 \ldots 4\}$. The candidate heuristics are in Language A of size ≤ 15. We also show the size of each candidate heuristic.



(c) The number of evaluations required to have a 50% chance of finding a neighbour fitter than the candidate heuristic in neighbourhoods described by $N(3)$.



(d) The number of evaluations required to have a 50% chance of finding a neighbour fitter than the candidate heuristic in neighbourhoods described by $N(4)$.

Figure 5.4: The number of evaluations required to have a 50% chance of finding a neighbour fitter than the candidate heuristic in neighbourhoods described by $N(n)$, for $n \in \{1 \ldots 4\}$. The candidate heuristics are in Language A of size ≤ 15. We also show the size of each candidate heuristic. (Continued)

Apart from $N(1)$, all graphs show some change in the number of required evaluations as we pass a fitness of around 40. This was somewhat expected, as we already know that a tiny proportion of all heuristics have a fitness > 40, and it stands to reason that to find fitter neighbours, a larger number of evaluations will be required. Yet, we note that it is still possible to find fitter neighbours at these high fitness values, even if the number of evaluations may be comparatively high in number.

The reader may note that the data in Figure 5.4 is coloured according to the candidate heuristic's size. This data was collected to ascertain whether there was any noticeable difference in the number of required evaluations for different sized heuristics. Generally we see no evidence for this across the whole data set. However, for high fitness values there appear to be some heuristics of size ≤ 14 that require more evaluations to find a fitter neighbour than some heuristics of size 15. We believe this discrepancy is due to the fact that our subset of heuristics is bound at size 15. Therefore, we are not considering the heuristics of size 15's "true" neighbourhood, as they likely contain heuristics of larger sizes which are not taken into consideration in our experiments. We assume that we would see some change in these results if we had access to the complete neighbourhood.

In this section we have performed several experiments analysing the neighbourhoods of heuristics. Through these experiments, we believe that we have shown it is possible to find fitter heuristics by analysing a neighbourhood, that neighbourhoods are not too large to explore, and that generally the number of evaluations required to find a fitter neighbour is relatively small. From this data we believe that the upper bound on the neighbourhood size should be set at 3 for Language A. A heuristic's $N(n)$ neighbourhood appears to grow exponentially with n, and from these results $N(4)$ seems to offer little advantage compared to $N(3)$ when finding fitter heuristics. In the next section we perform experiments that are a continuation of those seen here, where we consider a local search algorithm that moves through the space of heuristics by continually probing the neighbourhood.

5.4 Simulated Local Search Experiments

In the previous section we presented data suggesting that the neighbourhood of a heuristic (defined in terms of the MTED metric) could be used to find fitter heuristics. In this section we perform experiments which simulate an iterative

Algorithm 5.2 LOCAL-SEARCH-GREEDY

Input:	f	Specification criteria represented as a fitness function.
	L	The language.
Output:		The program tree with the highest fitness found according to f.

```
algorithm LOCAL-SEARCH-GREEDY(f, L)
    currentProgram = INITIALISE(L)                    ▷ Create initial program tree.
    while (¬TERMINATION-CRITERIA-MET()) do
        nextPrograms = NEIGHBOURHOOD-GENERATION(currentProgram)
        change = False
        bestProgramFromSet = currentProgram
        for (program ∈ nextPrograms) do
            if (f(program) > f(currentProgram)) then
                bestProgramFromSet = program
                change = True
        if (change = False) then return currentProgram
        else
            currentProgram = bestProgramFromSet
    return currentProgram
```

algorithm that repeatedly explores neighbourhoods, where the candidate heuristic used in each iteration is the heuristic found from the previous iteration's neighbourhood exploration. In this way, we describe a method of program synthesis that we consider to be analogous to local search.

Such a local search algorithm is simple in its formulation; an initial heuristic is created and set as the candidate heuristic. Its neighbourhood is generated, and explored until a fitter heuristic found. That fitter heuristic is made the new candidate heuristic, and the process continues. We consider two local search algorithms in this section; one "greedy" (picking the fittest heuristic in a neighbourhood) and one "random" (picking the first heuristic found that is fitter). Pseudocode for these two algorithms, which we call LOCAL-SEARCH-GREEDY and LOCAL-SEARCH-RND, is shown in Algorithms 5.2 and 5.3. The reader should note that, when the neighbourhood is generated in both algorithms, the ordering of the returned heuristics is randomised. This is to ensure no bias is introduced into the results through the ordering of the heuristics.

Algorithm 5.3 LOCAL-SEARCH-RND

Input: f Specification criteria represented as a fitness function.
L The language.
Output: The program tree with the highest fitness found according to f.

```
algorithm LOCAL-SEARCH-RND(f, L)
    currentProgram = INITIALISE(L)                ▷ Create initial program tree.
    while (¬TERMINATION-CRITERIA-MET()) do
        nextPrograms = NEIGHBOURHOOD-GENERATION(currentProgram)
        change = False
        for (program ∈ nextPrograms) do
            if (f(program) > f(currentProgram)) then
                currentProgram = program
                change = True
                break
        if (change = False) then return currentProgram
    return currentProgram
```

The format of this section is as follows; in Section 5.4.1 we describe the methodology used in the local search experiments, and in Section 5.4.2 we present each experiment's results.

5.4.1 Methodology

Each local search experiment is described by the triple (ls, n, s), where ls is a local search algorithm, n is a neighbourhood bound and s is a heuristic size. Each experiment is performed as follows; the local search algorithm ls is initialised with a heuristic chosen randomly from all heuristics in Language A of size $\leq s$. The local search algorithm then progresses, with the neighbourhood generated on each iteration described by $N(n)$. The local search algorithm continues to termination. We note that the neighbourhoods created can only contain heuristics of size $\leq s$. Each experiment is performed $1,000$ times.

The local search algorithm ls used in each experiment is either LOCAL-SEARCH-GREEDY or LOCAL-SEARCH-RND, the n value used is in the integer range $\{1 \ldots 3\}$, and the s value used in the integer range $\in \{13 \ldots 15\}$. In total, 18 simulated local search experiments were performed. Together, these experiments took 4 hours to run

using the system described in Section 4.2.2. In Figure 5.5 we show the results from this experiment. Those graphs show the fitness value of the final heuristic found from each repetition. They are also ordered according to the fitness of the final heuristics. By visualising the results in this manner, we are able to make observations about the distribution of the final heuristic's fitness values found from each experiment.

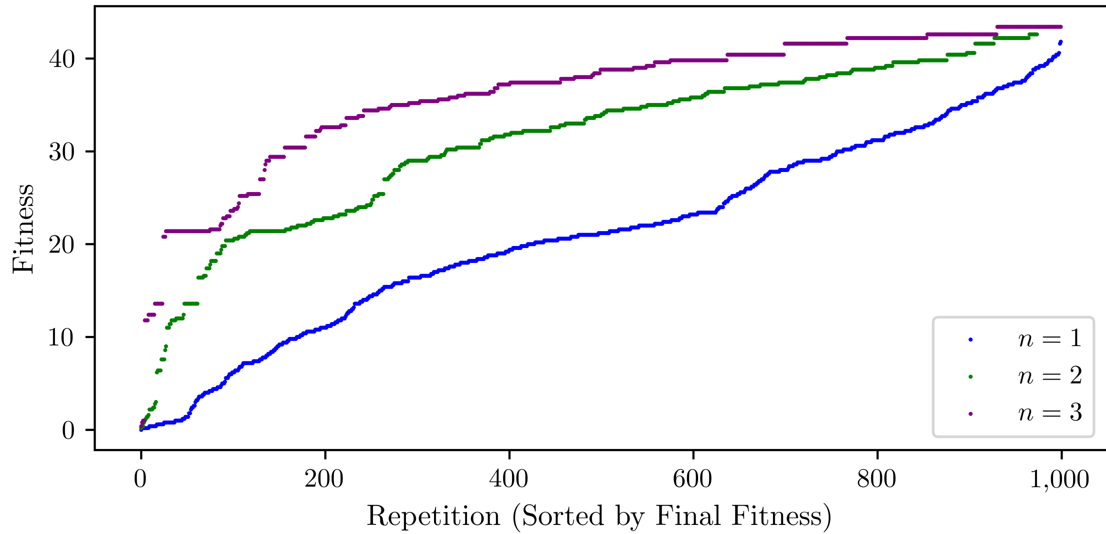
At an implementation level, these experiments are simulations, performed using the results from the experiments described in Chapter 4, and the neighbourhoods created from Algorithm 5.1. To be clear to the reader, the heuristics that can be created from these experiments are only those which have been previously memoized - therefore, it is impossible for these experiments to produce any new heuristics.

5.4.2 Results

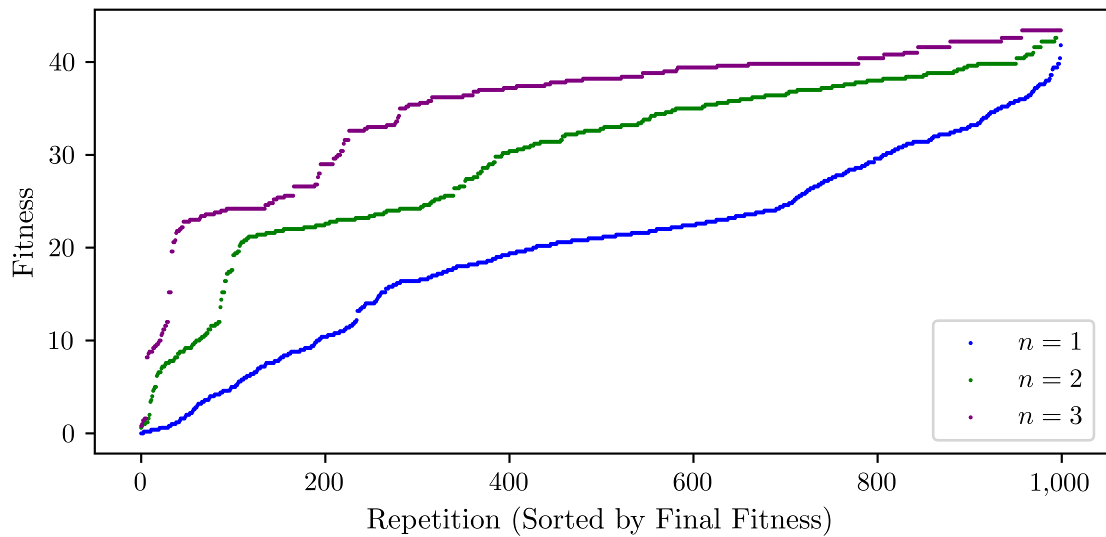
Each of the data points in each of the graphs in Figure 5.5 represents the fitness of the final heuristic found from a single repetition of local search. Using the terminology from Section 5.3, we could refer to these heuristics as the optimum in their neighbourhood. However, as we are emulating a local search algorithm, we feel the term "local optima" is more appropriate for these heuristics that have no fitter neighbours.

We can make two observations immediately about these results; firstly, the higher neighbourhood bound n used, the better the final quality of heuristic found. Secondly, the larger size of heuristics we consider, the better the quality of final heuristic found.

Of particular interest to us is the distribution of the final heuristic's fitness in each experiment. We had expected there to be some uniformity to this data; perhaps it would be somewhat linear, or plateau at high fitness values. Instead, there appear to be areas where there are many more heuristics with similar fitness values clustered together. These are not plateaus concentrated around higher fitness values, but at somewhat "average" fitness values. An example can be seen in the $n = 2$ and $n = 3$ simulations at a fitness value of ≈ 20 in Figure 5.5a. The distribution of fitness values initially rises quickly, then plateaus, before rising again. Our interpretation of this is that these heuristics are in basins. The overarching algorithm has been unable to select the correct sequence of edits to create a heuristic with a higher fitness value. However, as to why there is a basin at this fitness value is perplexing to us; no other data we have seen gives any indication as to why there would be a greater number of heuristics at this fitness value that have no fitter neighbours.

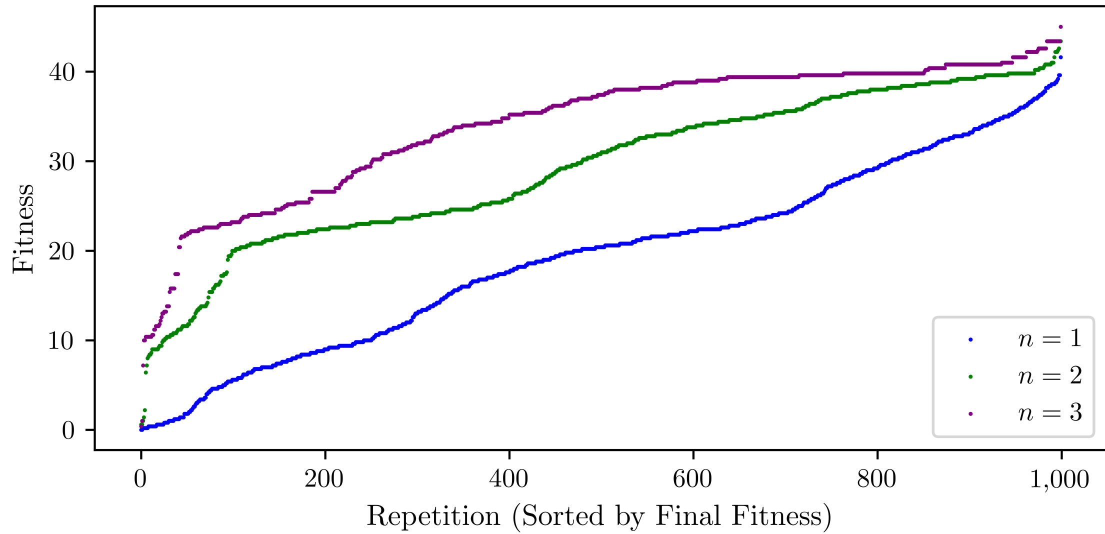


(a) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 13 using the algorithm Local-Search-Rnd.

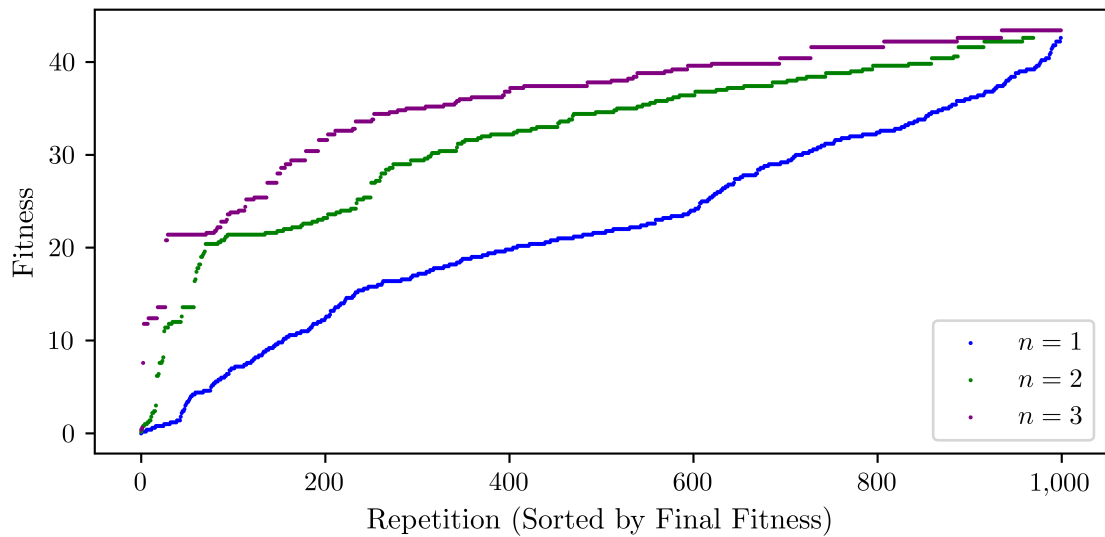


(b) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 14 using the algorithm Local-Search-Rnd.

Figure 5.5: Results from the simulated local search experiments performed on various subsets of heuristics in Language A. Each data point represents the final fitness of the heuristic found from that repetition of local search. For each experiment $1,000$ repetitions were performed.

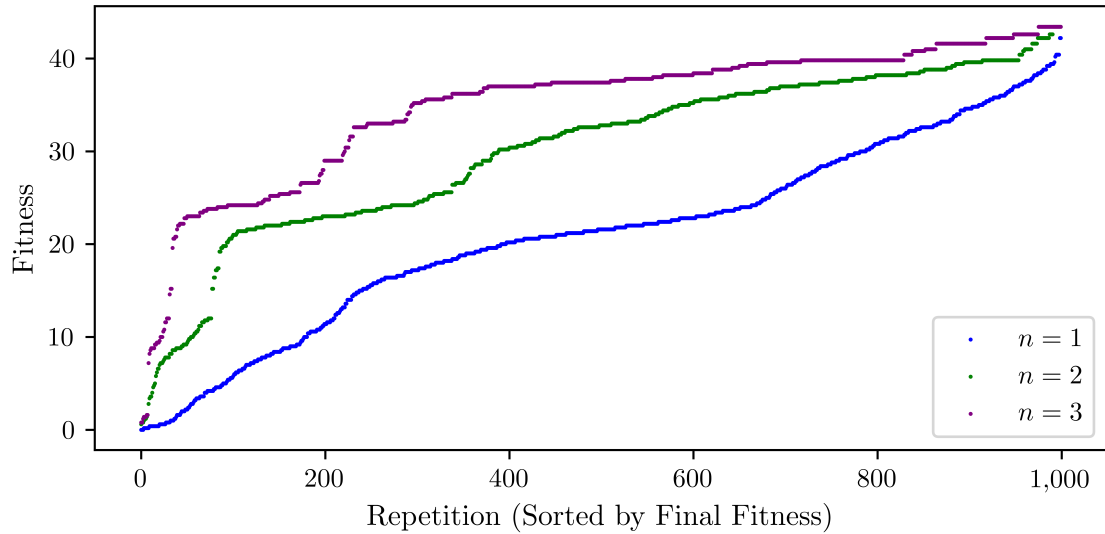


(c) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 15 using the algorithm Local-Search-Rnd.

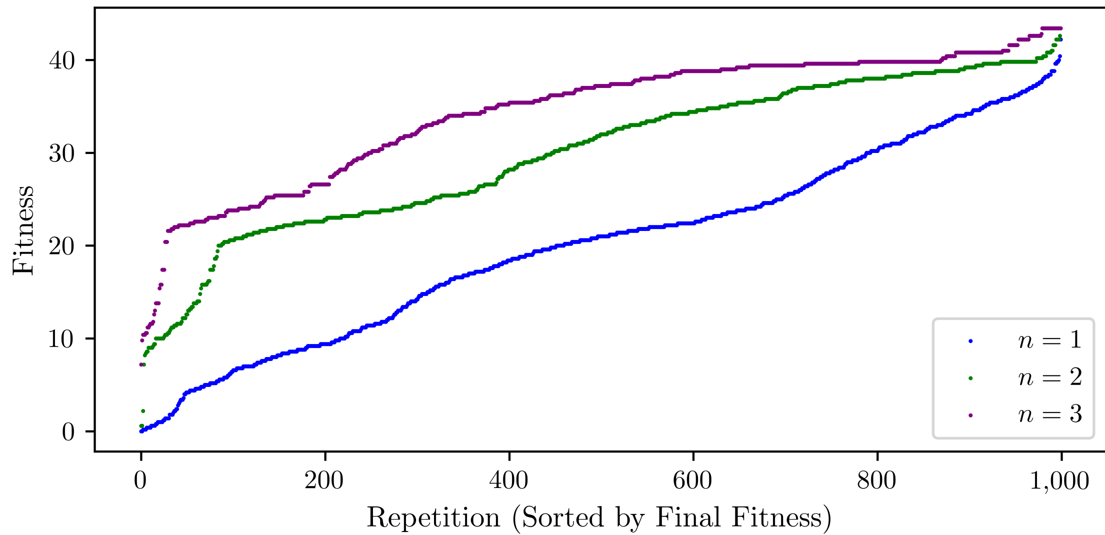


(d) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 13 using the algorithm Local-Search-Greedy.

Figure 5.5: Results from the simulated local search experiments performed on various subsets of heuristics in Language A. Each data point represents the final fitness of the heuristic found from that repetition of local search. For each experiment $1,000$ repetitions were performed. (Continued)



(e) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 14 using the algorithm Local-Search-Greedy.



(f) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 15 using the algorithm Local-Search-Greedy.

Figure 5.5: Results from the simulated local search experiments performed on various subsets of heuristics in Language A. Each data point represents the final fitness of the heuristic found from that repetition of local search. For each experiment $1,000$ repetitions were performed. (Continued)

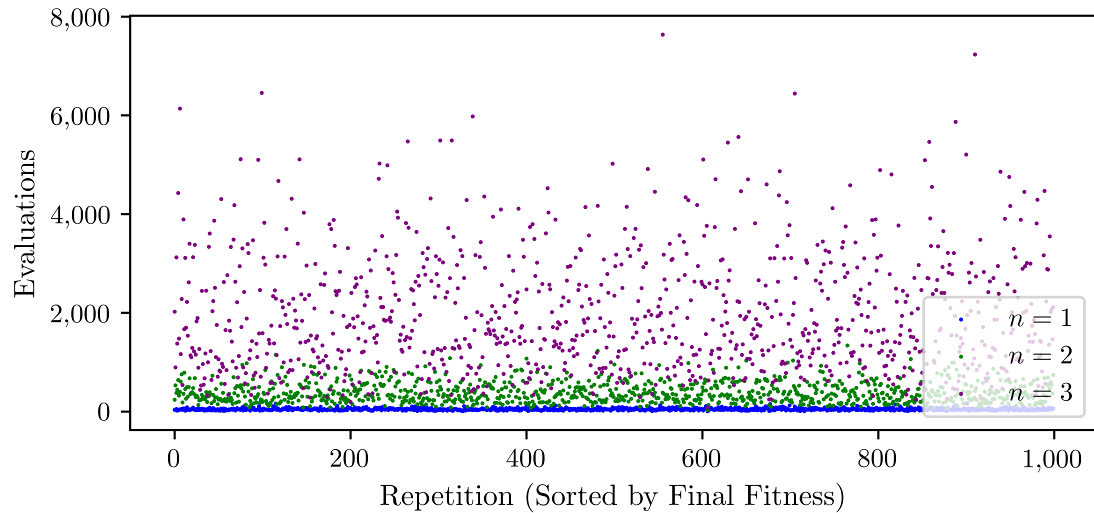
When we compare the algorithms LOCAL-SEARCH-GREEDY and LOCAL-SEARCH-RND to each other, we would state that there appears to be no clear difference between the two regarding the quality of results. In Figure 5.6 we show data pertaining to the number of evaluations required for all repetitions and neighbourhood bounds of both local search algorithms using heuristics in Language A of size ≤ 15. This data shows us that when using the LOCAL-SEARCH-RND algorithm, the number of evaluations required to find the local optima is much less than when using the LOCAL-SEARCH-GREEDY algorithm. In turn, this suggests that the LOCAL-SEARCH-GREEDY algorithm may be less desirable for real-world use. Though we do not present the data, this trend was seen in all comparisons of the two algorithms. We can also see from this data that there is no correlation between final heuristic fitness and the number of evaluations required for the local search to terminate. This tells us that, at least for the subset of heuristics considered, the best heuristics do not necessarily require more computational resources to find.

In Figure 5.7 we show the results from Figure 5.5 with the duplicate final heuristics removed. In these graphs we can clearly see that there are far fewer unique data points, which tells us that many search derivations are reaching the same local optima. This effect is more pronounced for larger neighbourhood bounds - for example, only 310 unique heuristics were found from all repetitions of the local search experiment described by the triple (LOCAL-SEARCH-RND, 3, 15).

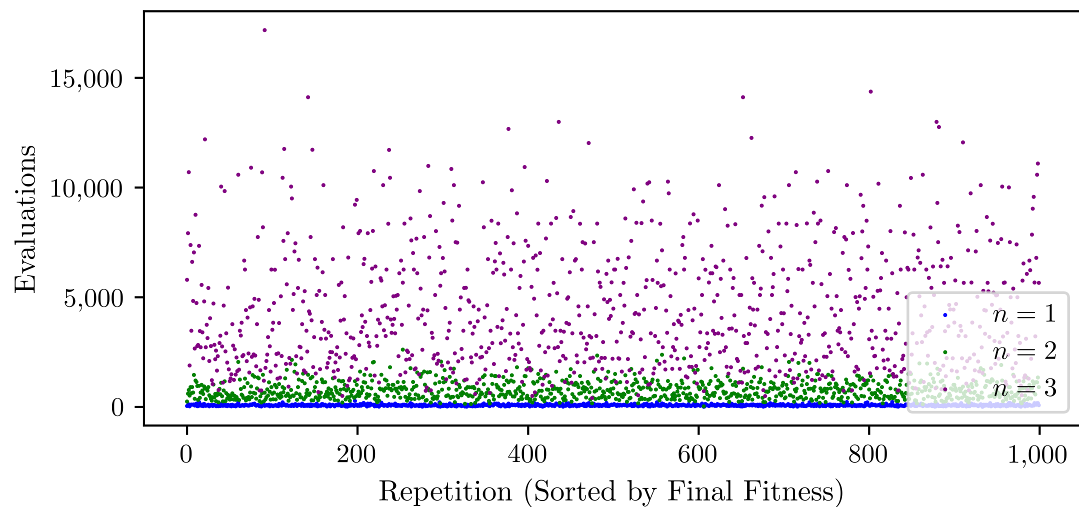
To explore this relationship further, we calculated the MTED between these 310 unique heuristics. We did this to gain insight into how "close" they are to each other under this metric. For example, if they are just outside the neighbourhood bound used, it would suggest that the heuristics are in basins relatively close to each other. If they are further away, it would suggest that there are multiple areas of the search space where there are heuristics with high fitness values.

In Figure 5.8 we show each heuristic's MTED to the fittest of the 310 heuristics plotted against its fitness. We can see that many heuristics are not near the bound of the neighbourhood (3), instead appearing to be much further away. We also calculated the fitness distance correlation from this data [91] and note that this suggests that there is some correlation between fitness and distance.

In Figure 5.9 we show the MTED between the fittest 20 heuristics from this set. Though this is only a snapshot of the full matrix, we can see that the distance between most heuristics is quite large, and very few lie just above the neighbourhood

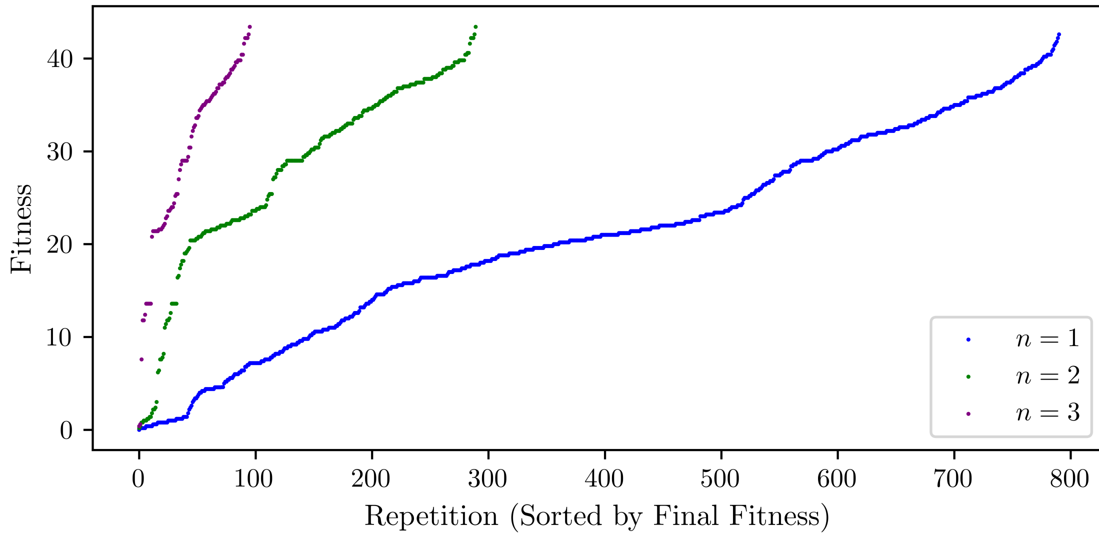


(a) The number of evaluations performed in each repetition of the simulated local search experiments performed on heuristics in Language A of size ≤ 15. These results are from the experiments that used the algorithm Local-Search-Rnd.

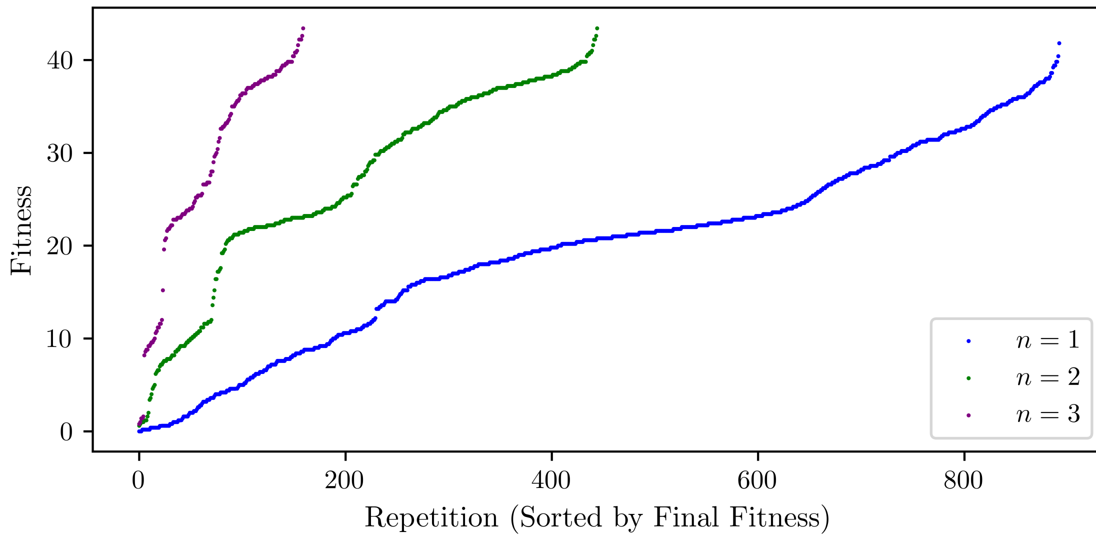


(b) The number of evaluations performed in each repetition of the simulated local search experiments performed on heuristics in Language A of size ≤ 15. These results are from the experiments that used the algorithm Local-Search-Greedy.

Figure 5.6: The number of evaluations performed in each simulated repetition of local search using heuristics in Language A of size ≤ 15.

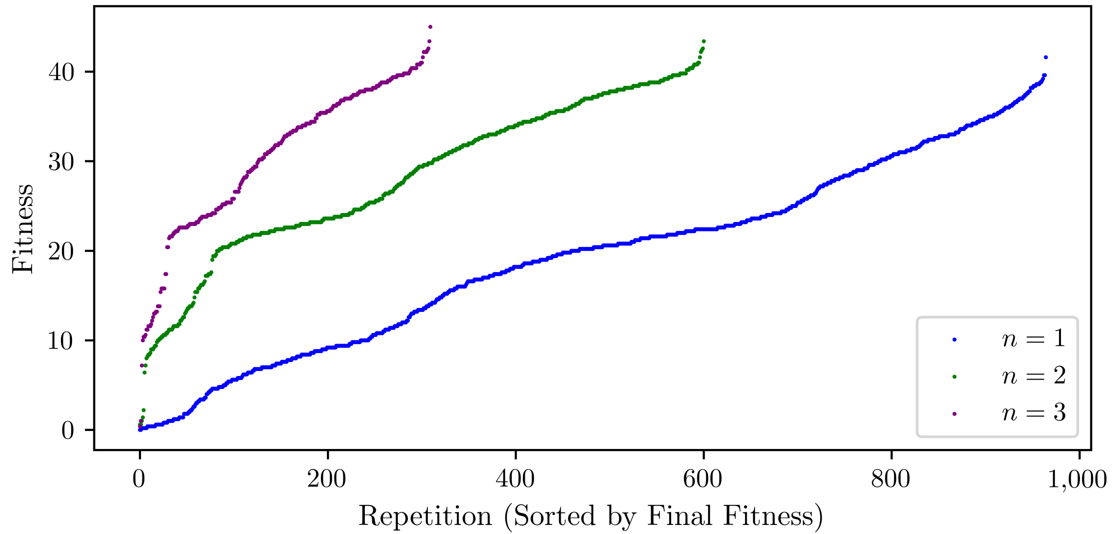


(a) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 13 using the algorithm LOCAL-SEARCH-RND, with duplicate results removed.

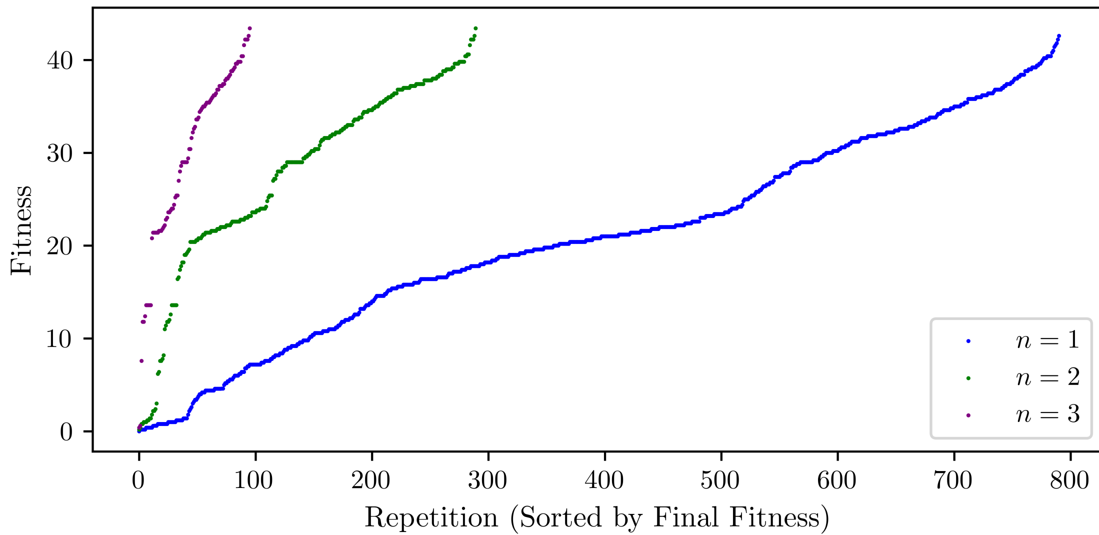


(b) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 14 using the algorithm LOCAL-SEARCH-RND, with duplicate results removed.

Figure 5.7: Results from the simulated local search experiments performed on various subsets of heuristics in Language A, with duplicate results removed. Each data point represents the final fitness of the heuristic found from that repetition of local search. For each experiment $1,000$ repetitions were performed.

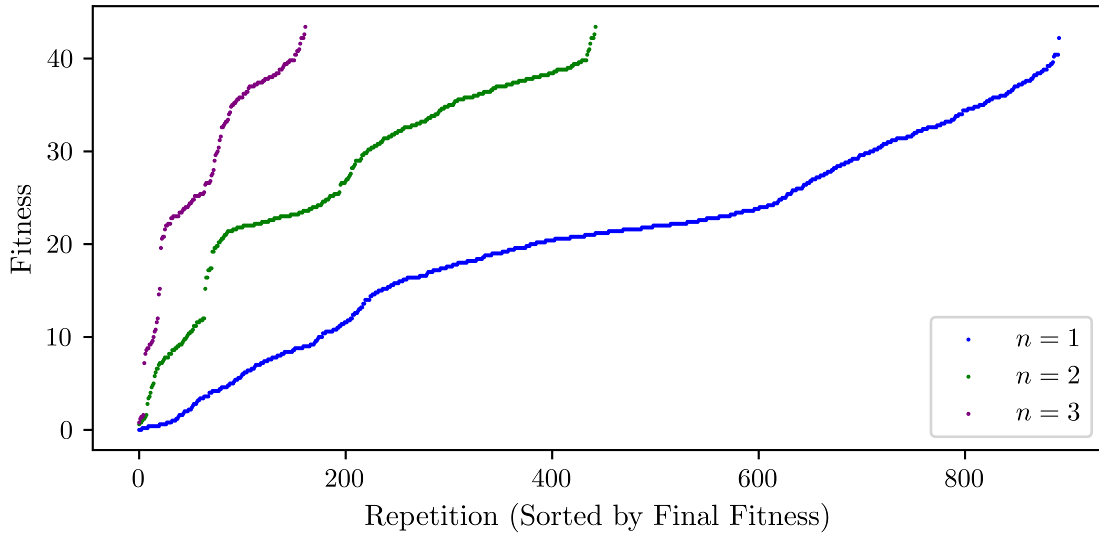


(c) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 15 using the algorithm LOCAL-SEARCH-RND, with duplicate results removed.

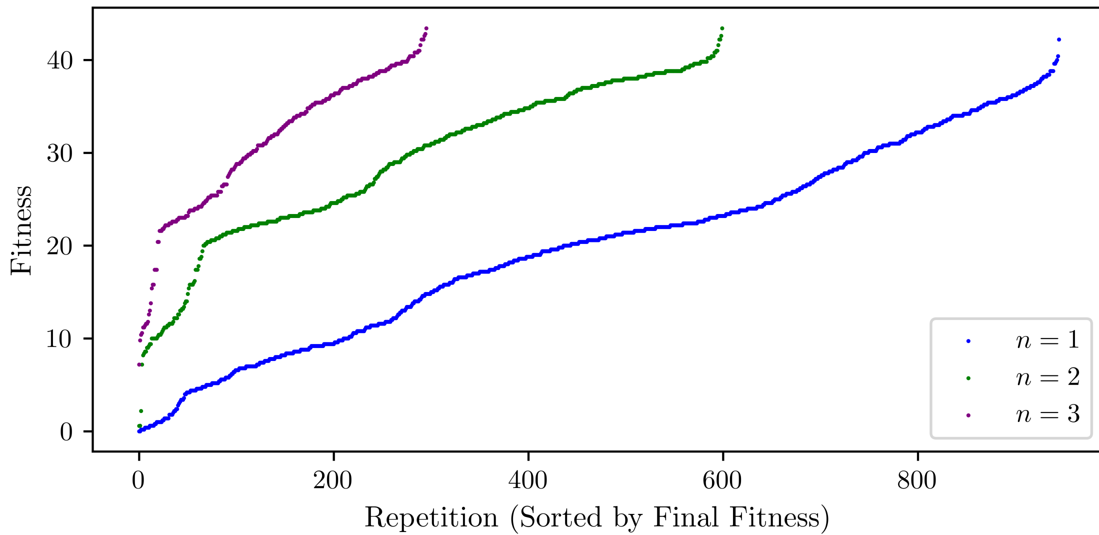


(d) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 13 using the algorithm LOCAL-SEARCH-GREEDY, with duplicate results removed.

Figure 5.7: Results from the simulated local search experiments performed on various subsets of heuristics in Language A, with duplicate results removed. Each data point represents the final fitness of the heuristic found from that repetition of local search. For each experiment $1,000$ repetitions were performed. (Continued)



(e) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 14 using the algorithm Local-Search-Greedy, with duplicate results removed.



(f) Results from the simulated local search experiments performed on heuristics in Language A of size ≤ 15 using the algorithm Local-Search-Greedy, with duplicate results removed.

Figure 5.7: Results from the simulated local search experiments performed on various subsets of heuristics in Language A, with duplicate results removed. Each data point represents the final fitness of the heuristic found from that repetition of local search. For each experiment $1,000$ repetitions were performed. (Continued)

bound. Of interest to the reader may be the two entries where the MTED is reported as being below the bound 3. Upon further investigation it was found that both pairs of heuristics were semantically identical to each other, and had reported the exact same fitness. Since our local search method only moves to the fitter heuristic, both heuristics in each pair are local optima, and exist on a plateau together.

Other researchers have performed experiments using the fitness distance correlation to analyse the relationship between heuristics, such as Ochoa, Qu, and Burke [136]. This work is much more comprehensive in scale than ours, and performed on a solution space that is enumerable - whereas the work described here is performed on a partial search space which is computationally expensive to produce. In future work, a more comprehensive analysis on the landscape of heuristics when analysed using the MTED metric could provide invaluable insight into how to improve our local search algorithm, and allow us to produce more effective LS-SAT heuristics.

5.5 Discussions & Conclusions

In this chapter we have performed an analysis on the heuristics in Language A of size ≤ 15 using the MTED metric. We have shown that neighbourhoods of heuristics can be defined through this metric, and illustrated how these neighbourhoods can be probed to find fitter heuristics. We then showed how, through the sequencing of the probing of neighbourhoods, we can simulate local search on this subset of heuristics. Through these simulations, we presented evidence suggesting that local search is a viable method of program synthesis.

From the results in Section 5.3 we surmised that, while a large neighbourhood bound would produce fitter neighbours, it could be considered too large to explore effectively. A neighbourhood described by $N(3)$ was considered to be the most effective; providing opportunities for fitter neighbours to be found, but not requiring a large number of evaluations to do so. In Section 5.4 we showed how highly effective heuristics could be created using two different local search algorithms. However, we found that the algorithm Local-Search-Greedy produced results that required many more evaluations to obtain than Local-Search-Rnd. Generally the quality of heuristics produced from all local search experiments was high, with those that used larger neighbourhood bounds producing the fittest heuristics.

Some of the results have shown us that larger neighbourhood bounds generally

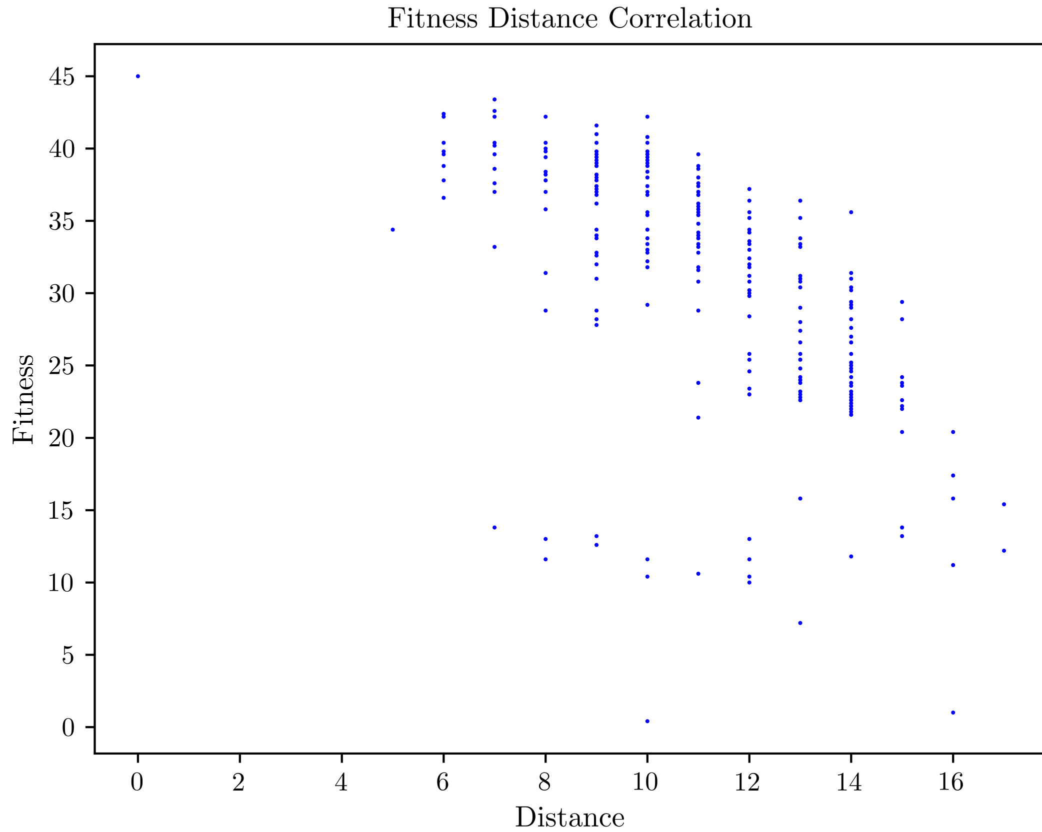


Figure 5.8: Graph showing the fitness distance data for the 310 unique heuristics found from the 1,000 runs of the local search experiment described by the triple (Local-Search-Rnd, 3, 15). For each heuristic h we show the fitness of h plotted against the MTED from h to the fittest of the 310 heuristics. The fitness distance correlation [91] - the relationship between fitness and distance - is -0.62749, suggesting some correlation between distance and fitness.

0																			
7	0																		
7	4	0																	
6	8	9	0																
10	10	7	10	0															
6	9	10	6	11	0														
8	5	4	9	7	9	0													
7	7	6	5	8	8	6	0												
9	6	7	8	8	8	7	8	0											
9	5	7	9	10	9	9	10	7	0										
9	10	9	9	7	7	8	7	8	11	0									
10	10	8	7	10	7	8	6	9	11	10	0								
10	10	8	7	10	7	8	6	9	11	10	2	0							
10	7	9	7	12	10	10	9	7	9	12	10	10	0						
10	7	9	7	12	10	10	9	7	9	12	10	10	2	0					
9	8	5	9	5	9	5	7	5	9	7	7	7	8	8	0				
8	10	9	9	9	11	10	6	8	12	7	11	11	12	12	9	0			
10	10	9	5	11	7	8	6	10	11	10	6	6	10	10	8	11	0		
7	7	8	7	7	6	8	4	6	9	7	8	8	9	9	7	6	9	0	
6	9	9	10	10	6	10	9	10	10	11	10	10	12	12	11	12	8	6	0

Figure 5.9: Matrix showing the MTED between the fittest 20 unique heuristics returned from the 1,000 runs of the local search experiment described by the triple (Local-Search-Rnd, 3, 15). As the MTED is symmetric, we only show the lower half of the matrix.

contain fitter heuristics. However, they have also shown us that large neighbourhood bounds can produce large neighbourhoods. As a potential avenue of future research, we suggest that an algorithm which uses a dynamic neighbourhood bound could prove to be effective at navigating the search space of heuristics. That is, a neighbourhood bound that changes as the algorithm progresses - for example, if the algorithm can find no fitter neighbours. Such an algorithm could partially alleviate concerns about the sizes of neighbourhoods with large bounds, yet would still allow them to be explored. In essence such an algorithm would equate to a variable neighbourhood search (VNS).

Through the experiments presented in this chapter, we believe we have provided enough evidence that local search on program trees, or *local search program synthesis*, is a viable technique for creating LS-SAT heuristics. However, in all the experiments that we have conducted, we have used memoized results, and performed computationally expensive calculations to find the MTED for all the heuristics in the set considered. For this local search algorithm to perform in a real-world setting, we require an algorithm to generate a neighbourhood. In the next chapter we present such an algorithm.

Chapter 6

Neighbourhood Generation

6.1 Introduction

In the previous chapter we showed how LS-SAT heuristics created using Language A could be compared to each other using the MTED metric. For a heuristic p and integer n representing the upper neighbourhood bound, we defined the neighbourhood of p as containing all heuristics that have a MTED cost of at most n from p. Taking a small subset of the results obtained in Chapter 4, we calculated the neighbourhoods for these heuristics and used them to perform local search on this subset of the search space. Through these results we provided evidence that, using this definition of a neighbourhood, local search is a viable method of program synthesis for this domain.

However, the method used to compute a heuristic's neighbourhood is impractical for a real-world local search algorithm. To understand why, let us examine the methodology again; first we created a set of heuristics S_m defined as all those heuristics containing at most m terms. Using the Cartesian product $S_m \times S_m$ we computed the MTED value between all unique pairs in this set. Finally, to find the neighbourhood of a heuristic p with a MTED cost of at most n from p, we filtered the results that met this criteria. In Algorithm 5.1 we showed pseudocode for this process.

Perhaps the most obvious reason for this being an impractical methodology concerns the size of the search space, as we know that it is infinite. Therefore algorithms based on this methodology will never be able to explore the entire search space. Even if we limited the search to only those heuristics with up to m terms, for many languages used with program synthesis techniques, the number of program

trees in relation to m grows exponentially. As this methodology requires storing all enumerated heuristics, for large values of m the storage requirement becomes impractically large.

The second step in this methodology requires the calculation of the MTED cost between all possible pairs in S_m. As the MTED metric is symmetric, for a set of size k, exactly $\frac{k^2-k}{2}$ calls to the MTED algorithm are needed. As the size of S_m grows exponentially with m, and the number of required calls grows quadratically with the size of the input set, it is infeasible to perform all the required MTED calculations for large values of m.

Through these observations, we believe that the method of computing a heuristic's neighbourhood described in Algorithm 5.1 to be impractical for a real-world local search algorithm. For our use-case, we require an algorithm that can create a heuristic's neighbourhood efficiently. We also require it to work on heuristics containing a reasonably large number of terms, so that the search is not constrained when exploring the infinite search space. Finally, we would like our method to be generalised enough that it can be used for any generic program tree - specifically, program trees written under languages that use a type system, as described in Section 2.5.1. From this point on, we refer to a program tree's neighbourhood, rather than a heuristic's neighbourhood, so as to remain consistent with the literature in Section 2.5 concerning program synthesis. In this chapter, we present an algorithm that meets these criteria.

The format of this chapter is as follows; in Section 6.2 we analyse the requirements of the neighbourhood generation algorithm, before formalising the inputs and outputs in a function prototype. We present this in an object-orientated paradigm, and call the object NEIGHBOURHOOD-GENERATION. The function prototype consists of a constructor and the function that returns a neighbourhood, GENERATESUCCESSORS. Informally in Section 6.3, we consider two different methods of generating the neighbourhood for a program tree; one that considers the entire input program tree as valid for performing edits on, and the other that relies on recognising patterns in a program tree, and re-using previously discovered edit sequences. We show how the first of these is infeasible.

In Section 6.4, we take the remaining methodology considered in the previous section, and use it to define GENERATESUCCESSORS. We specifically formalise the idea of the pattern and edit sequence, before abstracting these concepts. We

show how patterns are recognised and edits applied to the input program tree. In Section 6.5 we describe a randomised version of GenerateSuccessors, which allows the neighbourhood of a program tree to be probed without constructing the entire neighbourhood.

In Section 6.6, we show how the constructor for Neighbourhood-Generation is defined. Succinctly, it works by enumerating all possible patterns that could be encountered in a program tree, and then finds all possible edit sequences that could be relevant for generating any output program trees. Finally in Section 6.7 we present our conclusions from the research presented in this chapter.

6.2 Function Signature

In this section we analyse our use-case of the neighbourhood generation algorithm; that is to say, its role as a generator of a neighbourhood in a local search algorithm. Using the observations from this analysis, we show a function signature that we feel fulfils our needs from such an algorithm. We also re-introduce Language EX-1 (originally introduced in Figure 2.7), which is used in this chapter to show examples of how the algorithm works. We then discuss the limitations of the algorithm - specifically, those regarding what types of languages can be used with it. To be clear to the reader, the algorithm described in this chapter is a generic algorithm, for use with any typed language - not just those we use it with in this thesis.

To begin, let us formalise exactly what is required of the algorithm. In the introduction to this chapter, we stated that the algorithm Memoize-Neighbourhoods (see Algorithm 5.1) can be used to memoize the neighbourhoods of a set of program trees. This memoized data allows us to, when given a program tree p and a maximum neighbourhood bound n, get the set of program trees whose MTED from p is less than or equal to n. It is this functionality that we wish to emulate.

However, there are several generalisations we can make to this initial specification. In Chapter 5 we assumed that each edit had a cost of 1. However, in some MTED algorithms, different cost functions can be used. A cost function in the context of an MTED algorithm describes an attributed cost for each insertion and deletion of a node, as well as the cost of relabelling one node with another. For example, an insertion of a node t may have a cost of 3. There are some requirements of such a cost function for use with an MTED algorithm, which we discussed in Section 2.6.

The algorithm we describe in this chapter does not allow an arbitrary cost function to be used with it, but we do describe it in a form that allows some generalised use that retains consistency with the cost function requirements given in Section 2.6. Each term t in a language L has an associated cost, which we reference by writing $\textsc{Cost}(t)$. The cost of an insertion or deletion of t is $\textsc{Cost}(t)$. The cost of relabelling t_1 with t_2 is given as $\max(\textsc{Cost}(t_1), \textsc{Cost}(t_2))$. Relabelling a node with itself has a cost of 0.

Rather than the algorithm returning a single set of program trees, we design it so that it returns a structure that contains information pertaining to the cost and number of edits from the input program tree to each output program tree. The structure is in the form of a collection, where each element contains a map, which in turn contains a set of program trees. Each outer collection is indexed by k and each map indexed by i. A set of program trees at a specific k and i contains all program trees with an MTED from p that has a combined cost of k, and uses exactly i edits. This way, we can either consider the returned program trees as sets separated by the cost of the edits to create them, or as a single set through some inexpensive post-processing.

Throughout Chapter 5 we can observe that the n values given to the experiments were taken from a small pool of values. In all of the local search experiments performed, we generated program trees whose cost of edits was from between 1 and 3 from the candidate program tree. We assume that the maximum n value is known beforehand, and can be used to initialise the neighbourhood generation algorithm.

Though it may seem an obvious assertion, the input program trees provided to the algorithm, and those returned, should all be consistent with the type rules of the language; that is to say, they should be type safe.

Concerning the language that the program trees are written in, the methodology presented in Chapter 5 only makes use of the language when it is enumerating all program trees. However, for our neighbourhood generation algorithm, the language is a required input. We will need to relabel and insert nodes into program trees, and will need a complete library of all terms that are in the language.

Based on these observations, we have created a function prototype in an object-orientated design called NEIGHBOURHOOD-GENERATION, shown in Algorithm 6.1. It contains a constructor, and the function that returns the neighbourhood, GENERATESUCCESSORS.

Algorithm 6.1 NEIGHBOURHOOD-GENERATION Object Prototype

CONSTRUCTOR		
Input:	L	The language. Conceptually a list of pairs and a cost function. Each element in each pair contains a term and a type signature.
	n_{max}	The upper bound on the maximum cost of edits allowed.
GENERATESUCCESSORS		
Input:	p	The input program tree. Each node contains a term from L.
	n	The maximum cost of the edits allowed from p to any output program tree.
Output:		The set of trees that can be created from p using tree edits with a combined cost of at most n. It is represented as a collection. Each element at index $k \in \{0 \dots n\}$ contains a map. Each element of the map contains an integer i mapping to a set of trees. The edit sequences to obtain any of the program trees in the set at (k, i) from p have a combined cost of exactly k using exactly i edits. Each returned edit sequence has the minimum cost of all edit sequences which would create the same tree.

The reader should note that this is an initial function prototype and in Section 6.4.7 we change the function signature of GENERATESUCCESSORS, as additional details about the underlying algorithm become clear.

6.2.1 Mock Language

Throughout the rest of this chapter we will be using examples of program trees to illustrate to the reader how the algorithms described work. We will be using Language EX-1 in those examples. This language was previously introduced in Figure 2.7, and is outlined again in Figure 6.1, where we also show the COST value associated with each term in the language.

The reader may question why we do not use Language A in the examples given in this chapter, as it has been used extensively in previous chapters. Due to the many terms and functions in Language A, we feel that a smaller language with less principle types will aid in the understanding of the algorithms.

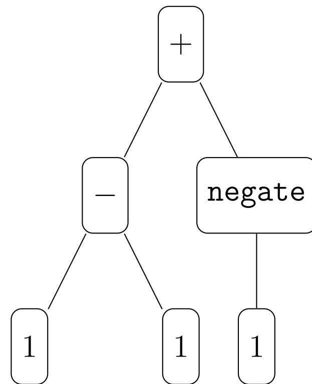
As an introduction to the notation used in this chapter to describe program trees, as well as providing an example of a program tree written in Language EX-1, we

Term	Type Signature							Cost
0							`Int`	1
1							`Int`	1
+			`Int`	→	`Int`	→	`Int`	1
−			`Int`	→	`Int`	→	`Int`	1
`negate`					`Int`	→	`Int`	1
`coinFlip`							`Bool`	1
`intIf`	`Bool`	→	`Int`	→	`Int`	→	`Int`	1
`lessThan`			`Int`	→	`Int`	→	`Bool`	1

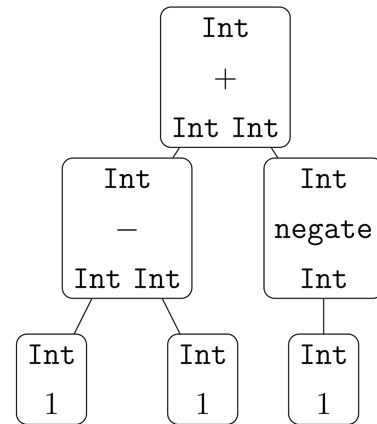
Figure 6.1: The Language EX-1. We use this language to show examples of the neighbourhood generation algorithm described in this chapter. We show each term with its associated type signature and Cost value. The set of principle types in Language EX-1 is {`Bool`, `Int`}.

show two identical program trees in Figure 6.2. The program tree in Figure 6.2a is presented as the other program trees have been in this thesis; that is to say, each node contains a term from the language. In the program tree in Figure 6.2b, each node contains a term and that term's relevant typing information. Above each term t is the return type of t. Below t are the required types of the arguments to t. This visualisation allows us to quickly look at a program tree and see whether it is valid under a language's typing rules; that is to say that it is type safe, or it type checks. We will be using this representation (or variants of it) for the remainder of this chapter, as we believe it aids in the understanding of the algorithms.

The types of language that the neighbourhood generation algorithm described in this chapter can be used with are those explicitly described in Section 2.5.2. Succinctly, it can be used with those languages with a monomorphic type system that prohibit currying.



(a) An example of a program tree written in Language EX-1. Each node in the tree is annotated with a term from the language.



(b) An example of a program tree written in Language EX-1. Each node is annotated with a term, the terms return type (above) and the terms arguments (below). We can see that this program tree type checks.

Figure 6.2: An example program tree under Language EX-1 that type checks. It is shown in two alternate representations. The left shows just the terms in each node. The right shows additional typing information.

6.3 Example Neighbourhood Generation Algorithms

In this section we informally describe two methods of creating the GenerateSuccessors algorithm. The observations made in this section allow us to create an implementation of Neighbourhood-Generation in Section 6.4 and Section 6.6.

In Section 6.3.1 we show some examples of input and output from GenerateSuccessors, and introduce a DSL that is used to formulate sequences of tree edits.

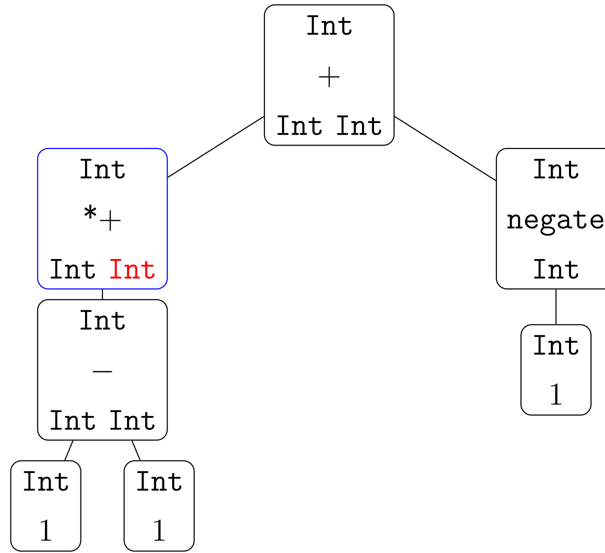
In Section 6.3.2 we informally describe one potential way that GenerateSuccessors could be defined. We term this the "naive method", and show that it is impractical to use as it requires the exploration of a search space that would be intractable for large program trees.

In Section 6.3.3 we show how patterns of terms can reoccur in program trees. It is this key observation that serves as an introduction to how the Neighbourhood-Generation algorithm is implemented in Section 6.4 and Section 6.6.

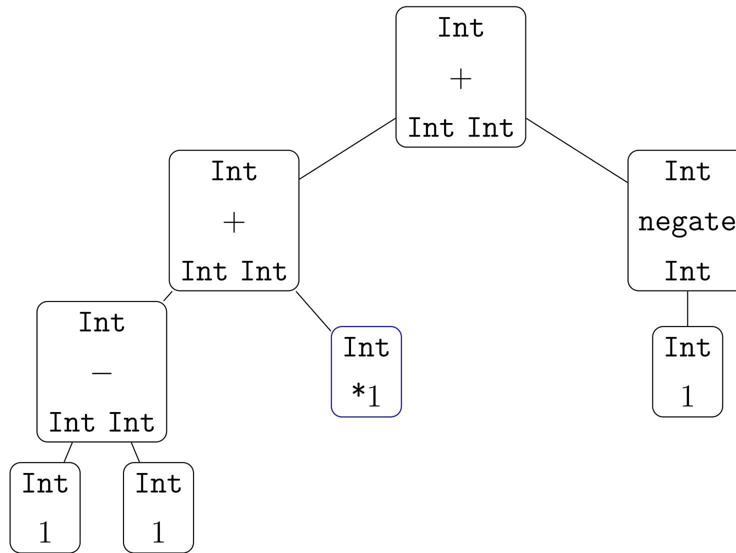
6.3.1 Notation & Example Output

In Figure 6.3 we show two program trees. This figure illustrates the process of taking the program tree in Figure 6.2 and applying 2 edits to it. The program tree after the first edit is shown in Figure 6.3a, and the program tree after the second edit in Figure 6.3b. The program tree in Figure 6.3b is an example of a program tree that would be returned from GenerateSuccessors when given the input program tree in Figure 6.2, and an n value of at least 2. The edits themselves are both insertions of new nodes, which are highlighted with blue borders. The reader can see that after the first insertion, the program tree is not type safe. It is only after the second node is inserted that it becomes type safe.

The input program tree can be seen as the *start state*, and all of the output program trees seen as the *end states*. The sequence of program trees between a start and an end state corresponds to a sequence of edits that identify each change made from one program tree to the next in the sequence. To describe edit sequences, we use the DSL TE-1 in Table 6.1. This language is designed to work on generic trees, rather than program trees under a specific language. Each edit is performed in relation to a current node, or *context* - that is to say, each edit is performed relative to a node in



(a) The program tree after a single node has been inserted into the program tree in Figure 6.2. This program tree does not type check and is considered an intermediary state in this sequence of program trees.



(b) The program tree after a second node has been inserted into the program tree in Figure 6.3a. This program tree type checks and is considered the end state in this sequence of program trees. When the function GenerateSuccessors is given the program tree in Figure 6.2 and an n value of at least 2, this program tree would be an example of output returned.

Figure 6.3: A pair of program trees written in Language EX-1. Together with the program tree in Figure 6.2, they can be viewed as a sequence. Each tree is obtained by inserting a node into the previous. In each tree the inserted node is shown by a blue outline. The node with an asterisk in is the current context.

Table 6.1: The Language TE-1. This DSL is designed to describe tree edits in relation to a current node or context. It is formulated in a generic manner, so that it can be used on generic trees. Sequences of edits can be chained together to perform multiple edits on a tree.

Edit	Arguments	Description
`Insert`	t The node to insert k Insertion position i Number of arguments	Inserts node t underneath the current context c at position k. i subsequent children of c after position k are taken as t's children. t becomes the new context.
`Delete`	k Deletion position	Delete node at index k under the current context c. The sequence of children that the node at k had are inserted where k was under c. The context does not change.
`Relabel`	t The new label k Relabel position b Boolean variable. Used for type-based edits	Relabels the node under the current context c at position k with the label t. t becomes the new context.
`MoveUp`	None	Moves the context to the parent of the current node.
`MoveDown`	k Move position	Moves the context to the kth child of the current context.

the tree. At the beginning of a sequence of edits, the context is set to the root, and it changes as edits are performed. The DSL includes two primitives used for moving through a tree without making any edits; `MoveUp` for moving to a parent node and `MoveDown` for moving to an indexed child node.

We call an edit sequence that operates on terms defined by a language a *term-based* edit sequence. An example of a term-based edit sequence is presented in Figure 6.4. It shows the sequence of tree edits that correspond to the intermediary steps shown in Figure 6.3. In the program trees in Figure 6.3 the node with the current context is shown by an asterisk.

Move Name	t	k	i	b
`Insert`	+	0	1	N/A
`Insert`	1	1	0	N/A

Figure 6.4: A term-based edit sequence. It can be used to transform the program tree in Figure 6.2 into the program tree in Figure 6.3b via the program tree in Figure 6.3a. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

6.3.2 Naive Method

The sequence of program tree states shown in Figure 6.3 illustrates one possible way that GenerateSuccessors could work - that is, by treating the generation of output trees as a search problem. By this we mean, the input program tree is considered the start state, and all possible next steps in the search are found by performing every possible edit on this start state. Every subsequent successor state in turn has every possible edit applied to it, creating a new set of successor states. This process continues until the cost of the sum of edits is at most n. Any valid output program trees (or in this context, end states) that are found are returned.

In this subsection we will analyse this method, and show why it is not viable due to the exponentially large search space. Each one of the edits that can be performed (insert, relabel and delete) are analysed below to calculate the number of successor states that will be created for a generic program tree with k nodes, each of which contains a term from a language that has a total of $|L|$ terms. We will assume in this subsection that each edit has a cost of 1 and therefore, the cost of n edits will be exactly n.

Deletion

The number of successor states for a given program tree that are created when deleting a single node is simple to compute; since there are k nodes, k different deletions can occur, and therefore there are k possible successor states.

Relabel

For a single node in a tree, it can be relabelled to $|L| - 1$ different terms. The -1 comes from the fact that the node cannot be relabelled with itself. Since there are k terms, the number of possible successor states is $k(|L| - 1)$.

Insertion

The number of possible successor states that are created when an insertion occurs is much harder to compute. It is impossible to know the exact number of successor states for a tree with k nodes. However, we can determine an upper bound.

To do so, rather than calculating the number of successor states found from inserting terms from a language L, we will consider the number of successor states that can be found from inserting an unlabelled node. Later we will consider how the language changes the number of successor states.

Let us first consider a node a with m children. We can state that there are the following number of successor states that can be created from inserting a single unlabelled node underneath a:

- $m+1$ insertions of a node with no children. In each successor state a has $m+1$ children.
- m insertions of a node with a single previously existing child. In each successor state a has m children.
- $m-1$ insertions of a node with 2 previously existing children. In each successor state a has $m-1$ children.
- ...
- 2 insertions of a node with $m-1$ previously existing children. In each successor state a has 2 children.
- 1 insertion of a node with all previously existing children. In the successor state a now has 1 child.

Using the arithmetic series, we can state that there are $\frac{1}{2}(m+2)(m+1)$ possible successor states created by inserting an unlabelled node under a.

In general for any tree, inserting a single unlabelled node above the root will create a single successor state. For a tree with l leaves, inserting an unlabelled node underneath any of these leaves will create l successor states.

We know from the arithmetic series and the deductions in the previous paragraph that, given a tree with k nodes, where $k-1$ of these nodes are the children of the root, there are $k+\frac{1}{2}(k+1)k$ possible successor states created by inserting an unlabelled

node. We call this tree configuration c_1. We will now prove that, for an arbitrary tree with k nodes, this is the upper bound on the number of successor states created from inserting an unlabelled node.

Let us begin by considering a tree with k nodes in a different configuration to c_1. We call this configuration c_2. In this tree the root node has $k-s-1$ children, where $0 < s < k-1$. One of the children of the root itself has s children. c_2 can be thought of as being created by splitting c_1's $k-1$ nodes into two sets. The number of successor states found from the insertion of an unlabelled node for c_2 is given by $k-1+\frac{1}{2}(k-s+1)(k-s)+\frac{1}{2}(s+2)(s+1)$.

Below we show that there are more successor states created from the insertion of an unlabelled node in c_1 than there are from the insertion of an unlabelled node in c_2.

$$k+\frac{1}{2}(k+1)k > k-1+\frac{1}{2}(k-s+1)(k-s)+\frac{1}{2}(s+2)(s+1) \tag{6.1}$$

Expand brackets

$$k+\frac{1}{2}(k^2+k) > k-1+\frac{1}{2}(k^2-2ks+s^2+k-s)+\frac{1}{2}(s^2+3s+2) \tag{6.2}$$

Expand brackets

$$k+\frac{k^2}{2}+\frac{k}{2} > k-1+\frac{k^2}{2}-ks+\frac{s^2}{2}+\frac{k}{2}-\frac{s}{2}+\frac{s^2}{2}+\frac{3s}{2}+1 \tag{6.3}$$

Simplify and subtract $k+\frac{k^2}{2}+\frac{k}{2}$ *from both sides*

$$0 > -ks+\frac{s^2}{2}-\frac{s}{2}+\frac{s^2}{2}+\frac{3s}{2} \tag{6.4}$$

Simplify and add ks *to both sides*

$$ks > s^2+s \tag{6.5}$$

Divide both sides by s *then subtract* 1

$$k-1 > s \tag{6.6}$$

We know from our description of c_2 that $k-1 > s$ is always true, therefore we can state that inserting an unlabelled node into c_1 will always create more successor states than inserting an unlabelled node into c_2.

We can take the tree configuration c_1 and split its k nodes into two subsets in the same manner as described for c_2. By repeatedly performing this splitting of nodes, it is possible to describe any tree configuration with k nodes. As we have shown inserting an unlabelled node into a tree split in this way will create less successor states than inserting an unlabelled node into the original tree, it follows that the upper bound on the number of successor states from inserting an unlabelled node in a tree with k nodes is $k+\frac{1}{2}(k+1)k$.

Returning to our original goal, we can state that, for a language L, there are at most $|L|(\frac{1}{2}(k+1)k)+k$ successor states that will be created from the insertion of a node into a tree containing k terms.

Number of States

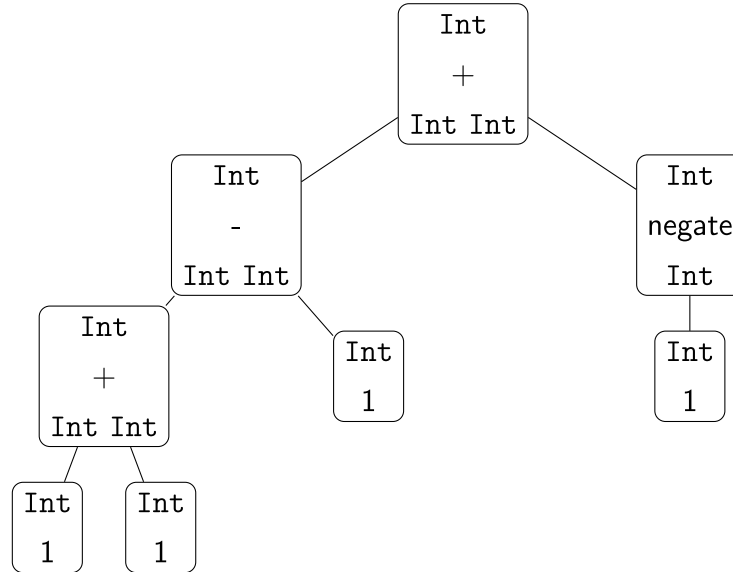
By taking the summation of these results, we can state that there is an absolute maximum of $(|L|(\frac{1}{2}(k+1)k)+k)+k+k(|L|-1)$ successor states for a single edit. It is obvious that insertion makes this quadratic in complexity. We can state that $(|L|(\frac{1}{2}(k+1)k)+k)+k+k(|L|-1) \in \mathcal{O}(k^2|L|)$. For n edits, this means that the number of end states is $\in \mathcal{O}(k^{2n}|L|^n)$.

We feel that this provides enough evidence that this method is impractical for large program trees, or program trees written under languages with many terms. Attempts were made to construct an algorithm using this methodology, but it quickly became apparent that it was infeasible as it took several minutes to compute results for a single program tree. For this reason, we focused our efforts on alternative methodologies for creating the NEIGHBOURHOOD-GENERATION algorithm.

6.3.3 Identifying Common Patterns

In this subsection we illustrate to the reader an alternate methodology for generating a program tree's neighbourhood. Let us consider Figure 6.5, which shows a term-based edit sequence and a program tree. The edit sequence in Figure 6.5b can be applied to the program tree in Figure 6.2 to obtain the program tree shown in Figure 6.5a.

This program tree and edit sequence are vitally important for understanding the methodology used in this chapter for generating a program tree's neighbourhood. The reader can see that the edit sequences in Figures 6.4 and 6.5b are nearly identical. The only difference is that in the edit sequence in Figure 6.5b, the context is moved downwards initially. The edits themselves can be seen as being performed on the same "core" part of a program tree; that is to say, at the point where the first insertion is performed, one could interpret the contexts as being the same. They are both at a point where their current node requires two `Int` arguments and has two children, both of which return an `Int` argument. A visualisation of this pattern can be seen in Figure 6.6a, which can be considered the start state. By applying the edit sequence in Figure 6.4, we can produce the intermediary state and end state program trees in Figures 6.6b and 6.6c.

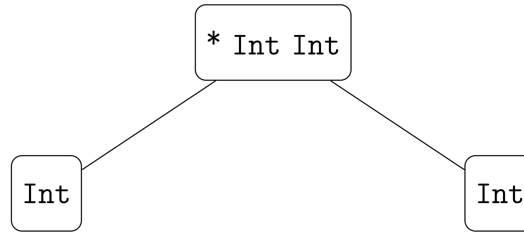


(a) A program tree written in Language EX-1. When the function GENERATESUCCESSORS is given the program tree in Figure 6.2 and an n value of at least 2, this program tree would be an example of output returned.

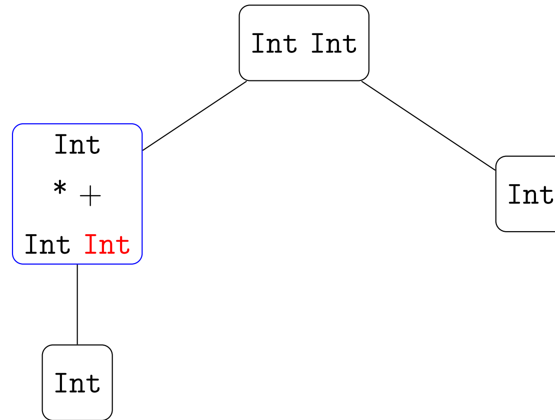
Move Name	t	k	i	b
`MoveDown`	N/A	0	N/A	N/A
`Insert`	+	0	1	N/A
`Insert`	1	1	0	N/A

(b) A term-based edit sequence. It can be used to transform the program tree in Figure 6.2 into the program tree in Figure 6.5a. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

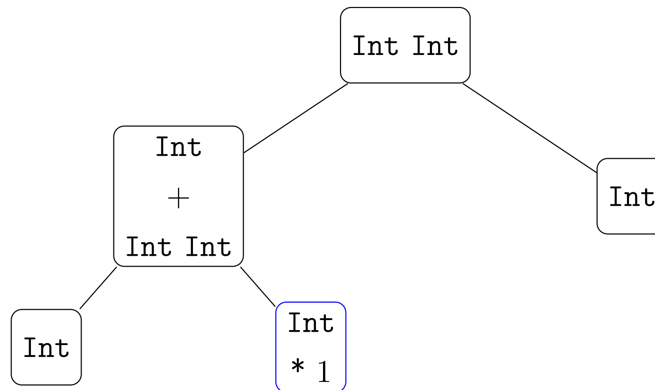
Figure 6.5: An example of a program tree and a term-based edit sequence. The edit sequence can be used to transform the program tree in Figure 6.2 into the program tree in Figure 6.5a.



(a) A tree representing a pattern of nodes in the program tree in Figure 6.2. The pattern type checks and is considered the start state in this sequence of trees.



(b) The tree representing the pattern in Figure 6.6a after a single node has been inserted into it. This pattern does not type check and is considered an intermediary state in this sequence of trees.



(c) The tree representing the pattern in Figure 6.6b after a second node has been inserted into it. This pattern type checks and is considered the end state in this sequence of trees.

Figure 6.6: Three trees, each an abstract representation of a part of a program tree written in Language EX-1. They can be viewed as a sequence. Each tree is obtained by inserting a node into the previous tree. The edit sequence in Figure 6.4 can be used to transform the start state tree into the end state tree. In each tree the inserted node is shown by a blue outline. The node with an asterisk in is the current context.

This use of patterns illustrates one way that the NEIGHBOURHOOD-GENERATION algorithm may work. Rather than directly searching an input program tree, patterns are identified. A search is conducted on these patterns for valid sequences of edits that lead to type safe, end state patterns. The edit sequences can then be directly applied to the original program tree, to produce new program trees. This has the major advantage of allowing previously found results to be reused, which should reduce the amount of search required considerably.

For a language L and maximum score n_{max}, we can generate all possible start state patterns that could appear in any type safe program tree under that language. We can search these start state patterns to find valid end state patterns that can be moved to using edits with a cost of at most n_{max}. When input program trees are given to the algorithm, and these patterns recognised in the input program tree, the corresponding edit sequences can be applied to produce output program trees. This is the method we use in this chapter to create NEIGHBOURHOOD-GENERATION; the generation and search of the patterns occurs in the constructor, and the identification of a pattern and the application of a pattern's corresponding edit sequences performed in the function GENERATESUCCESSORS. In the following three sections, we give details as to how these ideas are formalised, and provide pseudocode showing how the constructor and GENERATESUCCESSORS are defined.

6.4 Defining Generate Successors

In Section 6.3.3 we provided an example of a pattern that could be extrapolated from two separate parts of the same program tree. We showed how this pattern could have tree edits applied to it, creating an end state pattern that was type safe. These patterns could be recognised in an input program tree, and their associated edits applied to produce type safe output program trees without any additional computation being required - essentially allowing the edit sequences to be reused. We stated that this is the way in which the NEIGHBOURHOOD-GENERATION algorithm that is described in this chapter will work. We also stated that the search for patterns and edit sequences would be conducted in the constructor, and the application of the edit sequences in the function GENERATESUCCESSORS. In this section, we show how GENERATESUCCESSORS is defined.

This section's contents can be described as follows; in Section 6.4.1 we formalise

the patterns that we recognise in a program tree. In Section 6.4.2 we describe an abstraction of these patterns that allows multiple patterns to be contained in a single structure, and show how this extends to edit sequences. In Section 6.4.3 we discuss the maximum size of the patterns that we will be required to recognise and search, which is based off the maximum cost of edits allowed. In Section 6.4.4 we describe an algorithm to find all the patterns in a given input program tree. In Section 6.4.5 we identify an edge case of our pattern identification strategy, and show how to augment the input program tree to accommodate it. In Section 6.4.6 we show how edit sequences are applied to a given input program tree. Finally, in Section 6.4.7 we finish off defining GenerateSuccessors, and show how, to create the correct set of output program trees, the algorithm may have to identify multiple patterns and apply multiple edit sequences.

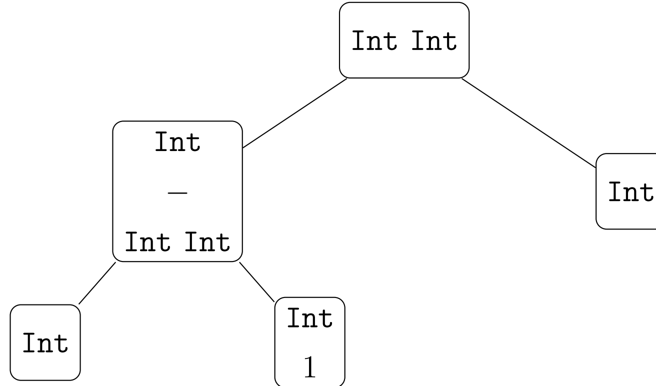
6.4.1 Formalising Patterns

In the previous section, we informally described patterns contained within a program tree. We call these abstract representations, or patterns, partially-typed partial-program trees. We define them as follows:

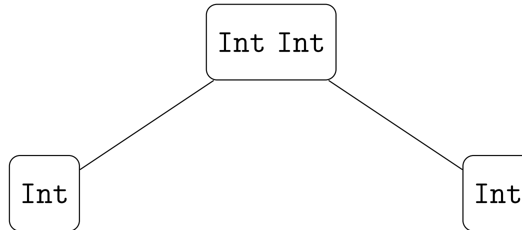
Definition 24 (Partially-Typed Partial-Program Tree)
A partially-typed partial-program tree (PTPPT) under a language L is a tree with the following properties. The root of the tree must contain a non-empty vector of principle types from L. All child nodes of the root are rootless PTPPTs (RPTPPTs). An RPTPPT is a tree where each node is either a named node or a typed node. A named node contains a term from L. A typed node contains a single principled type from L. Typed nodes cannot have child nodes.

Using this definition of a PTPPT, we can see that the trees in Figure 6.6 are PTPPTs. Figure 6.6a is a start state PTPPT, and Figure 6.6c is an end state PTPPT. We can also see that PTPPTs can be type checked, in a similar manner to how type checking is performed on full program trees. The PTPPTs in Figures 6.6a and 6.6c type check, and the intermediary PTPPT in Figure 6.6b does not.

To be clear to the reader, start state PTPPTs can contain named nodes, not just typed nodes. Figure 6.7 shows an example of a start state and an end state PTPPT, together with an edit sequence to move between them. We can see that the start state contains a named node. An example of this pattern applied to the program tree in Figure 6.2 is shown in Figure 6.8.



(a) A PTPPT representing a pattern of nodes in the program tree shown in Figure 6.2. This PTPPT type checks and is considered the start state in this sequence of PTPPTs.



(b) The PTPPT after two nodes have been deleted from the PTPPT shown in Figure 6.7a. This PTPPT type checks and is considered the end state in this sequence of PTPPTs.

Move Name	t	k	i	b
`Delete`	N/A	0	N/A	N/A
`Delete`	N/A	1	N/A	N/A

(c) A term-based edit sequence. It can be used to transform the PTPPT in Figure 6.7a into the PTPPT in Figure 6.7b. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

Figure 6.7: A start state PTPPT, end state PTPPT and edit sequence. The edit sequence can be used to transform the start state into the end state. The start state PTPPT mirrors a configuration of nodes in the program tree shown in Figure 6.2.

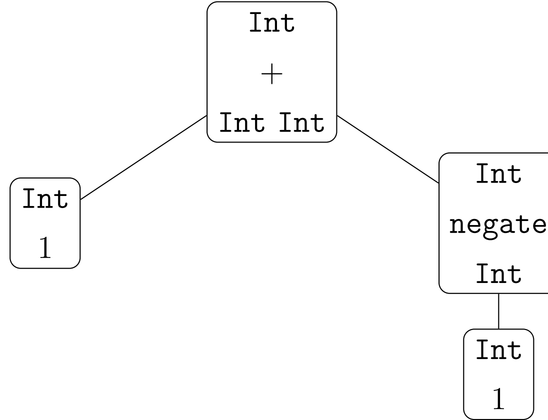
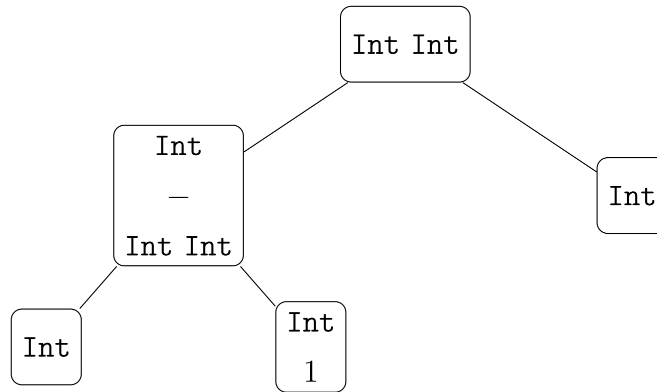


Figure 6.8: A program tree written in Language EX-1. When the function GenerateSuccessors is given the program tree in Figure 6.2 and an n value of at least 2, this program tree would be an example of output returned. It can be obtained by taking the program tree in Figure 6.2, moving the context to the left subtree of the root, and then applying the edit sequence in Figure 6.7c.

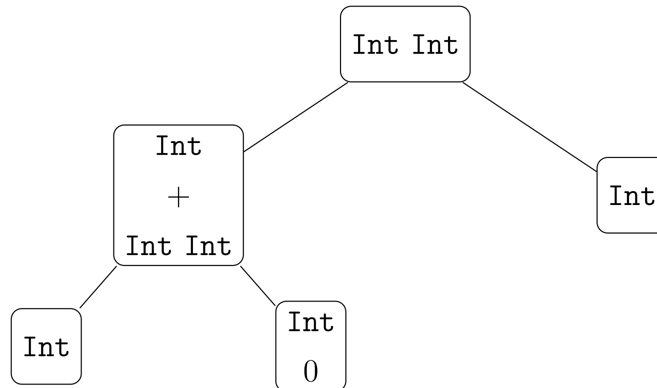
6.4.2 Abstracting Patterns

The reader may assume that we would now present the algorithms showing how to identify PTPPTs in program trees, and how to apply their associated edit sequences to create new program trees. However, our implementation of NeighbourhoodGeneration does not work like this. Instead of using PTPPTs and term-based edit sequences, we use abstractions of these constructs. These abstractions are used because they reduce the complexity of the search for valid edit sequences, performed in the constructor. Further detail of this is provided in Section 6.6.

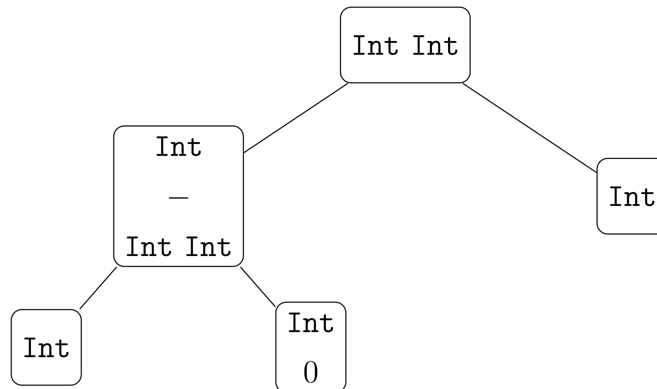
To begin, we will show the intuition behind the abstractions. Consider the three end state PTPPTs in Figure 6.9 that can be obtained from the PTPPT in Figure 6.6a using edits with a cost of 2. Together with the PTPPT in Figure 6.6c, we can observe that these four PTPPTs are similar to each other. They have all had nodes inserted in the same positions in their original trees, and the nodes inserted had identical type signatures. Specifically, the first element inserted had a type signature of `Int` $\rightarrow$ `Int`$\rightarrow$ `Int` and the second a type signature of `Int`. This is made clearer by the term-based edit sequences shown in Figure 6.10 for the PTPPTs in Figure 6.9, together with the edit sequence shown in Figure 6.4 for the PTPPT in Figure 6.6c.



(a) A PTPPT constructed by applying the edit sequence in Figure 6.10a to the PTPPT in Figure 6.6a. This PTPPT type checks and is considered an end state PTPPT.



(b) A PTPPT constructed by applying the edit sequence in Figure 6.10b to the PTPPT in Figure 6.6a. This PTPPT type checks and is considered an end state PTPPT.



(c) A PTPPT constructed by applying the edit sequence in Figure 6.10c to the PTPPT in Figure 6.6a. This PTPPT type checks and is considered an end state PTPPT.

Figure 6.9: Three PTPPTs that can be constructed by applying the edit sequences in Figure 6.10 to the PTPPT in Figure 6.6a.

Move Name	t	k	i	b
`Insert`	$-$	0	1	N/A
`Insert`	1	1	0	N/A

(a) A term-based edit sequence. It can be used to transform the PTPPT in Figure 6.6a into the PTPPT in Figure 6.9a.

Move Name	t	k	i	b
`Insert`	$+$	0	1	N/A
`Insert`	0	1	0	N/A

(b) A term-based edit sequence. It can be used to transform the PTPPT in Figure 6.6a into the PTPPT in Figure 6.9b.

Move Name	t	k	i	b
`Insert`	$-$	0	1	N/A
`Insert`	0	1	0	N/A

(c) A term-based edit sequence. It can be used to transform the PTPPT in Figure 6.6a into the PTPPT in Figure 6.9c.

Figure 6.10: Three term-based edit sequences that can be used to transform the PTPPT in Figure 6.6a into the PTPPTs shown in Figure 6.9. They are described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

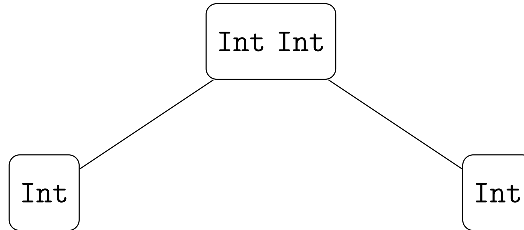
We now introduce abstractions of the PTPPT and term-based edit sequence called the typed partial-program tree and type-based edit sequence. The typed partial-program tree is defined as follows:

Definition 25 (Typed Partial-Program Tree)
A typed partial-program tree (TPPT) under a language L is a tree with the following properties. The root of the tree must contain a non-empty vector of principle types from L. All child nodes of the root are rootless TPPTs (RTPPTs). An RTPPT is a tree where each node is either a typed-named node or a typed node. A typed-named node contains a pair of type signature and cost that belongs to at least one of the terms in L. A typed node contains a single principled type from L. Typed nodes cannot have child nodes. The cost of an RTPPT is the sum of the costs of all the typed-named nodes contained within. The cost of a TPPT is the sum of the costs of all child RTPPTs.

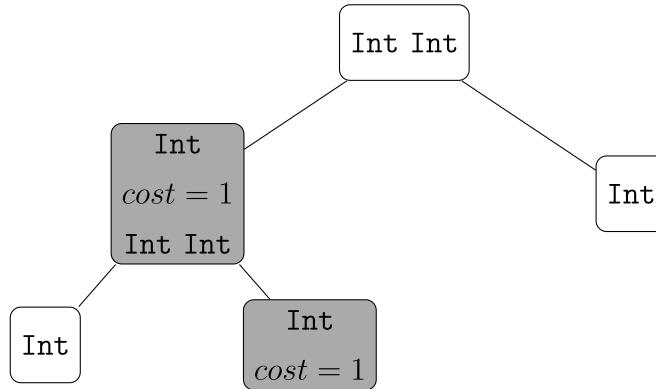
A type-based edit sequence is an edit sequence defined in terms of the DSL EX-1 in Table 6.1. It inserts, deletes and relabels nodes that contain a pair of type signature and cost. This pairing of type signature and cost must belong to at least one of the terms in the language L. A type-based edit sequence can be applied to a TPPT to produce a new TPPT.

In Figure 6.11 we show two examples of TPPTs, and a type-based edit sequence accompanying them. The edit sequence in Figure 6.11c can be applied to the TPPT in Figure 6.11a to produce the TPPT in Figure 6.11b. These TPPTs and edit sequence capture the essence of the PTPPTs shown in Figures 6.6c and 6.9 and term-based edit sequences in Figures 6.4 and 6.10. They highlight the basis of our abstractions; the TPPT represents multiple PTPPTs with an identical structure. The nodes in these PTPPTs contain terms which have identical type signatures and costs. In a similar way, type-based edit sequences can be thought of as representing multiple term-based edit sequences. To be clear to the reader, any TPPT represents at minimum a single PTPPT, and any type-based edit sequence represents at minimum a single term-based edit sequence.

The reader may note that the distinction between the different nodes in a TPPT is more fine-grained than in a PTPPT. In a TPPT, a typed-named node and a typed node can have similar labelling. This is because there are principle types that are also the type signatures of terms in the language. Language EX-1 has a principle type of `Int`, but also has terms with that type signature. Figure 6.11b shows an example of



(a) A TPPT. This TPPT is representative of a single PTPPT shown in Figure 6.6a, itself a representation of a pattern of nodes identified in the program tree shown in Figure 6.2. This TPPT type checks and is considered the start state in this sequence of TPPTs.



(b) The TPPT after two nodes have been inserted into the TPPT shown in Figure 6.11a. It is representative of the PTPPTs shown in Figures 6.6c and 6.9. This TPPT type checks and is considered the end state in this sequence of TPPTs.

Move Name	t	k	i	b
`Insert`	(`Int` $\rightarrow$ `Int` $\rightarrow$ `Int`, 1)	1	0	N/A
`Insert`	(`Int`, 1)	0	0	N/A

(c) A type-based edit sequence. It can be used to transform the TPPT in Figure 6.11a into the TPPT in Figure 6.11b. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

Figure 6.11: A start state TPPT, end state TPPT and edit sequence to transform the start state into the end state. The start state and end state can be said to be abstractions of PTPPTs.

this. To make the distinction between the two clear, we colour typed-named nodes grey, and typed nodes white.

Through these abstractions, the way in which the NEIGHBOURHOOD-GENERATION algorithm works changes slightly; it recognises TPPTs and applies type-based edit sequences. Since TPPTs do not mirror an input program tree directly, some additional processing is required. This is also true of the type-based edit sequences. We still assume that, in the constructor, the complete set of all patterns and associated edit sequences are generated, but these patterns are now the TPPT and the term-based edit sequence.

6.4.3 Size of TPPTs

Though we have identified what a TPPT is, and by extension what needs to be recognised in an input program tree, we have not discussed any limitation on the size of the TPPTs that need to be recognised. Since the maximum cost of the edits we can make is bounded, the maximum size of any relevant TPPT is too. Yet, the definition of a TPPT places no limit on their size. Theoretically, if the TPPTs we were searching for could be as big as the input program tree, there would not be much difference between our algorithm and the naive method outlined in Section 6.3.2.

Consider an invocation of the neighbourhood generation algorithm that allows edits with a cost of up to 2. An example edit sequence under Language EX-1 could consist of 2 insertions, each with a cost of 1. A start state TPPT that had this edit sequence applied to it would not need to identify any typed-named nodes, as there are no deletions or relabels in the edit sequence. Similarly, an edit sequence consisting of 2 relabels (or deletions), each with a cost of 1, would definitively require exactly 2 typed-named nodes with a cost of 1 to work from.

In general, for invocations of GENERATESUCCESSORS that allow edits with a combined cost of at most n, all TPPTs with a cost of at most n are required that can be found in the input program tree. It would depend on the set of start state TPPTs and edit sequences found in the constructor as to whether all of these would be used to create output program trees, however since it is possible that any of them may be start state TPPTs known to have an associated type-based edit sequence, they must all be identified.

As to how many TPPTs there are in a program tree with l nodes that have a cost of n, it is impossible to answer definitively, as we would need to know the exact

configuration of the tree to answer this. However, we can provide some information about how many TPPTs there are for a single node with k children, where every node has a cost of 1. Assuming that each node has no children, the number of TPPTs that can be identified is described by Equation (6.7).

$$\sum_{i=0}^{n} \binom{k}{i} \tag{6.7}$$

It shows the number of different combinations of nodes that can be created by picking from between 0 and n of the k nodes to be the typed-named nodes in a TPPT. For any configuration where any of the k children have children themselves, this number would increase further. In the next subsection, we show the algorithm to find all the TPPTs that have a cost of at most n.

6.4.4 Finding TPPTs

In this section we show the algorithms that allow us to identify the TPPTs in an input program tree that have a cost of at most n. To do this, the tree is traversed and the algorithm FIND-TPPTs called on every node, which identifies the TPPTs from that node and its children. FIND-TPPTs itself consists of two parts; the first constructs the RTPPT portions of all TPPTs. The second part constructs the root for all the RTPPTs to create the set of TPPTs. When a TPPT has been identified, the algorithm notes the position of where it has been found in the input program tree, so that the type-based edit sequences can be applied at that point. In Section 6.4.6 we show the algorithm that allows type-based edit sequences to be applied to the input program tree to produce the output program trees. To be clear to the reader, the TPPT at this point has no use other than for identification of a pattern.

Algorithm 6.2 contains the first component of the overarching algorithm to find TPPTs called FIND-RTPPTs. It takes two arguments; a cost value n and the current node in the input program tree p. It identifies all relevant sequences of RTPPTs that represent the children of p which have a cost of at most n. It returns a vector of collections of sequences of RTPPTs. Each vector element's index represents the sum of the cost of every sequence of RTPPTs contained within that element's collection.

FIND-RTPPTs works in two parts; a recursive part that calls itself on each child c of p, and a second part that builds every relevant sequence of RTPPTs using the recursive results. If p has no children, then the function returns an empty collection.

Algorithm 6.2 FIND-RTPPTS

Input: n The maximum cost of the typed-named nodes allowed in the output.
p A node in the input program tree.

Output: A vector of collections of sequences of RTPPTs. Each vector element's index i is in the range $\{0 \dots n\}$, representing the total cost of all the sequences of RTPPTs contained in that element's results.

algorithm FIND-RTPPTS(n, p)
    $results = []$
    if (p.CHILDREN().SIZE() $= 0$) **then** $results[0] = []$
    $nextLevel = []$
    for ($i \in \{0 \dots p$.CHILDREN().SIZE() $- 1\}$) **do**
        $c = p$.CHILDREN()[i]
        $nodeCost$ = COST(c)
        $recResults$ = FIND-RTPPTS($n - nodeCost$, c)
        $nextLevel[0][i]$ = [GET-RETURN-TYPE(c)]
        for ($n_{local} \in \{0 \dots n - nodeCost\}$) **do**
            for ($result \in recResults[n_{local}]$) **do**
                $tree$ = MAKE-TREE(c, $result$)
                $nextLevel[nodeCost + n_{local}][i]$.APPEND($tree$)
    for ($n_{local} \in \{0 \dots n\}$) **do**
        $results[n_{local}]$ = ALL-COMBINATIONS(n_{local}, $nextLevel$)
    return $results$

In the first part, for each child c of p, the algorithm works as follows; it calls FIND-RTPPTS recursively on c with an n value of $n-\text{COST}(c)$. This returns all possible sequences of RTPPTs that can be found underneath c which have a combined cost of at most $n-\text{COST}(c)$. It then takes each sequence of RTPPTs s and creates an RTPPT with a typed-named representative of c as its root with s as its children. This creates a collection of RTPPTs with a cost of at most n. Finally, the algorithm adds one additional result; an RTPPT containing the typed node variant of c with no children.

This process can be confusing, so we will show an example. Consider the input program tree in Figure 6.2, with the context set at the root and an n value of 2. For each of the 2 children c, for values $0 \ldots n$, there are a set of RTPPTs which are created from combining the typed-named node representation of c with the recursive results. These are shown in Figure 6.12. The reader may note the $n_i = 0$ case, where no recursive results are used and each subtree is represented by a single typed node.

The second part to FIND-RTPPTS takes these results, and, using only one result from each child of p, builds all possible sequences of RTPPTs containing typed-named nodes with a cost of at most n. That is to say, each sequence of RTPPTs contains only one recursive result from each child, creating a sequence of RTPPTs that mirrors the children under p.

For example, consider the recursive results in Figure 6.12 and $n = 2$. The sequences of RTPPTs returned would have typed-named nodes with a combined cost of at most n. In Table 6.2 we show the various valid configurations of sequences of RTPPTs that would be returned.

The reader may note that the pseudocode for FIND-RTPPTS contains several functions that we have not defined. These can be described as follows:

- GET-RETURN-TYPE. Given a type signature and cost, finds the return type of the type signature.
- MAKE-TREE. Given a label t and a vector of trees vs, creates a tree with the root having a label of t and the vector vs as its children.
- APPEND. Given a vector and an element, appends the element to the vector.

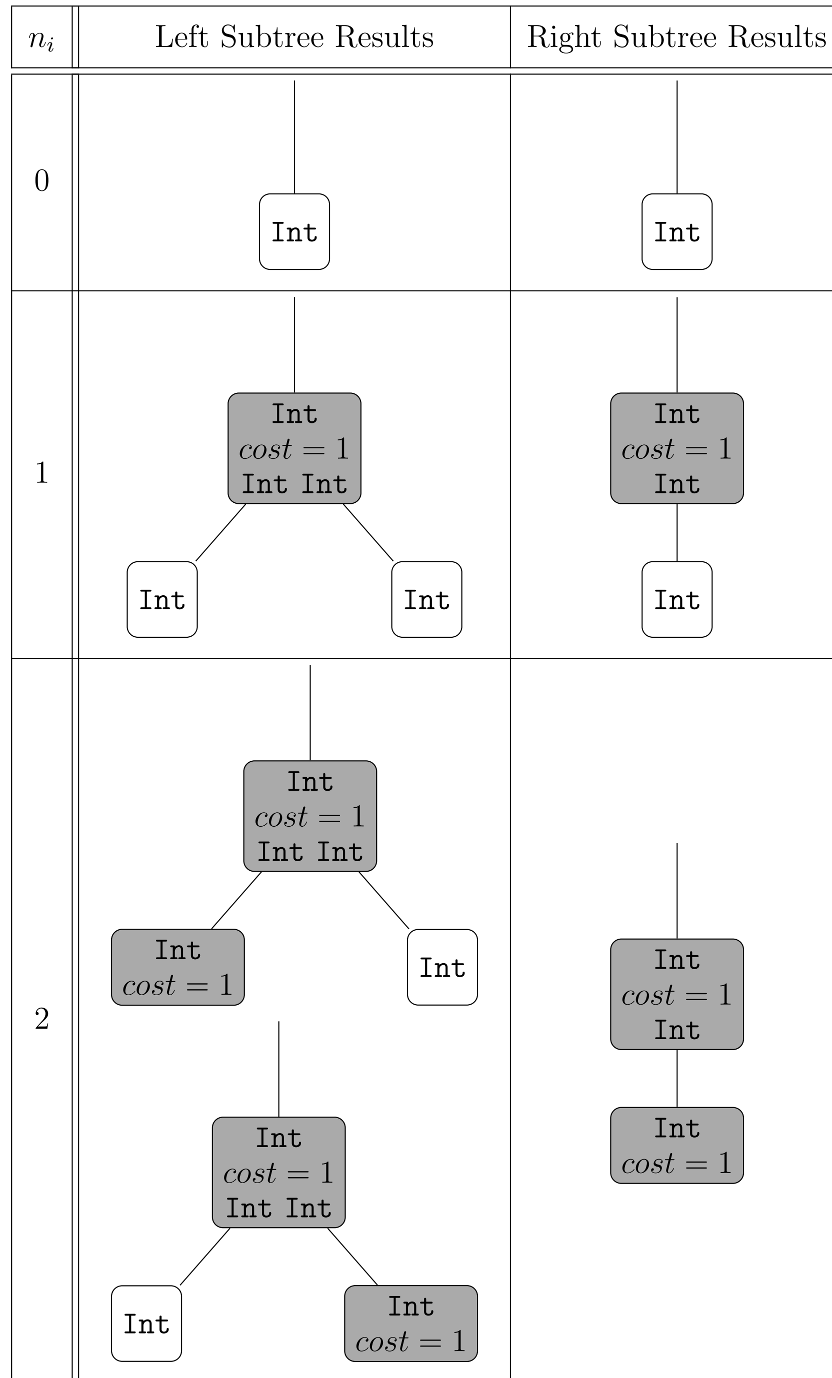


Figure 6.12: The set of results stored in the *nextLevel* variable in the FIND-RTPPTS algorithm when given a context set at the root of the program tree in Figure 6.2, and an n value of 2. For each $n_i \in \{0 \ldots n\}$, and for each child, there are a set of results. We refer to each result using either L or R, and two numbers. For example, L-2-2 refers to the second result of the left subtree's results at $n_i = 2$, while L-2-1 refers to the first.

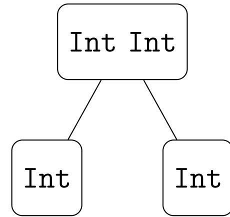
Table 6.2: All the collections of sequences of RTPPTs created from FIND-RTPPTS when given a context set at the root of the program tree in Figure 6.2, and an n value of 2. For each $n_i \in \{0 \dots n\}$ we show the collection of sequences that correspond to that n_i value. These results are extrapolated from the RTPPTs shown in Figure 6.12, which also introduces the shorthand used.

n_i	Collection of sequences of RTPPTs whose COST $= n_i$
0	[[L-0-1,R-0-1]]
1	[[L-1-1,R-0-1], [L-0-1,R-1-0]]
2	[[L-2-1,R-0-1], [L-2-2,R-0-1], [L-0-1,R-2-1], [L-1-1,R-1-1]]

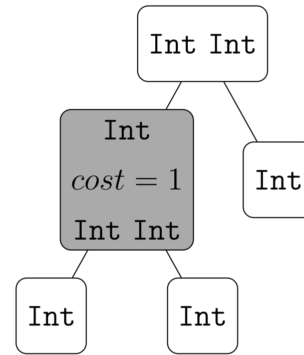
- ALL-COMBINATIONS. Takes as input a cost value n_{cost} and a two-dimensional vector of RTPPTs, where the outer index i is in the range $\{0 \dots n\}$ and the inner index j references the jth child of the input tree p in the range $\{0 \dots p.\text{CHILDREN}().\text{SIZE}() - 1\}$. This function builds a vector of sequences of RTPPTs which all have the following properties; they are of exactly size $p.\text{CHILDREN}().\text{SIZE}() - 1$, the sum of their RTPPTs is exactly n_{cost}, and they have been constructed by taking exactly one RTPPT from each child $c \in p$'s recursive results.

The second stage of the overarching algorithm FIND-TPPTS is much simpler. It takes each RTPPT $treeVect$ from calling FIND-RTPPTS on p's children, and calculates the root for each, building a collection of TPPTs. The pseudocode for FIND-TPPTS is shown in Algorithm 6.3. The 7 TPPTs that are generated from the RTPPTs presented in Figure 6.12 are shown in Figure 6.13.

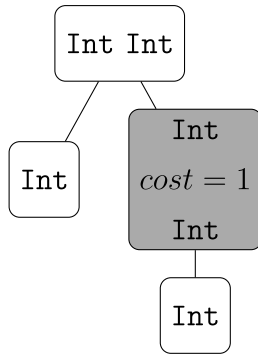
In practice, there are many ways in which we can make these algorithms more efficient; we can call FIND-RTPPTS once on every node in the tree, and memoize the results. We can also reduce the required computation time when generating all combinations of results in FIND-RTPPTS by keeping a local copy of the data structure, and only changing one element at a time. However, for ease of understanding, we have attempted to keep the explanation of the algorithms relatively simple and omit these additional optimisations in our explanations.



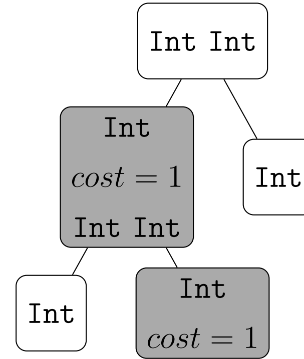
(a) A TPPT with a cost of 0.



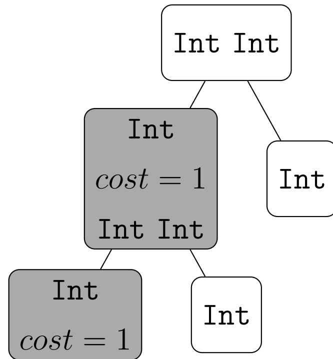
(b) A TPPT with a cost of 1. It has been created from the subtrees L-1-1 and R-0-1 in Figure 6.12.



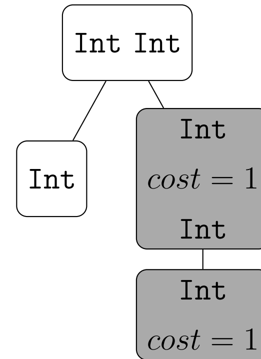
(c) A TPPT with a cost of 1. It has been created from the subtrees L-0-1 and R-1-1 in Figure 6.12.



(d) A TPPT with a cost of 2. It has been created from the subtrees L-2-1 and R-0-1 in Figure 6.12.



(e) A TPPT with a cost of 2. It has been created from the subtrees L-2-2 and R-0-1 in Figure 6.12.



(f) A TPPT with a cost of 2. It has been created from the subtrees L-0-1 and R-2-1 in Figure 6.12.

Figure 6.13: The set of TPPTs returned from the FIND-RTPPTS algorithm when given a context set at the root of the program tree in Figure 6.2, and an n value of 2.

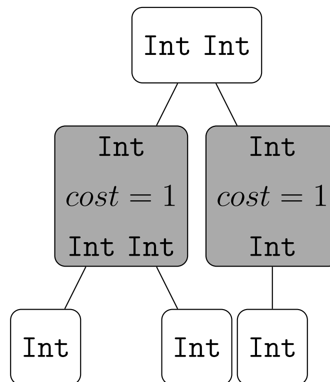
Algorithm 6.3 FIND-TPPTS

Input: n The maximum cost of the TPPTs returned.

p A node in the input program tree.

Output: A collection of TPPTs. Each returned TPPT has a cost of at most n.

```
algorithm FIND-TPPTS(n, p)
    vects = FIND-RTPPTS(n, p)
    retVal = []
    for (set ∈ vects) do
        for (treeVect ∈ set) do
            rootNode = []                        ▷ The vector of types used as the root.
            for (tree ∈ treeVect) do
                returnType = GET-RETURN-TYPE(tree)
                rootNode.APPEND(returnType)
            tree = MAKE-TREE(rootNode, treeVect)
            retVal.APPEND(tree)
    return retVal
```



(g) A TPPT with a cost of 2. It has been created from the subtrees L-1-1 and R-1-1 in Figure 6.12.

Figure 6.13: The set of TPPTs returned from the FIND-RTPPTS algorithm when given a context set at the root of the program tree in Figure 6.2, and an n value of 2. (Continued)

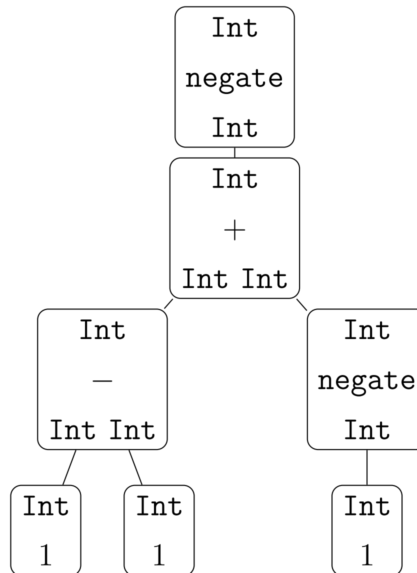


Figure 6.14: A program tree written in Language EX-1. When the function GENERATESUCCESSORS is given the program tree in Figure 6.2 and an n value of at least 1, this program tree would be an example of output returned.

6.4.5 Pre-processing

In Figure 6.14 we show a program tree that serves as an example of output that should be obtained from GENERATESUCCESSORS when given the program tree in Figure 6.2 and an n value of at least 1. The start state and end state TPPTs that represent the pattern and edit sequence used to obtain this output tree are shown in Figure 6.15.

Using the algorithm FIND-TPPTS in the previous section, we would not be able to identify a TPPT that allows us to create this output. It is not possible because in the output program tree the new node is inserted *above* the root of the input program tree. To recognise a TPPT, its root node must be representative of a currently existing node. As there is no node above the root, the start state TPPT cannot be identified and the edit sequence cannot be applied.

To allow us to identify this TPPT, the input program tree requires some additional pre-processing. Specifically, a new root node is set, making the previous root node its only child. The term in this node is a new placeholder term we call the *identity function*. It has a type signature that takes the return type of the old root, and returns the same type. In the case of the program tree in Figure 6.2, this is `Int` →

(a) A TPPT. This TPPT is representative of a pattern of nodes in Figure 6.2. It type checks and is considered the start state in this sequence of TPPTs.

(b) The TPPT after a node has been inserted into the TPPT shown in Figure 6.15a. It type checks and is considered the end state in this sequence of TPPTs.

Figure 6.15: A start state and end state TPPT. The start state is representative of a pattern of nodes above the root in Figure 6.2.

`Int`. An example of the program tree in Figure 6.2 with this pre-processing step applied to it is shown in Figure 6.16.

We stipulate that this new root cannot be deleted or relabelled. All output program trees have this new root node removed from them.

6.4.6 Applying Edit Sequences

After a TPPT has been identified in a program tree, we can discard it. We only need to use the type-based edit sequences that are associated with the TPPT. It is these that are applied to the input program tree to create the output set of program trees.

The methodology behind the application of an edit sequence is relatively simple; by analysing the language and each type-based edit, we can extrapolate all term-based edits from that single edit. For example, a typed-based edit which inserts a term in the form ($\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$, 1) can be instantiated in two possible ways as a term-based edit under Language EX-1. To make this process easier to understand, it is advantageous to consider an alternate representation of the language.

In Figure 6.17 we show the *type-compressed* form of Language EX-1. A type-compressed form of a language groups terms with identical type signatures and costs together in sets. Each element in each set is also given a numerical identifier.

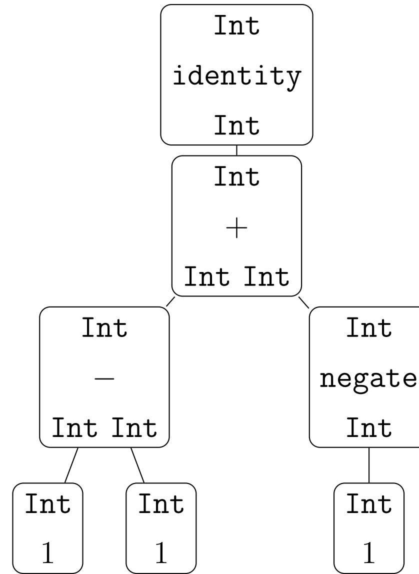


Figure 6.16: The program tree shown in Figure 6.2 after it has been pre-processed. This pre-processing step involves adding an additional node that becomes the new root of the tree. The new node is removed from all output program trees.

Terms	Cardinality	Type Signature	Cost
$\{\ 0_0,\ 1_1\ \}$	2	Int	1
$\{\ +_0, -_1\}$	2	Int → Int → Int	1
$\{\texttt{negate}_0\}$	1	Int → Int	1
$\{\ \texttt{coinFlip}_0\}$	1	Bool	1
$\{\texttt{intIf}_0\}$	1	Bool → Int → Int → Int	1
$\{\texttt{lessThan}_0\}$	1	Int → Int → Bool	1

Figure 6.17: Language EX-1 in its type-compressed form. Terms with the same type signature and cost are grouped together in sets. We also show the cardinality of each set.

The algorithm CREATE-OUTPUT-TREES creates the set of output trees from a given edit sequence. It takes as arguments a type-based edit sequence *seq* and a context within a program tree *p*. Its pseudocode is shown in Algorithm 6.4. It works as follows; upon initialisation, a copy of *p* is created and put into the collection *progs*. For each type-based edit in *seq*, a set of new output program trees is created from the previous set *progs*, and the previous set discarded. This continues, increasing the number of trees in *progs* as more edits are consumed, until the sequence is finished. The elements in *progs* are returned as the output program trees.

CREATE-OUTPUT-TREES makes use of some additional functions which we do not provide pseudocode for. Those not described in previous sections are given as follows:

- CREATE-COPY. This function creates a copy of a program tree and context.
- INSERT-NODE. Inserts a node into a tree at that tree's context.
- DELETE-NODE. Deletes a node from a tree at that tree's context.

We have purposefully not included the pseudocode for the `Relabel` edit in CREATE-OUTPUT-TREES. This is because there is a subtle issue regarding the relabelling of nodes, which is addressed below.

Relabelling

In Figure 6.18 we show a start state and end state PTPPT, together with the term-based edit sequence to obtain the end state PTPPT from the start state PTPPT. In Figure 6.19 we show a start state and end state TPPT, together with the type-based edit sequence to obtain the end state TPPT from the start state TPPT. These examples are designed to mirror each other; one uses PTPPTs and a term-based edit sequence, while the other represents these constructs as TPPTs and a type-based edit sequence.

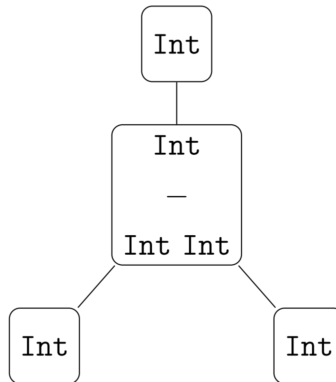
A naive approach to processing a `Relabel` edit could be to use the same method as when processing an `Insert` edit. That is to say, the set of terms *ts* that match the pair of type signature and cost are generated, and then the successor trees constructed from relabelling the original term with elements in *ts*. Using this approach, a set of term-based edit sequences obtained from the type-based edit sequence in Figure 6.19c are shown in Figure 6.20. However, there is a glaring error in the term-based edit

Algorithm 6.4 CREATE-OUTPUT-TREES

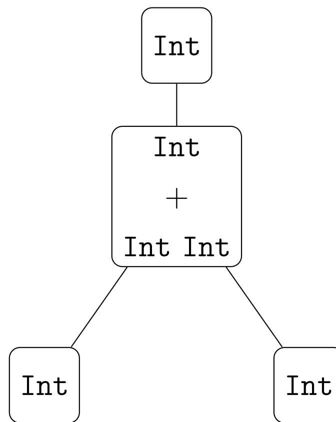
Input: *seq* A type-based edit sequence.
p A program tree and a context. The context denotes where *seq*'s associated TPPT was recognised.
Output: The set of output program trees created.

```
algorithm CREATE-OUTPUT-TREES(seq, p)
    progs = [CREATE-COPY(p)]                              ▷ The original program tree.
    for (edit ∈ seq) do                                   ▷ For each edit.
        newProgs = []
        for (prog ∈ progs) do                             ▷ For each program.
            rtrndProgs = PROCESS-EDIT(edit, L, prog)      ▷ The language L is a
            newProgs.APPEND(rtrndProgs)                     member variable..
        progs = newProgs
    return progs

algorithm PROCESS-EDIT(edit, L, p)
    retVal = []
    p_copy = CREATE-COPY(p)
    switch (edit.MOVENAME()) do
        case Insert:
            for (term ∈ L.AT(edit.T())) do
                p_copy.INSERT-NODE(term, edit.K(), edit.I())
                retVal.APPEND(p_copy)
            return retVal
        case Relabel:
            retVal = PROCESS-RELABEL(L, edit, p_copy)
            return retVal
        case Delete:
            p_copy.DELETE-NODE(edit.K())
            retVal.APPEND(p_copy)
            return retVal
```



(a) A PTPPT. This PTPPT type checks and is considered the start state in this sequence of PTPPTs.

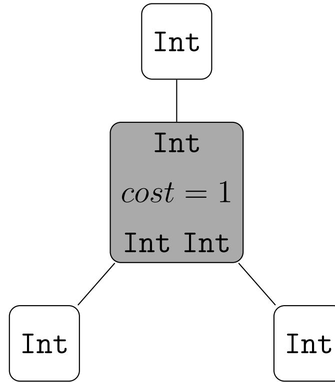


(b) The PTPPT after a node has been relabelled in the PTPPT shown in Figure 6.18a. This PTPPT type checks and is considered the end state in this sequence of PTPPTs.

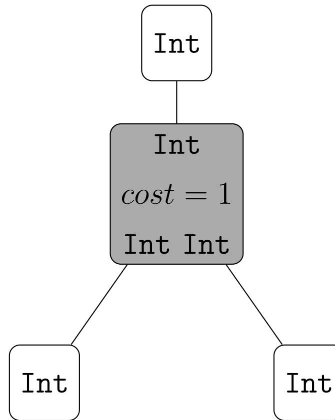
Move Name	t	k	i	b
`Relabel`	$+$	0	N/A	N/A

(c) A term-based edit sequence. It can be used to transform the PTPPT in Figure 6.18a into the PTPPT in Figure 6.18b. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

Figure 6.18: A start state PTPPT, end state PTPPT and edit sequence to transform the start state into the end state.



(a) A TPPT. This TPPT type checks and is considered the start state in this sequence of TPPTs.



(b) The TPPT after a node has been relabelled in the TPPT shown in Figure 6.19a. This TPPT type checks and is considered the end state in this sequence of TPPTs. It is exactly the same as the start state.

Move Name	t	k	i	b
`Relabel`	(`Int` → `Int` → `Int`, 1)	0	N/A	N/A

(c) A type-based edit sequence. It can be used to transform the TPPT in Figure 6.19a into the TPPT in Figure 6.19b. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

Figure 6.19: A start state TPPT, end state TPPT and edit sequence to transform the start state into the end state.

Move Name	t	k	i	b
`Relabel`	$+$	0	0	N/A

(a) A term-based edit sequence. It has been extracted from the type-based edit sequence shown in Figure 6.19c.

Move Name	t	k	i	b
`Relabel`	$-$	0	0	N/A

(b) A term-based edit sequence. It has been extracted from the type-based edit sequence shown in Figure 6.19c. This is an incorrect result, as the term that is being relabelled was originally a $-$.

Figure 6.20: The set of term-based edit sequences that can be extracted from the type-based edit sequence shown in Figure 6.19c. They have been created by substituting the t term in the single edit in Figure 6.19c for every term that unifies with t's type signature and cost. The edit sequences are described in terms of the DSL TE-1, an overview of which is given in Table 6.1. The results are incorrect, as one of the edit sequences relabels a node with the term originally in that node.

sequences; there are more of them than there should be. Specifically the term-based edit sequence in Figure 6.20b would relabel a term in the PTPPT in Figure 6.18a with itself, and should have been omitted. If this term-based edit sequence were applied to a program tree, it would return the same original input program tree, thus the entire algorithm would produce incorrect results.

The core issue with processing relabels in this way is that, when a relabel edit appears in a type-based edit sequence, we do not know what the original type of the term being relabelled was. If the node is being relabelled with a term that has the same type signature and cost as the original, then we require a mechanism to ensure that the case where the algorithm tries to relabel a term t with t is not performed.

We can correct this easily. In the DSL TE-1 in Table 6.1 we included a boolean variable in the relabel edit. When creating the type-based edit sequences, if relabelling a node with a type signature and cost that is the same as before the edit, then the boolean variable should be set to *True*. By doing this, the algorithm that processes relabels can then check the original term to ensure that it does not relabel that term with itself. The amended type-based edit sequence for the TPPTs in Figure 6.19 can be seen in Figure 6.21, and pseudocode to process relabels is shown in Algorithm 6.5.

Move Name	t	k	i	b
`Relabel`	(`Int` $\rightarrow$ `Int` $\rightarrow$ `Int`, 1)	0	N/A	$True$

Figure 6.21: A type-based edit sequence. It has been changed when compared to the edit sequence shown in Figure 6.19c, to ensure that incorrect term-based edit sequences are not created from it. It can be used to transform the TPPT in Figure 6.19a into the TPPT in Figure 6.19b. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1.

Algorithm 6.5 PROCESS-RELABEL

Input: L A map of the language. Each key corresponds to the type and cost of a term. Each element is a vector of terms that have that type signature and cost.

edit A type-based edit.

p A program tree and a context.

Output: The set of output program trees created.

```
algorithm Process-Relabel(L, edit, p)
    retVal = []
    if (edit.B() = True) then
        prevNode = p.Children()[edit.K()]
        for (term ∈ L.At(edit.T())) do
            if (term ≠ prevNode) then
                p_copy = Create-Copy(p)
                Relabel-Node(p_copy, edit.K(), term)
                retVal.Append(p_copy)
    else
        for (term ∈ L.At(edit.T())) do
            p_copy = Create-Copy(p)
            Relabel-Node(p_copy, edit.K(), term)
            retVal.Append(p_copy)
    return retVal
```

Algorithm 6.6 BUILD-TREES

Input:	p	The current input program tree.
	max_{edit}	Maximum number of edits allowed.
	max_{sum}	Maximum sum of the cost of edits allowed.
	max_{SES}	Maximum cost allowed of each edit sequence.
Output		Set of output program trees.

```
algorithm BUILD-TREES(p, max_edit, max_sum, max_SES)
    output = []
    treeContextsStack = [GET-POSITION(p)]
    do
        ctxt = treeContextsStack.POP()                                    ▷ Get context.
        tppts = FIND-TPPTS(COST-ALLOWED(ctxt, max_SES, max_sum), ctxt)
        for (tppt ∈ tppts) do
            for (seq ∈ tppt.GET-SEQUENCES()) do
                results = CREATE-OUTPUT-TREES(seq, ctxt)
                output.INSERT(results, max_edit, max_sum, max_SES)
        moveContextDown = MOVE-CONTEXT-DOWN(ctxt)
        treeContextsStack.APPEND-ALL(moveContextDown)
    while (treeContextsStack.SIZE() > 0)
    return output
```

6.4.7 Compound Moves & Final Algorithms

At this stage in our explanation of GENERATESUCCESSORS, we now have two algorithms; FIND-TPPTS which identifies TPPTs, and CREATE-OUTPUT-TREES which applies type-based edit sequences. In this subsection, we use these to build the algorithm BUILD-TREES that, given an input program tree p, applies edit sequences to p to create the output set of program trees. The pseudocode for BUILD-TREES is shown in Algorithm 6.6.

Informally, BUILD-TREES can be described as follows; the input program tree p is traversed, and at each node a search is conducted for TPPTs. For any that are found, their associated type-based edit sequences are used to generate the output program trees. These new trees are added to the set of output trees returned, and the algorithm moves to the next node in the search.

Build-Trees uses several functions we have not introduced previously. These can be described as follows:

- Move-Context-Down. This function takes the current context given as an argument and returns a set of new contexts. The new contexts are set at the child of the context given as the argument.
- Cost-Allowed. This function computes the n argument to give to Find-TPPTs from the parameters of Build-Trees.

The reader may note that Build-Trees uses several parameters that have not been referenced before. These parameters are described in the rest of this subsection, where we discuss compound moves.

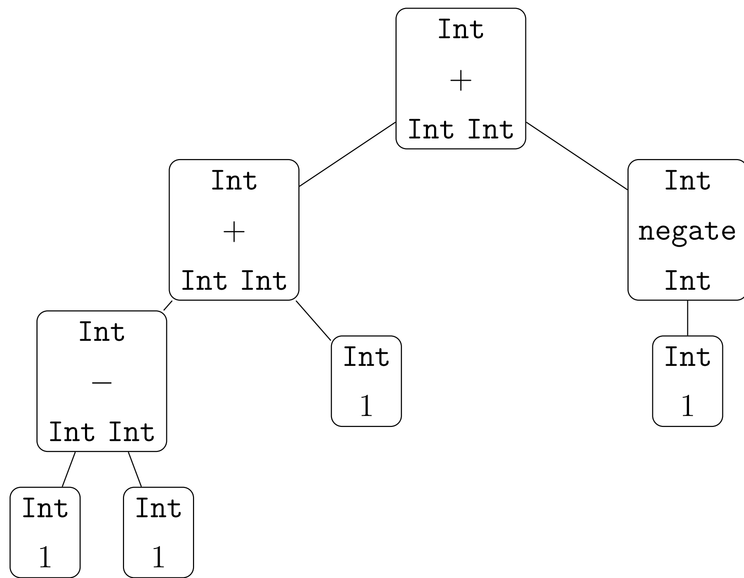
Compound Moves

In this part of the subsection we provide details about the structure that contains the returned output program trees, as well as providing pseudocode for GenerateSuccessors - which was the function that we set out to create in this section. However to do either of these, we must first discuss *compound moves*.

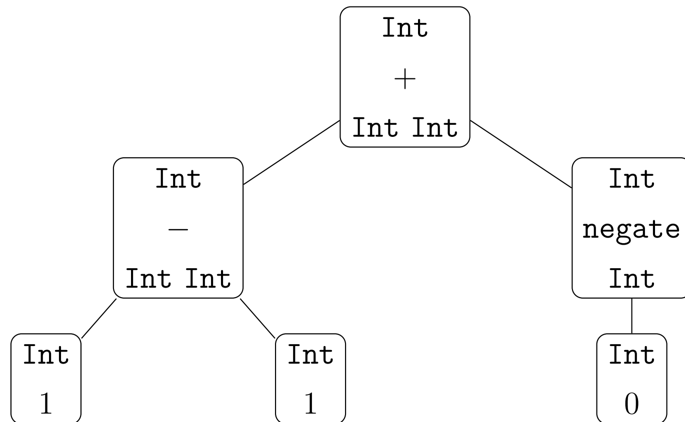
Up to this point, we have assumed that output program trees are created by applying a single edit sequence to an input program tree. However, we will now show an example of a program tree that is within a set cost of edits n that is created by two separate edit sequences being applied at different points in the original program tree.

Consider the three program trees in Figure 6.22. The program tree in Figure 6.22a can be created by applying an edit sequence with a cost of 2 to the program tree in Figure 6.2. The program tree in Figure 6.22b can be created by applying an edit sequence with a cost of 1 to the program tree in Figure 6.2. The program tree in Figure 6.22c can be created from the program tree in Figure 6.2 in two distinct ways; either by creating the program tree in Figure 6.22a and then applying a further edit sequence with a cost of 1, or by creating the program tree in Figure 6.22b and then applying a further edit sequence with a cost of 2. Both of these ways of obtaining the program tree in Figure 6.22c use edits with a cost of 3.

This can be seen clearly by the diagram shown in Figure 6.23, which illustrates the two different methods that can be used to obtain the program tree in Figure 6.22c. In both of these methods the final program tree is obtained by first creating an

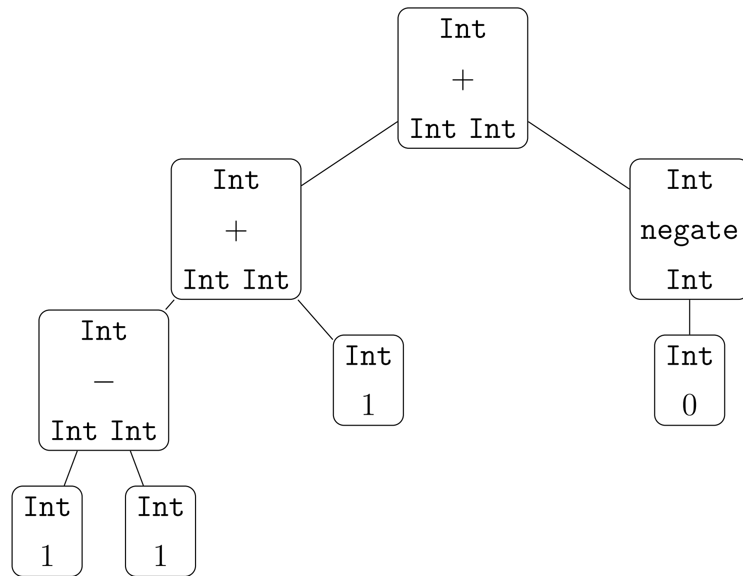


(a) A program tree written in Language EX-1. When the function GENERATESUCCESSORS is given the program tree in Figure 6.2 and an n value of at least 2, this program tree would be an example of output returned.



(b) A program tree written in Language EX-1. When the function GENERATESUCCESSORS is given the program tree in Figure 6.2 and an n value of at least 1, this program tree would be an example of output returned.

Figure 6.22: Three program trees written in Language EX-1. When the function GENERATESUCCESSORS is given the program tree in Figure 6.2 and an n value of at least 3, these program trees would be examples of output returned.



(c) A program tree written in Language EX-1. When the function GenerateSuccessors is given the program tree in Figure 6.2 and an n value of at least 3, this program tree would be an example of output returned.

Figure 6.22: Three program trees written in Language EX-1. When the function GenerateSuccessors is given the program tree in Figure 6.2 and an n value of at least 3, these program trees would be examples of output returned. (Continued)

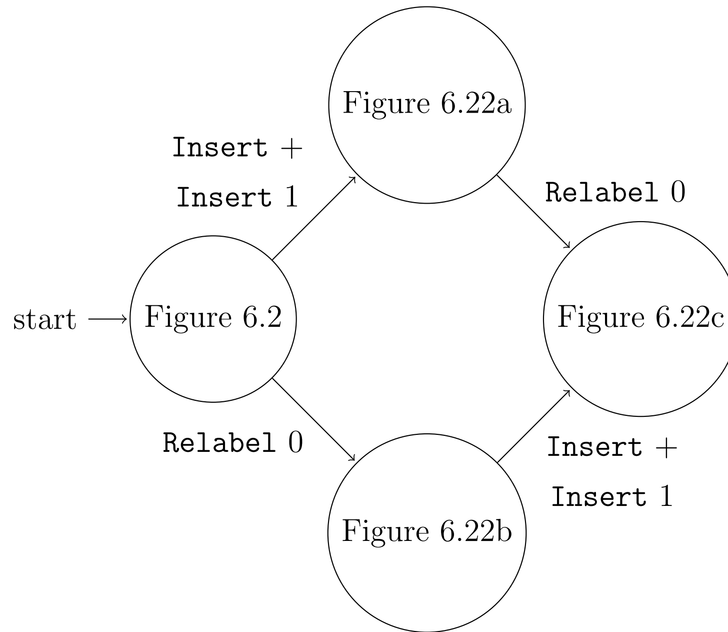


Figure 6.23: Diagram showing two ways in which edit sequences can be chained together to move via intermediary trees to transform the program tree in Figure 6.2 into the program tree in Figure 6.22c.

intermediary program tree. The reader should note that both intermediary trees would also be returned from GenerateSuccessors.

Figure 6.23 serves as an example of the way in which more than one edit sequence can be applied to a program tree to produce an output program tree. We call an edit sequence that is made up of two or more singular edit sequences, each of which could be applied individually to a type correct program tree to produce another type correct program tree, a compound move. Consequently, this suggests the need for another layer to the GenerateSuccessors algorithm; one which takes each of the created output program trees, and applies further edits to them if there are valid edit sequences that can be made according to the parameters imposed by the algorithm. We can reduce the additional computational overhead this adds with the following technique; after a TPPT t is recognised and t's associated edit sequence applied to an input program tree to produce an output program tree p, any subsequent edits applied to p must be performed after the node in p that corresponds to the root node of t. By "after", we mean the ordering of nodes as described by a pre-order traversal.

The additional parameters in Build-Trees come from the use of compound moves. Instead of having one cost parameter n in the algorithm, we now have three;

max_{edit}, max_{sum} and max_{SES}. max_{edit} refers to how large the cost of a single edit sequence can be. max_{sum} refers to how large the sum of all of the edit sequences can be, and is analogous to the original n value. max_{SES} refers to how many individual edit sequences can be applied to the original program tree.

In Algorithm 6.7 we show the final component of the algorithms described in this section, GenerateSuccessors. It works as follows; first, it adds the initial tree to the stack of unexplored type safe program trees, then proceeds to a do-while loop. On each iteration of the loop, an element is removed from the stack. This element is then added to the output collection if the cost of the edit sequences to create it are appropriate according to the input parameters. We then pass this tree to Build-Trees. The reader should note that the language L is a variable in the object Neighbourhood-Generation, and we are therefore able to reference it. All output generated from Build-Trees is added to the stack, and this process continues until the stack is empty.

The reader should note that the function GenerateSuccessors has changed compared to the function prototype shown in Algorithm 6.1. We have added additional parameters max_{sum}, max_{edit} and max_{SES}. Their meaning is identical to that described above, however the reader should note that, to use GenerateSuccessors to generate a neighbourhood according to $N(n)$, the parameters would have to be instantiated as follows; $max_{edit} = n$, $max_{sum} = n$ and $max_{SES} = n$. We have discussed previously how the constructor of Neighbourhood-Generation is given the n_{max} variable. We stipulate as a precondition that the max_{sum} and max_{SES} variables cannot be greater than n_{max}.

This concludes our explanation of the GenerateSuccessors algorithm. In the next section we present a randomised version of the GenerateSuccessors algorithm called GenerateSuccessors-Rnd.

6.5 Generate Successors Randomly

In the previous section we described GenerateSuccessors, an algorithm for creating the neighbourhood of a program tree. However, as we have seen in Chapter 5, some neighbourhoods can be very large and to generate all neighbours may require computational resources that are not available to us. In this section, we describe an algorithm that probes a neighbourhood randomly, without generating all neighbours.

Algorithm 6.7 GENERATESUCCESSORS

Input:	p	The input program tree.
	max_{edit}	Maximum amount of edit sequences allowed.
	max_{sum}	Maximum sum of all edit sequences allowed.
	max_{SES}	Maximum sum of each individual edit sequence allowed.
Output:		Returns a collection of output program trees which can be created from p using edit sequences *seq* which meet the following criteria; *seq* contains no more than max_{edit} individual edit sequences. Any single edit sequence in *seq* has a cost of no more than max_{SES}. The maximum sum of the edit sequences in *seq* is at most max_{sum}.

algorithm GENERATESUCCESSORS(p, max_{edit}, max_{sum}, max_{SES})
    $output = []$
    $stack = [p]$
    do
        $tree = stack$.POP()
        $output$.INSERT($tree$)
        res = BUILD-TREES($tree$, max_{edit}, max_{sum}, max_{SES})
        for ($t \in res$) **do**
            $output$APPEND(t)
    while ($stack$.SIZE() > 0)
    return $output$

The randomised algorithm described in this section is similar to the algorithm described in the previous, and therefore we do not provide the same level of in-depth explanation. The algorithm, called GenerateSuccessors-Rnd, is shown in Algorithm 6.8.

GenerateSuccessors-Rnd works as follows; using some user defined criteria, the algorithm first decides how many compound moves it will attempt to make. This value, called $nMovesToMake$, is in the range $\{1 \ldots max_{edit}\}$. The algorithm then proceeds in a loop, which is completed $nMovesToMake$ times. On each iteration the algorithm takes the previous type safe tree and applies an edit sequence to it. On the first iteration, the previous tree is the input tree. The algorithm picks a context at random in the tree, then finds any TPPTs that can be recognised at that point. From all those TPPTs that have been found, it picks a term-based edit sequence at random from one of the type-based edit sequences. It then applies this edit sequence to the previous tree, creating a new tree. This process continues until either the number of required compound moves have been performed, or some other input parameter has been exceeded.

GenerateSuccessors-Rnd may sometimes be unable to find a program tree, due to the TPPT chosen not having any valid edit sequences, or the randomly chosen node in the tree not having any associated TPPTs. Though we do not show it here, there are mechanisms in place to ensure that if a tree cannot be created for any reason, then a new attempt is made. This continues until some pre-defined termination criteria is met.

The reader should note that the program trees returned from GenerateSucc-essors-Rnd may not be created through an edit sequence with the minimum cost. That is to say, they will still be within the bounds described by the input parameters, but the edit sequence used to obtain a program tree may have a cost that is not the minimum of all possible edit sequences that could be used to obtain that program tree. In the experiments performed in Chapter 5, we did not make any distinction between the program trees in the neighbourhood that were obtained through edit sequences with different costs, and therefore using this method of generating the neighbourhood in those experiments would not fundamentally change how the program trees were picked. However, if further experiments were performed that took the cost of the edit sequences into consideration, then this randomised algorithm may not be appropriate for those needs. Our randomised algorithm cannot guarantee that it will find the

Algorithm 6.8 GENERATESUCCESSORS-RND

Input:	p	The input program tree.
	max_{edit}	Maximum amount of edit sequences allowed.
	max_{sum}	Maximum sum of all edit sequences allowed.
	max_{SES}	Maximum sum of each individual edit sequence allowed.
	rnd	Random generator.
Output:		Returns an output program tree which can be created from p using edit sequences seq which meet the following criteria; seq contains no more than max_{edit} individual edit sequences. Any single edit sequence in seq has a cost of no more than max_{SES}. The maximum sum of the edit sequences in seq is at most max_{sum}.

```
algorithm GENERATESUCCESSORS-RND(p, max_edit, max_sum, max_SES, rnd)
    nMovesToMake = PICK-MOVES(max_edit, rnd)
    currentTree = p
    for (i ∈ {1...nMovesToMake}) do
        treeCxt = PICK-NODE-IN-TREE-RND(currentTree, rnd)
        tppts = FIND-TPPTS(
                COST-ALLOWED(treeCxt, max_SES, max_sum), treeCxt)
        tppt = tppts[rnd.RANDOMNUMBER(tppts.SIZE())]
        sequences = tppt.GET-SEQUENCES()
        seq = sequences[rnd.RANDOMNUMBER(sequences.SIZE())]
        results = CREATE-OUTPUT-TREES(seq, treeCxt)
        result = results[rnd.RANDOMNUMBER(results.SIZE())]
        currentTree = result
    return currentTree
```

edit sequence with the minimum cost because, as we do not store all neighbours, the algorithm is unable to tell if the edit sequence used has the minimum cost of all edit sequences that could be used to obtain that program tree.

6.6 Searching for Edit Sequences

In Section 6.2 we introduced the object Neighbourhood-Generation and two function signatures that make up the algorithm to find the neighbours of a given input program tree. In Section 6.4, we described one of these functions, GenerateSuccessors. In this section, we describe how the second of these functions, the constructor to Neighbourhood-Generation, is formulated. It is designed to identify all possible start state TPPTs that can be created with a cost of up to n_{max}, and search them to find all type-based edit sequences that lead to type safe end state TPPTs. It is these start state TPPTs that are recognised in GenerateSuccessors and the accompanying edit sequences applied to input program trees.

This section is broken up into four parts; in Section 6.6.1, we show how we generate every possible type safe start state TPPT for a language. In Section 6.6.2 we discuss some properties of the MTED problem and type systems that direct the design of our algorithm. In Section 6.6.3 we show some examples of how single edits are considered in the overarching algorithm. In Section 6.6.4 we provide the pseudocode for finding all edit sequences. Finally in Section 6.6.5 we provide some empirical data concerning the algorithm's performance when generating all edit sequences for some of the languages used in this thesis.

6.6.1 Finding all TPPTs

In this subsection we detail the methodology used when finding all possible start state TPPTs for a given language L and maximum sum of edits score n_{max}. It is these that we use as the start point to search for end state TPPTs and edit sequences.

The recognition of TPPTs is one of the core components of our algorithm to generate a program tree's neighbourhood. It allows us to identify patterns in a program tree, so that we can apply the correct edit sequences to the input program. It is vitally important that we generate all possible type safe TPPTs and explore them, so that we can generate the neighbourhood to any provided candidate tree. As we use a closed language in the algorithm, it is possible for us to generate all possible

$$\{ \quad [\texttt{Int}, \texttt{Int}] \quad , \quad [\texttt{Int}] \quad , \quad [\texttt{Bool}, \texttt{Int}, \texttt{Int}] \quad , \quad [\texttt{Bool}] \quad \}$$

Figure 6.24: The set of unique vectors that are used as the root nodes when generating every possible TPPT for Language EX-1.

start state TPPTs that would ever be encountered in an input program tree.

The algorithm is recursive in nature, and similar in design to the algorithm to generate all programs exhaustively shown in Algorithm 2.17. It is split into three parts. The first CREATE-ALL-ROOTS generates all possible roots of all required TPPTs. The second algorithm, CREATE-RTPPTS, generates all possible configurations of RTPPTs for a given TPPT root. Finally, in CREATE-TPPTS these two algorithms are combined to create all possible TPPTs.

Create-All-Roots

In the example TPPTs (and PTPPTs) we have seen up to this point, all root nodes consist of a vector of types. Every input tree given to the GENERATESUCCESSORS function should type check, and therefore when finding TPPTs within an input program tree, we can assume that we will only scan for type safe TPPTs. The set of initial TPPTs that we create in the constructor will only be those TPPTs that type check. The root node of every (type safe) TPPT directly mirrors a term in the language. Or more specifically, directly mirrors the arguments of that term. To generate all roots, we simply need to scan the language and extract every unique vector of arguments from each term. As an example, we show the set of unique vectors of arguments for Language EX-1 in Figure 6.24.

The reader may note two things; firstly, terms that have no arguments are not considered - so there are no empty vectors. Secondly, not every unique term in Language EX-1 is directly mirrored by a vector. For example, `lessThan` has the same required arguments as $+$ and $-$, yet no distinction between these terms is required.

Several additional vectors must also be added to this set. These vectors represent the `identity` function that is added to all input program trees in GENERATESUCCESSORS (see Section 6.4.5). This allows the GENERATESUCCESSORS algorithm to operate on any function described under a given language L. For example a function

Algorithm 6.9 CREATE-ALL-ROOTS

Input: None.
Output: A set of vectors. Each vector's elements are principle types from the language L.

```
algorithm CREATE-ALL-ROOTS()
    set = []
    for (t ∈ L) do                        ▷ The language L is a member variable.
        vector = GET-ARGS(t)
        set.INSERT(vector)
        identityFunc = GET-RETURN-TYPE(t)
        set.INSERT([identityFunc])
    return set
```

could be given to GENERATESUCCESSORS that has a type of `Bool`. In Figure 6.24, only one additional vector is added to the set of roots through this process, that of [`Bool`].

The algorithm to realise this process is called CREATE-ALL-ROOTS and is presented in Algorithm 6.9. Its functionality is simple; it iterates through every term in the language and adds a vector representative of that term's arguments to the set to be returned. It also iterates through each return type of every function, to ensure that a TPPT root that mirrors the `identity` function is represented for any potential input program tree.

Create-RTPPTs

The algorithm we describe here, called CREATE-RTPPTS, creates all sequences of RTPPTs that have the required cost and adhere to the types specified in the root, which is provided as an argument. It works in a recursive manner, similar in style to FIND-RTPPTS, which was shown in Algorithm 6.2. Each invocation takes a cost value n, a vector of return types *vector* that the created sequences of RTPPTs should satisfy, and returns a vector of collections of sequences of RTPPTs. That is to say, for each $cost \in \{0 \ldots n\}$ multiple sequences of RTPPTs may be returned. The pseudocode for CREATE-RTPPTS is shown in Algorithm 6.10.

The algorithm has two main steps; a recursive step, and a building step. The recursive step works as follows; for each element $v \in vector$, all typed-named nodes

Algorithm 6.10 CREATE-RTPPTS

Input:	n	The maximum cost of the RTPPTs to be created.
	vector	Vector of types.
Output:		A vector of collections of sequences of RTPPTs. The vector index i is in the range $\{0 \ldots n\}$, signifying the total cost that all the sequences of RTPPTs contained within i's collection have.

algorithm CREATE-RTPPTS(n, *vector*)
  results = []
  if (*vector*.SIZE() = 0) **then** *results*[0] = []
  nextLevel = []
  for ($i \in \{0 \ldots vector.\text{SIZE}() - 1\}$) **do**
    $c = vector[i]$
    nextLevel[0][i] = [GET-RETURN-TYPE(c)]
    for (*tnt* $\in$ GET-UNIFIED-TYPES(n, c)) **do**
      cost = COST(*tnt*)
      newArgs = *tnt*.GET-ARGS()
      recResults = CREATE-RTPPTS($n - cost$, *newArgs*)
      for ($n_{local} \in \{0 \ldots n - nodeCost\}$) **do**
        for ($result \in recResults[n_{local}]$) **do**
          tree = MAKE-TREE(c, *result*)
          $nextLevel[cost + n_{local}][i]$.APPEND(*tree*)
  for ($n_{local} \in \{0 \ldots n\}$) **do**
    $results[n_{local}]$ = ALL-COMBINATIONS(n_{local}, *nextLevel*)
  return *results*

ts are generated that have a return type equal to v, and whose cost is $\leq n$. For all of ts, Create-RTPPTs is called on the set of argument types of each $t \in ts$, with an updated maximum cost. These recursive results are then paired together with their respective t to create a set of singular RTPPTs with t as their root.

In Figure 6.25 we show all singular RTPPTs generated from the principle type `Int` and an n value of 2 in Language EX-1.

The second part of Create-RTPPTs considers all combinations of results, such that their total cost is at most n and their root nodes directly mirror the required types in *vector*. This is achieved in a similar manner to that shown in Algorithm 6.2. From the trees shown in Figure 6.25, the total number of RTPPTs for [`Int`] with a cost of at most 2 would be 27, and for [`Int`, `Int`] 65.

We can see that, even for a small n value, and with Language EX-1 being a small language, there are many possible RTPPTs. This also illustrates why we decided to use TPPTs instead of PTPPTs; the number of RPTPPTs would be even greater, increasing the computational overhead when searching for edit sequences.

Create-TPPTs

We can now create an algorithm to build all TPPTs. This algorithm, called Create-TPPTs, is shown in Algorithm 6.11. It works as follows; it calls Create-All-Roots to generate the set of roots, then calls Create-RTPPTs on each root. It then adds the correct root to each result. It is similar to Find-TPPTs shown in Section 6.4.4.

This algorithm will build all possible TPPTs that will be relevant for a given language, up to a cost of n_{max}. In the next subsections it is these TPPTs that we will use as the starting point to search for type-based edit sequences.

6.6.2 Searching TPPTs: Intuition

Up to this point in our description of the algorithms in the preceding sections, we have presumed that the type-based edit sequences have been generated. In this subsection, Sections 6.6.3 and 6.6.4 we detail how the search for valid edit sequences is conducted.

Broadly, this search can be considered a microcosm of the initial problem; given a program tree, find the type safe program trees within a set cost of edits from the initial tree. Rather than working on program trees, we are working on the TPPT

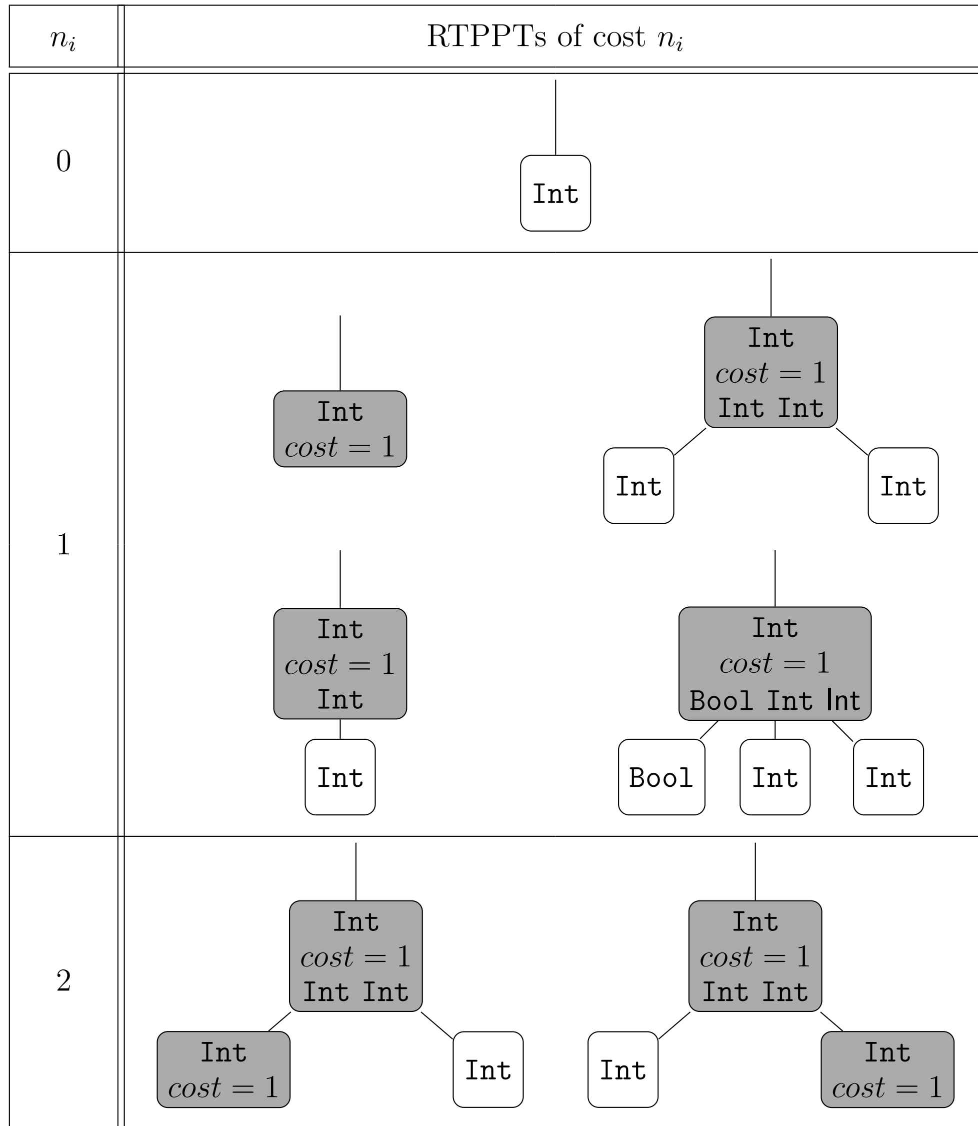


Figure 6.25: The set of results stored in the *nextLevel* variable in the CREATE-RTPPTS algorithm when given the $vector = [\texttt{Int}]$ and an n value of 2. For each $n_i \in \{0 \ldots n\}$, a set of results are shown for each element in *vector*.

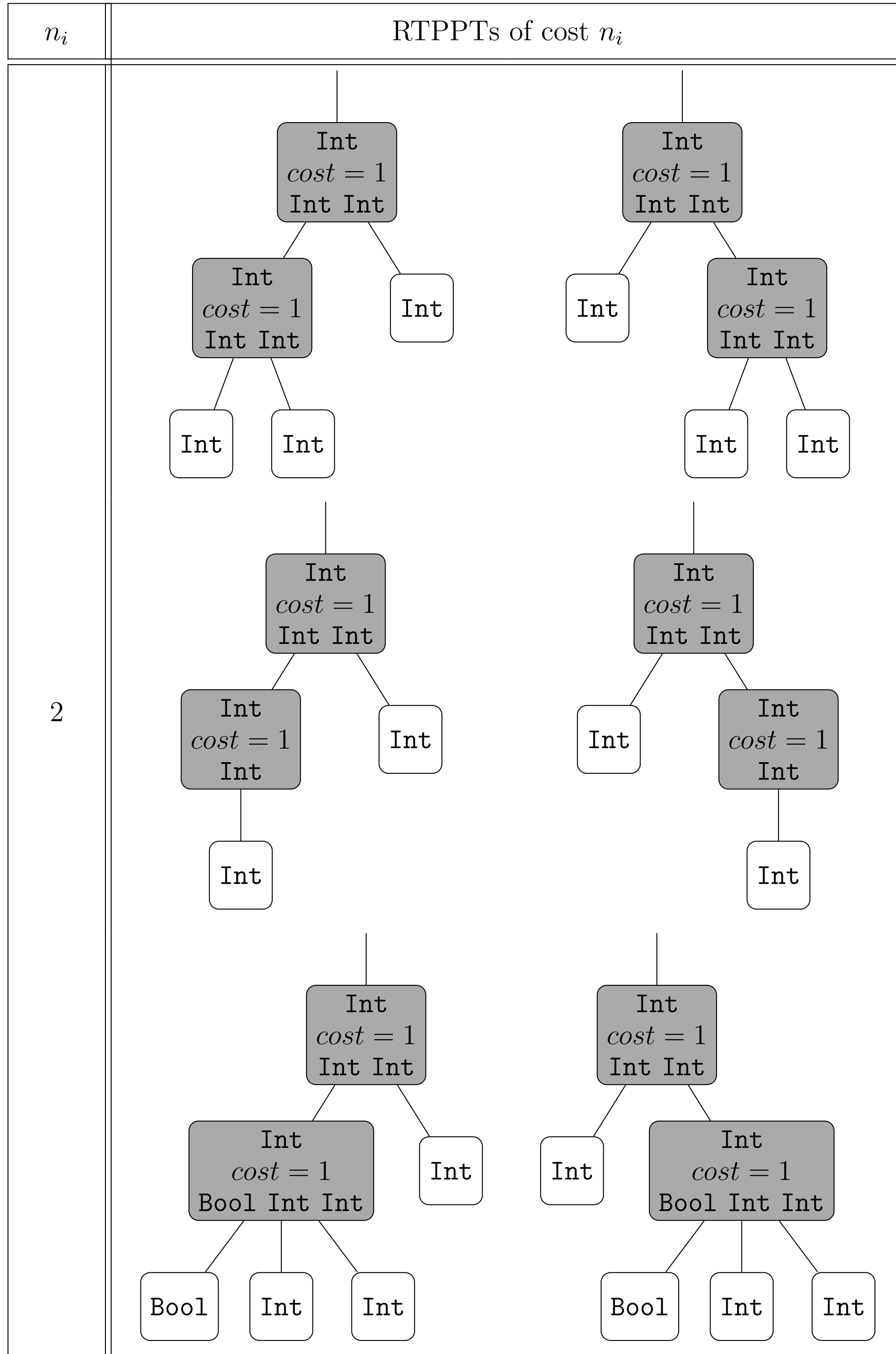


Figure 6.25: The set of results stored in the *nextLevel* variable in the CREATE-RTPPTs algorithm when given the *vector* = [Int] and an n value of 2. For each $n_i \in \{0 \ldots n\}$, a set of results are shown for each element in *vector*. (Continued)

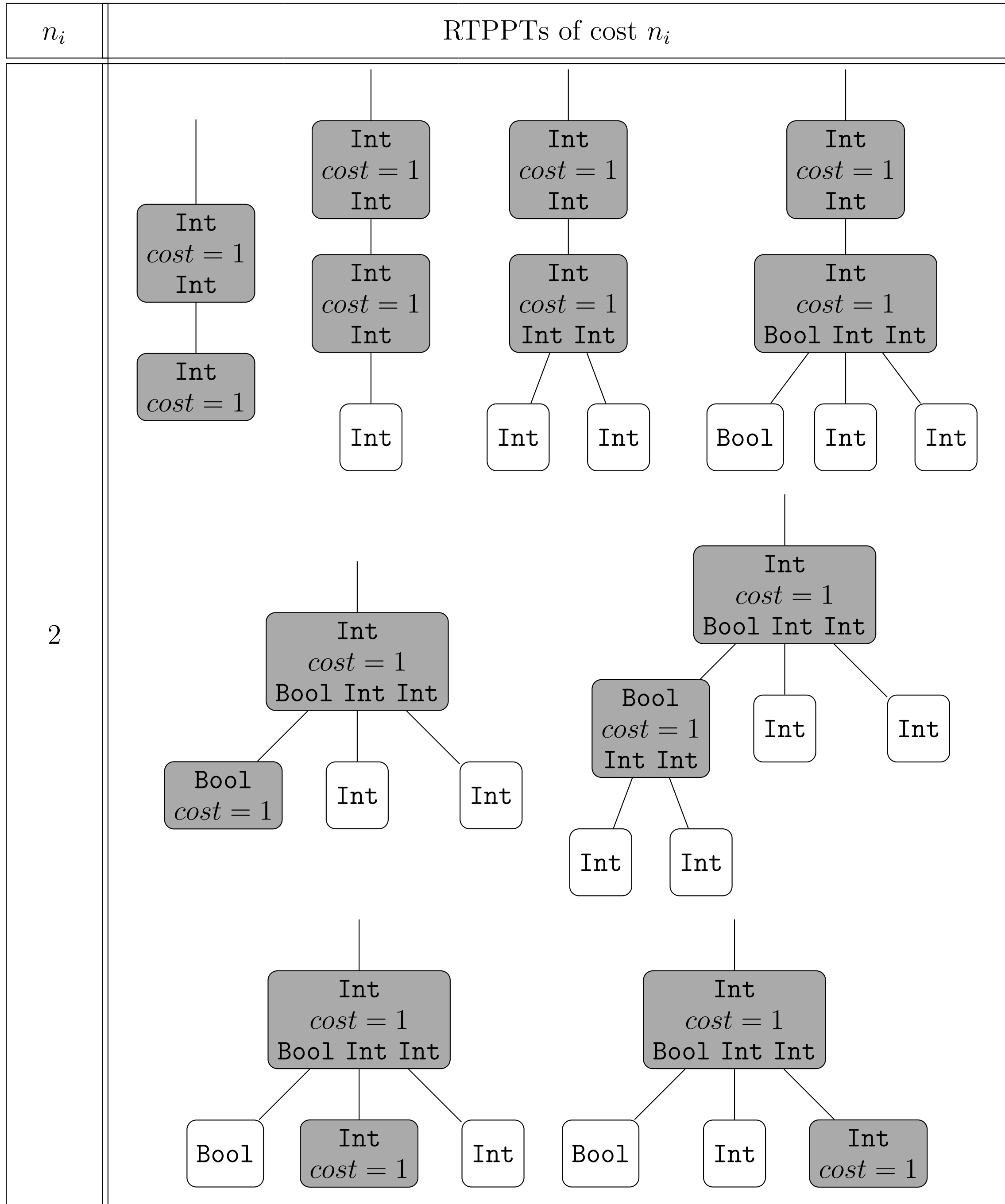


Figure 6.25: The set of results stored in the *nextLevel* variable in the CREATE-RTPPTS algorithm when given the $vector = [\texttt{Int}]$ and an n value of 2. For each $n_i \in \{0 \ldots n\}$, a set of results are shown for each element in *vector*. (Continued)

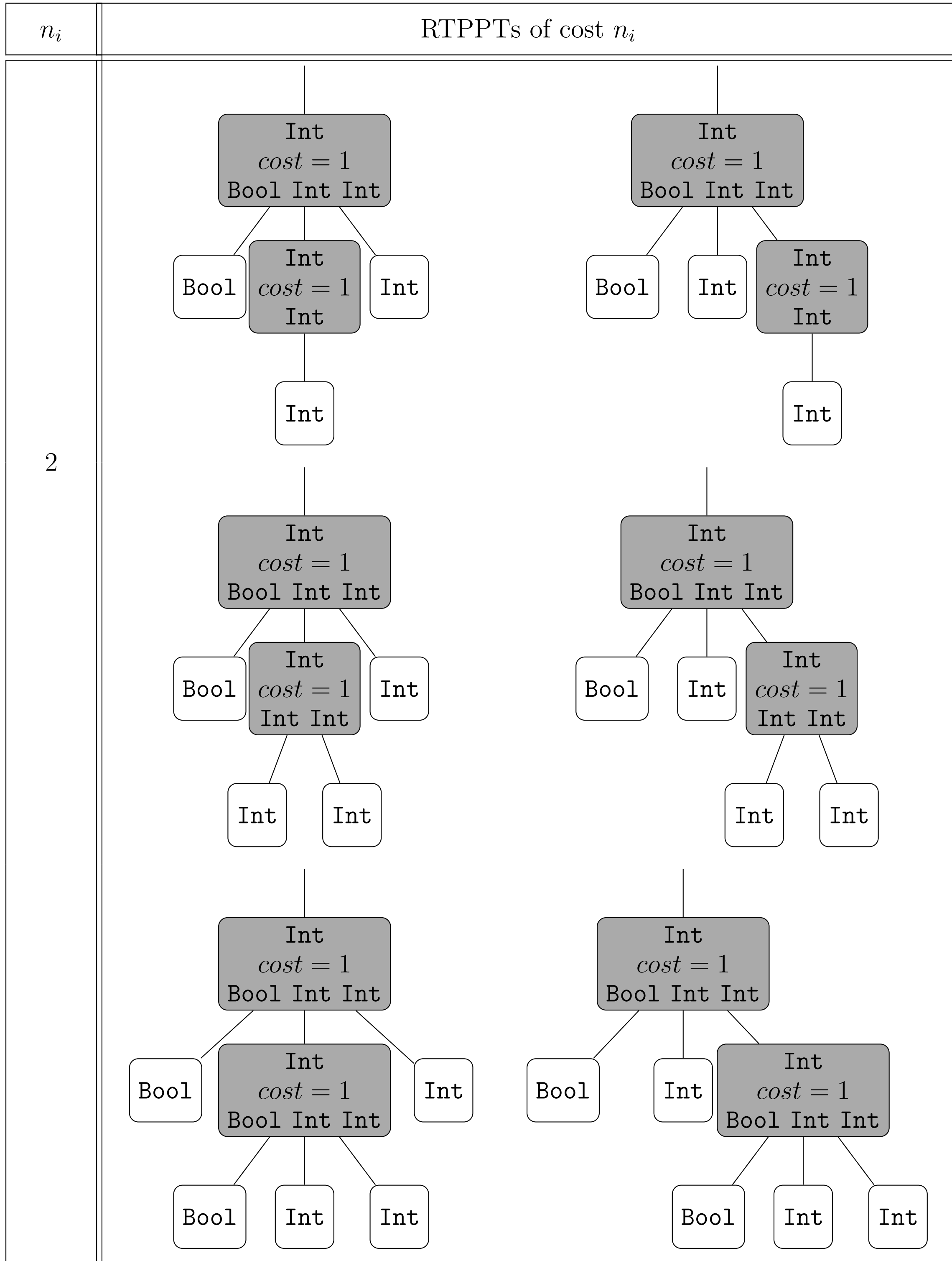


Figure 6.25: The set of results stored in the *nextLevel* variable in the CREATE-RTPPTS algorithm when given the *vector* = [Int] and an n value of 2. For each $n_i \in \{0 \ldots n\}$, a set of results are shown for each element in *vector*. (Continued)

Algorithm 6.11 GENERATE-TPPTS

Input: n_{max} The maximum cost of the created TPPTs.
Output: The set of start state TPPTs that are used to search for edit sequences.

```
algorithm GENERATE-TPPTS(n_max)
    retVal = []
    roots = CREATE-ALL-ROOTS()
    for (root ∈ roots) do
        tpptChildrenVect = CREATE-RTPPTS(n_max, root)
        for (tpptChildren ∈ tpptChildrenVect) do
            tppt = MAKE-TREE(root, tpptChildren)
            retVal.APPEND(tppt)
    return retVal
```

representation. To be clear, our goal is, given a start state TPPT, to find all end state TPPTs, for which the cost of the edit sequences are within the bounds stipulated.

In this subsection we make several observations about the problem, to provide intuition to the reader about the design choices made when constructing the algorithm.

Type Checking TPPTs

Our first observation is about type systems, and specifically type checking TPPTs. A TPPT contains all the required typing information to be type checked; a node p containing a type signature $\{t_1 \dots t_n\}$ and children $c_i \in \{c_1 \dots c_{n-1}\}$ is correct according to the type system if each c_i's return type is identical to t_i. A type checking algorithm could be created for TPPTs, similar to that shown in Algorithm 2.16. The key observation from such an algorithm is that the types of the children of a node can be checked in any order, and the correct result will be produced.

This suggests to us that, if an ordering were imposed on where edits could be made, for example using a pre-order traversal, then we could use this ordering to type check nodes as we make edits. By doing so, we could prune search derivations if they do not type check, rather than having to finish the search derivation before type checking.

MTED Intuition

It is important to note that, when computing a sequence of edits to transform one tree to another, the ordering of these edits can be important. While it is not true for all edit sequences, for some there are insertions and deletions that must occur in a specific order. The relabelling of a node can occur at any time, but it is of no consequence to the observations we make here.

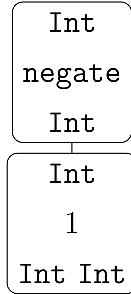
In Figure 2.13 we presented a recursive solution for computing the MTED between two trees. In that solution, all possible edit sequences to move between the two trees are computed, and that which has the minimum cost is returned. Remembering that this solution operates on forests rather than trees, the nodes in the forest are visited from right-to-left, with the solutions to the subproblems computed before moving to the previous element in the forest. Some algorithms [142] used to solve the MTED problem consider trees in a left-to-right order in the forest, and solve the subproblems for one tree before progressing to the next. This is in essence, a pre-order traversal.

Our second observation about the MTED metric concerns the undoing of edits. After an edit has been made to a node in a tree, that node is not considered for further editing. Consider the term-based edit sequence in Figure 6.26b which, when applied to the program tree in Figure 6.26a, produces the same program tree. This kind of edit sequence should not be produced by our algorithm, as it would not be the edit sequence with the minimum cost - the empty edit sequence would have the minimum cost.

In general, undoing a previous edit will not progress the search. A deleted node should not be re-inserted, an inserted node not deleted or relabelled, and a relabelled node not re-relabelled or deleted. For our algorithm, we stipulate that after a node has been edited - either inserted or relabelled - it should not be considered for further edits. We do not stipulate that deleted nodes should not be re-inserted. Re-inserting a deleted node only creates a single successor state, whereas those we do prohibit - re-relabelling for example - can create an exponential number of redundant successor states.

Bounds on Edits

In Section 6.4.4 we described an algorithm that can create all TPPTs. It is TPPTs that will be the start states for our search algorithm. However, as noted in Section 6.4.3, if an edit sequence is only comprised of insertions, then no typed-named nodes



(a) A program tree written in Language EX-1.

Move Name	t	k	i	b
`Delete`	N/A	0	N/A	N/A
`Insert`	`negate`	0	1	N/A

(b) A term-based edit sequence. It is described in terms of the DSL TE-1, an overview of which is given in Table 6.1. When applied to the program tree in Figure 6.26a, the same output tree is produced.

Figure 6.26: A start state program tree and an edit sequence. When the edit sequence is applied to the program tree in Figure 6.26a, the same program tree is produced.

are required. In fact, the format of the TPPT that we use provides us with the opportunity to limit which types of edits are allowed. Specifically, if a TPPT has a cost of n, then the maximum cost of the deletions or relabels we can perform is n, and the maximum cost of the insertions we can perform is $n_{max} - n$.

It is these observations that direct our choices when designing the search algorithm. In the next subsection we describe conceptually how the search algorithm works, and provide examples of how specific edits effect the progression of the search.

6.6.3 Searching TPPTs: Examples

In this subsection we describe conceptually how the algorithm to find all valid edit sequences works. We also show, through examples, how each edit is processed by the algorithm.

The algorithm itself is simple in nature; from a high-level viewpoint, it can be considered a search on the set of start state TPPTs for any valid end state TPPTs. Every edit is considered on one TPPT to create a set of successor TPPTs, and any that are found to lead to type safe TPPTs are returned with their accompanying edit sequence.

However, from a low-level viewpoint, the way in which the algorithm works is more nuanced. On initialisation, a context is set at the root of the start state TPPT. From the algorithm's viewpoint, it can only see the root node and its children, no other nodes. Edits are only considered on this context in the tree. Instead of an edit occurring and a new TPPT being created, every time an edit occurs the TPPT is split into either one or two subproblems, depending on the edit that took place.

These subproblems are TPPTs in their own right, created from parts of the parent problem. This process continues, with edits taking place and new subproblems being created. Any that are found to create type safe TPPTs are returned, with their accompanying edit sequence. The edit sequences from these subproblems are then combined, and returned to the previous subproblem. This continues until all valid edit sequences are returned to the start state TPPT.

From our observations in the previous subsection, we know that once an edit has been performed, any additional edits cannot occur before that initial edit. We enforce this by creating subproblems that have removed any nodes before the edit according to a pre-order traversal. These removed nodes must be type checked, as this removed part of the problem may be incorrectly typed, and therefore any returned subproblems would not lead to type safe TPPTs. We also stipulate which edits can occur, which follow from the observations we made about the bounds on the different types of edits allowed.

In the remainder of this subsection we consider each of the three edits that can occur, and discuss how the algorithm creates subproblems from these edits.

Insert

Let us consider the TPPT shown in Figure 6.27. To our algorithm, all contexts in every TPPT are visualised in this manner; it can only see the parent node and the child nodes. It is only when subproblems are created and contexts changed that the rest of the tree may be considered. The algorithm is only aware of the return type of the child nodes, whether each child node is a typed-named node or not, and the argument types of the root.

When an insertion edit occurs, two subproblems are created, and one set of nodes must be type checked. If a node t is inserted under a root r at position k with i of r's children as its new children, then the subproblems created and the nodes that need to be type checked can be described as follows:

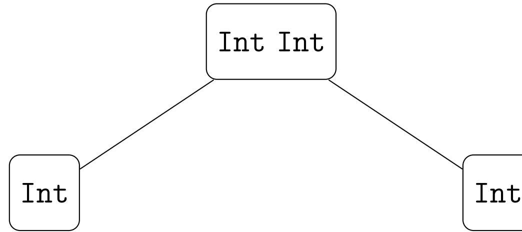


Figure 6.27: A TPPT. This TPPT type checks.

- One subproblem is created underneath the inserted node. This subproblem is represented by the TPPT *tp*. *t*'s argument types become the root of *tp*. The *i* children that previously belonged to *r* become the sequence of RTPPTs that are *tp*'s children. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them purple.

- One subproblem is created to the right of the inserted node. This subproblem is represented by the TPPT *rt*. *rt*'s root node is a subvector of *r*'s root node, from position *k* onward. *rt*'s child nodes are the subvector of *r*'s child nodes from position $k + i$ onward. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them blue.

- One set of nodes must be type checked. This set of nodes can be represented by the TPPT *tc*. *tc*'s root node is a subvector of *r*'s root node, from position 0 to $k - 1$. *tc*'s child nodes are the subvector of *r*'s child nodes from 0 to $k - 1$. These nodes must be type checked before any subproblems are created, as if they do not type check, then none of the subproblems solutions are valid. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them orange.

To begin, let us consider the insertion of an unlabelled node. We use an unlabelled node to show the various configurations of insertions that can occur, and show examples of how the subproblems and nodes to type check are formulated. In Figure 6.28 we show the six possible successor states of the TPPT in Figure 6.27 after an unlabelled insertion has occurred. In each configuration of nodes we show the TPPTs *tc* and *rt*.

We can see that one of the configurations of nodes after insertion will not lead to a type safe end state. The third configuration of nodes will create a subproblem represented by the TPPT *rt* that has an empty vector as its root and a single child.

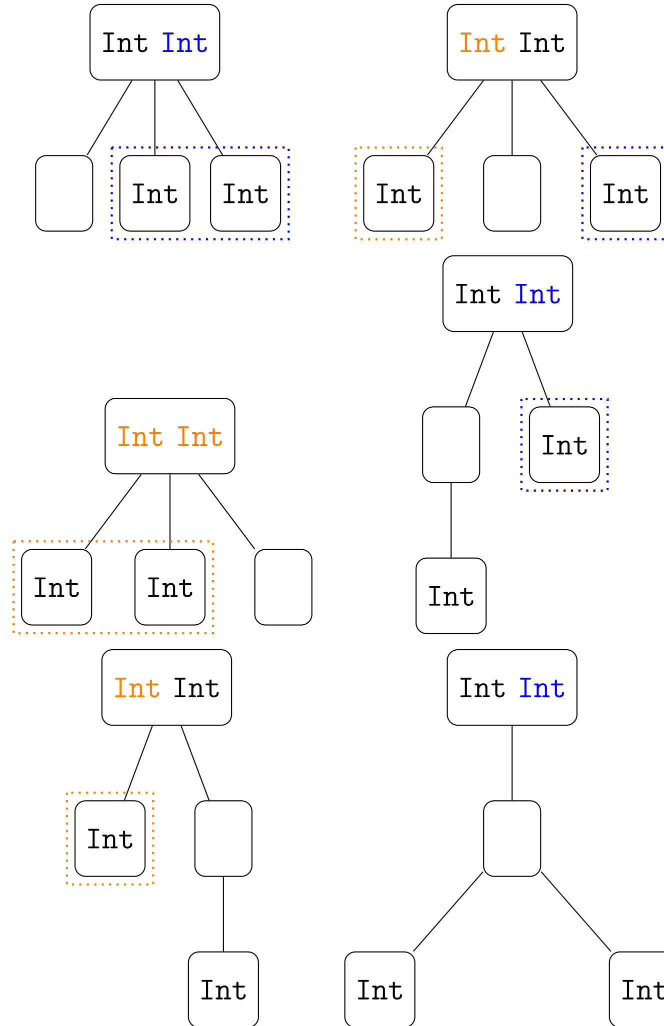


Figure 6.28: The six possible successor states created by inserting an unlabelled node into the TPPT shown in Figure 6.27. In each successor state we show the subproblem represented by the TPPT *rt*, highlighted in blue, and the TPPT *tc* that needs to be type checked for the search to continue, highlighted in orange. The reader may note in the final TPPT that there is a blue type in the root, but no blue child. The subproblem created here is one that requires an `Int` type, but has no child nodes.

(a) The TPPT after a node has been inserted in the TPPT shown in Figure 6.27. It is a valid successor state and shows the two subproblems created.

(b) The TPPT after a node has been inserted in the TPPT shown in Figure 6.27. It is an invalid successor state, as the label inserted does not satisfy the root's type. This search derivation would terminate.

Figure 6.29: Two TPPTs created by inserting a node into the TPPT in Figure 6.27. One is a valid successor state, one is not.

Though we do not know what we have inserted, we know that we cannot delete this node, and therefore it must have a type, which cannot be satisfied by the root.

As to which nodes can be inserted, we use the type-compressed form of Language EX-1 shown in Figure 6.17 to find appropriate typed-named nodes to insert. We can only insert a node as the kth child that has a return type which is the same as the kth type in the vector in the root. We can see two examples of successor state TPPTs in Figure 6.29 with a node inserted; one that is representative of the terms with a type signature of `Int` $\rightarrow$ `Int` $\rightarrow$ `Int`, and one that is representative of the terms with a type signature of `Int` $\rightarrow$ `Int` $\rightarrow$ `Bool`. The first is a valid successor state, and the second is not. Since we cannot perform further edits before the previously edited node (as we have stipulated that edits must occur in a pre-order traversal), the type inconsistency in the TPPT in Figure 6.29b cannot be resolved, and this search derivation can be pruned from the search space.

Relabel

When a relabel edit occurs, two subproblems are created, and one set of nodes must be type checked. It is performed in a similar way to how an insertion is performed.

If a node t is relabelled with a new label l_{new} under a root r at position k, then the subproblems created and the nodes that need to be type checked can be described as follows:

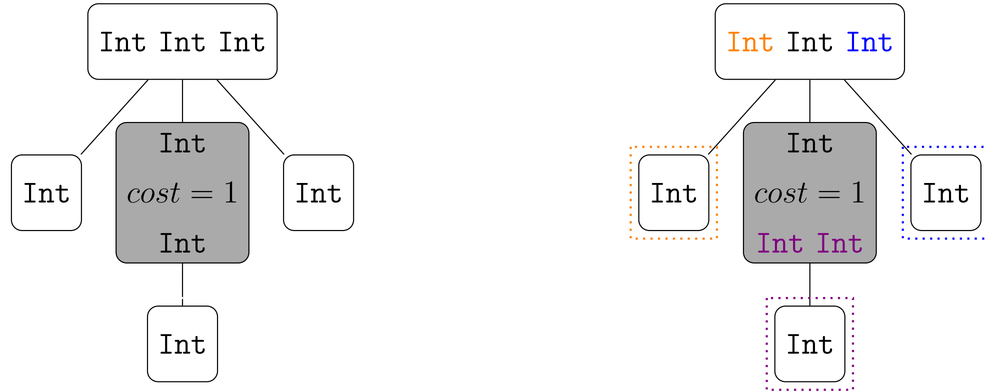
- One subproblem is created underneath the relabelled node. This subproblem is represented by the TPPT tp. l_{new}'s argument types become the root of tp. t's children become tp's children. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them purple.
- One subproblem is created to the right of the relabelled node. This subproblem is represented by the TPPT rt. rt's root node is a subvector of r's root node, from position k onward. rt's child nodes are the subvector of r's child nodes from position k onward. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them blue.
- One set of nodes must be type checked. This set of nodes can be represented by the TPPT tc. tc's root node is a subvector of r's root node, from position 0 to $k-1$. tc's child nodes are the subvector of r's child nodes from 0 to $k-1$. These nodes must be type checked before any subproblems are created, as if they do not type check, then none of the subproblem's solutions are valid. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them orange.

When we relabel a node at position k under the root, the node we relabel with must have a type that satisfies the kth node in the root of the start state TPPT. In Figure 6.30 we show an example of a relabel edit.

Delete

When a delete edit occurs, a single subproblem is created, and one set of nodes must be type checked. If the node t at position k is deleted under a root r, then the subproblem created and the nodes that need to be type checked can be described as follows:

- The subproblem created from a deletion is represented by the TPPT ds. r's root nodes from position k onward become the root nodes of ds. ds's child nodes can be described as the child nodes of t concatenated with the subvector



(a) A TPPT. This TPPT type checks and is considered the start state.

(b) The TPPT after a node has been relabelled in the TPPT shown in Figure 6.30a. It is a valid successor state and shows the two subproblems created, as well as the set of nodes that need to be type checked.

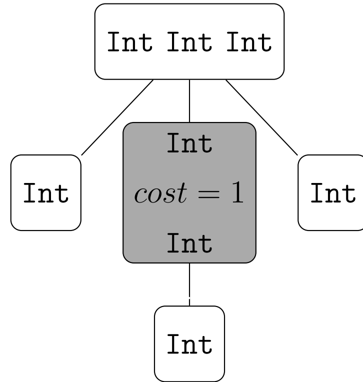
Figure 6.30: A start state TPPT and a valid successor state TPPT created by relabelling a node.

of r's children from position $k+1$ onward. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them blue.

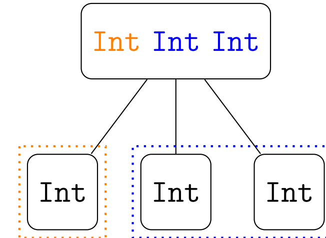
- One set of nodes must be type checked. This set of nodes can be represented by the TPPT tc. tc's root node is a subvector of r's root node, from position 0 to $k-1$. tc's child nodes are the subvector of r's child nodes from 0 to $k-1$. These nodes must be type checked before the subproblem is created, as if they do not type check, then none of the subproblem's solutions are valid. Where appropriate, we indicate the nodes and types that make up this TPPT by colouring them orange.

In Figure 6.31 we show an example of a deletion edit.

This concludes the explanations for each of the edits that can occur in an edit sequence. In the next subsection we show the pseudocode for the process to create the edit sequences, which follow from the examples shown here.



(a) A TPPT. This TPPT type checks and is considered the start state.



(b) The TPPT after a node has been deleted in the TPPT shown in Figure 6.31a. It is a valid successor state and shows the subproblem created, as well as the set of nodes that need to be type checked.

Figure 6.31: A start state TPPT and a valid successor state TPPT created by deleting a node.

6.6.4 Generating Edit Sequences

Through the examples in the previous subsections, we have explained conceptually how the algorithm to find end state TPPTs and edit sequences works. It can be described as a top-down DP algorithm, where the edit sequences found from a TPPT are stored for future use. In this subsection we show the pseudocode for this algorithm.

A broad outline of the algorithm can be given as:

- For a given start state TPPT, all possible edits are performed on that TPPT. For each edit, some subproblems will be created, and a set of nodes will need to be type checked.
- If the nodes that should type check do not, terminate this search derivation.
- If any of the created subproblems type check before any additional edits have been performed, return the empty edit sequence that is used to obtain them. Any additional edits to this subproblem would constitute a compound edit sequence, which we deal with separately in GenerateSuccessors.

Algorithm 6.12 CREATE-ALL-EDIT-SEQUENCES

Input: n The maximum cost of edits allowed.
Output: A map of start state TPPTs to sets of edit sequences.

```
algorithm CREATE-ALL-EDIT-SEQUENCES(n)
    tppts = GENERATE-TPPTS(n)
    rv = []                                          ▷ The map to be returned.
    map = []               ▷ The map of results from CREATE-EDIT-SEQUENCES.
    for (tppt ∈ tppts) do
        rv.INSERT(tppt, CREATE-EDIT-SEQUENCES(n, tppt, True, map))
    return rv
```

- Otherwise, search the subproblems represented as TPPTs. A set of edit sequences will be returned for each subproblem that corresponds to a type safe TPPT.

- Combine the edit sequences for each subproblem, moving back through the search until we reach the start state TPPT. All the edit sequences at this point will lead to end state TPPTs that type check.

- To be clear, the results for a given TPPT and n value are memoized. This is to ensure that if they are seen more than once then their results do not need to be re-computed.

The algorithm to create all edit sequences is called CREATE-ALL-EDITSEQUENCES. It is invoked from the constructor for NEIGHBOURHOOD-GENERATION. Its pseudocode, and the algorithms that it uses, are shown in Algorithms 6.12 to 6.16.

These algorithms use several functions we have not explained previously in this chapter. For clarity, we outline them as follows:

- CAN-INSERT. This function checks whether an insertion is allowed by checking the n value against the n_{max} value.

- GET-UNIFIED-TYPES. This function scans the language and returns any type signature and cost pair ts that is representative of a term in the language, which also meets the following criteria. Firstly ts's type's return type must unify with the given type argument t. Secondly ts's cost must be no greater than the given cost argument n.

Algorithm 6.13 CREATE-EDIT-SEQUENCES

Input:	n	The maximum cost of edits allowed.
	p	The input TPPT.
	startState	Boolean denoting whether this is a start state TPPT or not.
	map	Reference to map of previous results from calling CREATE-EDIT-SEQUENCES.
Output:		A collection of valid edit sequences that have a cost of at most n. Each edit sequence creates a valid end state TPPT.

```
algorithm CREATE-EDIT-SEQUENCES ALGORITHM(n, p, startState, map)
    if (¬map.CONTAINS({n, p, startState}) then
        if (TYPE-CHECK(p) ∧ startState = False) then
            return [[]]                    ▷ Return a single empty edit sequence.
        else
            res = []
            if (n > 0) then
                res.APPEND(CREATE-INSERTIONS(n, p, map))
                res.APPEND(CREATE-DELETIONS(n, p, map))
                res.APPEND(CREATE-RELABELS(n, p, map))
            map.INSERT({n, p, startState}, res)
    return map.AT({n, p, startState})
```

Algorithm 6.14 CREATE-INSERTIONS

Input:	n	The maximum cost of edits allowed.
	p	The input TPPT.
	map	Reference to map of previous results from calling CREATE-EDIT-SEQUENCES.
Output:		A collection of valid edit sequences that have a cost of at most n. Each edit sequence creates a valid end state TPPT.

```
algorithm CREATE-INSERTIONS(n, p, map)
    if (¬CAN-INSERT(n, p)) then
        return []
    rootTypes = p.NODE()
    children = p.CHILDREN()
    res = []
    for (pos ∈ {0 ... children.SIZE()}) do
        typesToInsert = GET-UNIFIED-TYPES(n, rootTypes[pos])
        if (¬TYPE-CHECK(rootTypes.SUB-VECTOR(0, pos − 1)) ∨
              typesToInsert = []) then
            continue
        for (args ∈ {0 ... children.SIZE() − pos − 1}) do
            takenChildren = children.SUB-VECTOR(pos, pos + args)
            remChildren = children.SUB-VECTOR(pos+args+1, children.SIZE())
            problem_1 = CREATE-TREE(rootTypes.SUB-VECTOR(pos + args + 1,
              rootTypes.SIZE()), remChildren)
            for (type ∈ typesToInsert) do
                newRootTypes = type.GET-ARGS()
                problem_2 = CREATE-TREE(newRootTypes, takenChildren)
                c = n−COST(type)
                res_1 = CREATE-EDIT-SEQUENCES(c, problem_1, False, map)
                res_2 = CREATE-EDIT-SEQUENCES(c, problem_2, False, map)
                allValidEdits = MAKE-VALID-EDIT-SEQUENCES(
                  n, (Insert, type, args, pos), res_1, res_2)
                res.APPEND(allValidEdits)
    return res
```

Algorithm 6.15 CREATE-RELABELS

Input:	n	The maximum cost of edits allowed.
	p	The input TPPT.
	map	Reference to map of previous results from calling CREATE-EDIT-SEQUENCES.
Output:		A collection of valid edit sequences that have a cost of at most n. Each edit sequence creates a valid end state TPPT.

```
algorithm CREATE-RELABELS(n, p, map)
    rootTypes = p.NODE()
    children = p.CHILDREN()
    res = []
    for (pos ∈ {0 ... children.SIZE() − 1}) do
        if (¬IS-TYPED-NAMED-TERM(children[pos])) then
            continue
        typesToRelabel = GET-UNIFIED-TYPES(n, rootTypes[pos])
        if (¬TYPE-CHECK(rootTypes.SUB-VECTOR(0, pos − 1)) ∨
              typesToRelabel = []) then
            continue
        remChildren = children.SUB-VECTOR(pos + 1, children.SIZE())
        problem_1 = CREATE-TREE(rootTypes.SUB-VECTOR(pos + 1,
          rootTypes.SIZE()), remChildren)
        for (type ∈ typesToRelabel) do
            newRootTypes = type.GET-ARGS()
            problem_2 = CREATE-TREE(newRootTypes, children[pos].CHILDREN())
            c = n−COST(type)
            res_1 = CREATE-EDIT-SEQUENCES(c, problem_1, False, map)
            res_2 = CREATE-EDIT-SEQUENCES(c, problem_2, False, map)
            allValidEdits = MAKE-VALID-EDIT-SEQUENCES(
              n, (Relabel, type, pos), res_1, res_2)
            res.APPEND(allValidEdits)
    return res
```

Algorithm 6.16 CREATE-DELETIONS

Input:	n	The maximum cost of edits allowed.
	p	The input TPPT.
	map	Reference to map of previous results from calling CREATE-EDIT-SEQUENCES.
Output:		A collection of valid edit sequences that have a cost of at most n. Each edit sequence creates a valid end state TPPT.

algorithm CREATE-DELETIONS(n, p, *map*)
  rootTypes = p.NODE()
  children = p.CHILDREN()
  res = []
  for ($pos \in \{0 \ldots children.\text{SIZE}() - 1\}$) **do**
    if (¬IS-TYPED-NAMED-TERM(*children*[*pos*])) **then**
      continue
    if (¬TYPE-CHECK(*rootTypes*.SUB-VECTOR(0, $pos - 1$)) **then**
      continue
    nodeCost = COST(*children*[*pos*])
    childrenOfDeleted = *children*[*pos*].CHILDREN()
    remChildren = *children*.SUB-VECTOR(*pos*, *children*.SIZE())
    allChildren = *childrenOfDeleted*.APPEND(*remChildren*)
    problem = CREATE-TREE(*rootTypes*.SUB-VECTOR(*pos*,
      rootTypes.SIZE()), *allChildren*)
    res = CREATE-EDIT-SEQUENCES($n - nodeCost$, *problem*, *False*, *map*)
    allValidEdits = MAKE-VALID-EDIT-SEQUENCES-SINGLE(
      n, (`Delete`, *pos*), *res*)
    res.APPEND(*allValidEdits*)
  return *res*

- Sub-Vector. This function takes the vector v and the indexes i and j and constructs a new vector containing the elements from i to j in v.
- Make-Valid-Edit-Sequences. This function takes a cost value n, a single edit e, and at most two vectors of edit sequences S_1 and S_2. It works as follows; first the Cartesian product $S = S_1 \times S_2$ is constructed. Then e is appended to every $s \in S$. Every sequence with cost $> n$ is removed from S. Finally the function returns S.
- Make-Valid-Edit-Sequences-Single. This function works identically to Make-Valid-Edit-Sequences, except that it only takes a single vector of edit sequences S, and doesn't construct a Cartesian product of edit sequences.

We feel that the algorithms and explanations presented in this section should be sufficient to understand how we generate all edit sequences. This is a computationally expensive algorithm. However, as we will see in the next subsection, the strategies employed ensure that it terminates in a reasonable amount of time - at least for the languages we use in this thesis.

6.6.5 Testing

In this subsection we provide data about the generation of edit sequences for Languages A and A1 using various n_{max} values. In Table 6.3 we show these results.

In this chapter we have not provided any information regarding the complexity of the algorithms presented. By analysing this data, it appears that the time taken and number of subproblems to solve grows exponentially in relation to the n_{max} value.

In Section 5.3.3 we presented evidence that larger neighbourhood bounds produced exponentially larger neighbourhoods, and we therefore chose to limit the neighbourhood bound to 3 in subsequent experiments in that chapter. In the data shown in Table 6.3, we can see that a larger neighbourhood bound drastically increases the time taken to find all relevant edit sequences for a language. If we were to perform the local search experiments with any higher bound than 3 on Language A, it would be difficult to justify the additional time taken to create the edit sequences required. Therefore, this data lends further justification to our choice of neighbourhood bound.

The results in Table 6.3 also shows us that the number of subproblems and start states created for an n_{max} value and either Language A and A1 are identical. This was expected due to the design of the algorithm. Succinctly, as it uses the

Table 6.3: Data gathered when calling CREATE-ALL-EDIT-SEQUENCES on different languages and n_{max} values. For each language (L), we show the number of functions (F) and terminals (T) in that language. In brackets are the number of terms with a unique type signature in each set. For each call to CREATE-ALL-EDIT-SEQUENCES we show the size of the set of TPPT start states, the total number of TPPTs for which all the edit sequences were found (we refer to these as problems solved), and the time taken to compute the edit sequences. Languages and A1 are described in Table 4.1.

L	F	T	n_{max}	Start States	Problems Solved	Time taken (ms)
A	9 (8)	26 (6)	1	134	433	10
			2	2,042	11,971	408
			3	38,016	526,630	29,685
			4	767,945	162,509,300	7,401,569
A1	9 (8)	27 (6)	1	134	433	14
			2	2,042	11,971	407
			3	38,016	526,630	28,639
			4	767,945	162,509,300	7,404,107

TPPT representation, it is the number of terms with a unique type in a language that effects how many start states created and subproblems solved when creating a neighbourhood's edit sequences.

In preliminary work when designing this algorithm, we attempted to use the PTPPT representation to search for edit sequences. However, we found that in this form the algorithm had to re-search what were essentially identical search derivations. In turn, this increased the overall computation time. It was this observation that directed us towards using the TPPT.

6.7 Discussions & Conclusions

In this chapter we have provided a thorough explanation of how the neighbourhood generation algorithm that we have designed works. Specifically, we have discussed our use-case of such an algorithm, the intuition behind the algorithm, and then described the two core components of the algorithm; the component which recognises patterns in the input program tree and applies edit sequences, and the component that searches for edit sequences.

We have taken great care throughout this chapter to describe the algorithms in terms of program synthesis and generic languages, rather than those specific languages used in this thesis. We believe that the methods described in this chapter could be utilised in other domains where program synthesis has found use, not just in automated heuristic creation. For example, patch generation [112] is an area where probing the neighbourhood of a program tree could prove to be an effective strategy in the overarching goal of automating the editing of programs. Much of the work in this area uses either generic, hand-written rules or evolutionary computation techniques such as GP when automatically editing a program. Our neighbourhood generation algorithm could provide an alternate methodology in this domain. The hand-written rules in patch generation are usually generic schema designed to edit a program fragment in a small way. Neighbourhood generation and local search program synthesis could be used as a method of automating this creation of rules, allowing for further generalisation without the need for a human expert.

Some of the work in this chapter raises additional questions relating to the experiments described in Chapter 5. In that chapter we considered different neighbourhoods according to various neighbourhood bounds. We also discussed how we considered

the program synthesis method we were simulating to be analogous to local search. However, the use of compound moves in Section 6.4.7 suggests that our definition of a neighbourhood in Definition 23 could be refined to include additional parameters that allow us to more accurately describe the set of moves that describe a neighbourhood. To that end, we consider the following revision to the definition of a neighbourhood.

Definition 26 (Neighbourhood of a Program Tree (Revised))
Given a language L, a candidate program tree h, an integer representing the maximum bound of a single edit n_s, two integers representing the minimum and maximum number of compound moves n_{m_0} and n_m and an integer representing the sum of the costs that all edits can have n_{sum}, the revised neighbourhood $N_r(h, n_s, n_{m_0}, n_m, n_{sum})$ is defined as all valid program trees under L that can be obtained from h through a series of sequences of tree edits with the following properties. Each sequence of tree edits describes a transformation between two valid program trees under L. Each sequence of tree edits has a cost of at most n_s. We assume that each edit has a cost of 1. There are between n_{m_0} and n_m sequences of tree edits. The total sum of the cost of all sequences of tree edits is at most n_{sum}. If the program tree is obvious from the context, we write $N_r(n_s, n_{m_0}, n_m, n_{sum})$.

Any neighbourhood defined using Definition 23 as $N(n)$ can be redefined using Definition 26 as $N_r(n, 1, n, n)$.

This revised definition tells us that in the experiments described in Chapter 5, the neighbourhoods could be considered to be composed of the application of multiple smaller edit sequences. We believe that the local search algorithms used as a basis for those experiments can be more appropriately described as variable-depth search (VDS) [83, Chapter 2] algorithms. Algorithms based on VDS have been found to be highly effective in certain domains - for example, the Lin-Kerringham algorithm for TSP can be considered a VDS algorithm [111]. We do not perform any experiments in this thesis that consider using a neighbourhood only comprised of program trees obtained through the application of single edit sequences. However, in future work we believe it may be worth exploring the role that the use of compound moves has on the quality of results obtained from local search program synthesis.

Regarding the definition of our algorithms, there are several potential improvements that could be made to the algorithm that generates all edit sequences which could reduce the overall computation time. For starters, our algorithms do not make use of multithreading. It would be simple to augment our algorithms to do so, by

having each thread search a single start state TPPT. Secondly, we are aware of some TPPTs that are encountered in the search for edit sequences which are similar to others. We believe it may be possible to exploit these similarities to reduce overall computation. For example, consider a language L with a principle type a, and two start state TPPTs in the form $[a]\{a\}$ and $[a, a]\{a, a\}$. In this notation, each child node of each TPPT is a typed node. We believe that many of the results from calculating the edit sequences for $[a]\{a\}$ could be reused when calculating the edit sequences for $[a, a]\{a, a\}$.

We also believe there may potentially be a less computationally expensive method of generating all edit sequences. When we are generating the set of start state TPPTs, we noticed that they were all used as an end state TPPT for some other start state TPPT. Rather than performing any search on the start state TPPTs, it may be less computationally expensive to use a MTED algorithm to compare all start state TPPTs to each other, find those within the bounds stipulated by the input parameters, and then generate the edit sequences between them. At one point in our research, we experimented with this approach. However, we found the created algorithm to be more computationally expensive than our final algorithm. In future work, it may be worth revisiting this approach with the goal of finding methods to reduce its computational overhead - for example, through the application of more sophisticated MTED algorithms.

In the next chapter we use the neighbourhood generation algorithm described in this chapter to perform local search on the space of LS-SAT heuristics. As we now have a method of generating a neighbourhood without requiring memoized results, we can evaluate local search as a method of program synthesis without being constrained by the size of the program tree.

Chapter 7

Local Search Program Synthesis

7.1 Introduction

In Chapter 5 we performed experiments simulating local search on a subset of the space of heuristics described by Language A. In those simulations we used the memoized results from the experiments performed in Chapter 4 to create a heuristic's neighbourhood. Creating the neighbourhoods in this way was a computationally expensive calculation. The neighbourhoods were constrained in that they could only contain heuristics that had been memoized. In turn, this limited which heuristics could be created from the overarching local search algorithm.

In Chapter 6 we described algorithms that can be used to build the neighbourhood of a program tree without requiring any memoized data. In this chapter we will use those neighbourhood generation algorithms to automatically create heuristics through local search. We will perform experiments evaluating this local search-based method of program synthesis, which in Section 5.5 we termed local search program synthesis (LSPS). The goal of this chapter is to determine whether LSPS is a viable technique for automatically creating LS-SAT heuristics.

The format of this chapter is as follows. In Section 7.2 we present a series of experiments that use LSPS to create LS-SAT heuristics using Language A. These experiments are designed as a series, where each subsequent experiment changes a single parameter compared to the previous. In Section 7.3 we present an LSPS experiment that uses the randomised neighbourhood generation algorithm described in Section 6.5. In Section 7.4 we present an LSPS experiment that uses an alternate definition of a neighbourhood. In Section 7.5 we present a series of program synthesis

experiments that use an alternate language, Language B. In Section 7.6 we present some of the heuristics created from the experiments described in this chapter. Finally in Section 7.7 we present our discussions and conclusions from the research described in this chapter.

7.2 Initial Experiments

In this section we present the methodology and results from four experiments. Each experiment can broadly be described as a set of repetitions of LSPS. The first experiment mimics one of the simulated experiments in Chapter 5. Each subsequent experiment changes a single parameter when compared to the previous. The final experiment in the series is designed to use no memoized data, and places no limits on the size of the heuristics that can be created.

The format of this section is as follows. In Section 7.2.1 we present the methodology and results from Experiments A1 and A2, two LSPS experiments which impose a bound on how large the created heuristics can be. In Section 7.2.2 we present the methodology and results from Experiments A3 and A4, two LSPS experiments which place no bound on the size of the created heuristics. Finally in Section 7.2.3 we provide a summary of the work undertaken in this section.

7.2.1 Bounded Experiments

In this subsection we describe the methodology and present the results from Experiments A1 and A2.

Methodology

Experiments A1 and A2 consist of a set of repetitions of an LSPS algorithm. Each experiment creates LS-SAT heuristics using Language A. Each experiment's underlying LSPS algorithm is based on LOCAL-SEARCH-RND (see Algorithm 5.2). Each repetition is initialised by randomly choosing a heuristic from Language A with size ≤ 15. We use the memoized results from Chapter 4 to perform this step. The neighbourhood generation algorithm GENERATESUCCESSORS is used to create the entire neighbourhood for each candidate heuristic. The neighbourhood defined as $N(3)$ is used. This neighbourhood can also be described as $N_r(3, 1, 3, 3)$ in terms of Definition 26. Each neighbourhood is filtered after it is created. In Experiment A1

all heuristics with size > 15 are removed, and in Experiment A2 all heuristics with size > 17 are removed.

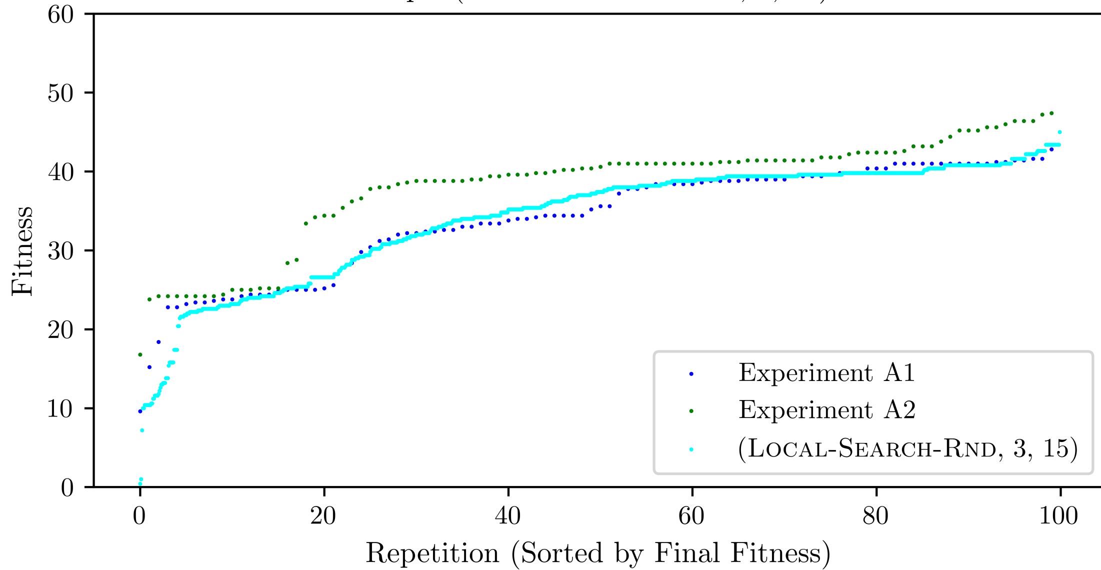
The heuristics are evaluated using the fitness function described in Section 3.4.1. Termination occurs either after all heuristics in the current neighbourhood have been evaluated and no fitter heuristic found, or after $100,000$ total heuristic evaluations have been performed. The heuristic with the highest fitness is returned from the algorithm. We call this returned heuristic the repetition's final heuristic. Both experiments consist of 100 repetitions. Experiments A1 and A2 were ran on the same system described in Section 4.2.2. Experiment A1 took 9 hours to terminate, and Experiment A2 took 13 hours.

Results & Analysis

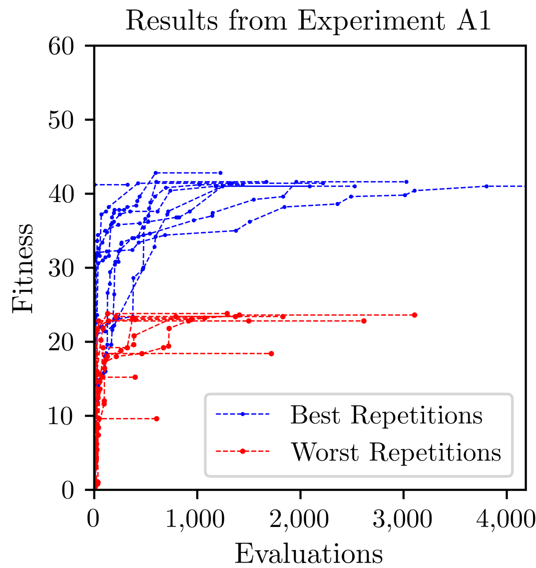
In Figure 7.1 we present three graphs; one shows the quality of the heuristics created from each repetition of each experiment, and the other two show how ten of the "best" and "worst" repetitions from each experiment progressed. We define the best and worst repetitions as those whose final heuristic had the highest and lowest fitness values respectively. In Table 7.1 we show additional statistical data pertaining to each experiment.

Throughout this chapter we compare the distributions of fitness values obtained from various experiments. When we refer to an experiment's distribution of fitness values, we specifically refer to the set of final heuristic's fitness values, derived from each repetition of the experiment. In this chapter we present this data in graphical form, as can be seen in Figure 7.1a. In these graphs we order the fitness values from lowest to highest. By visualising the fitness values in this way, we can provide detailed insight into the quality of the heuristics created by a specific experiment.

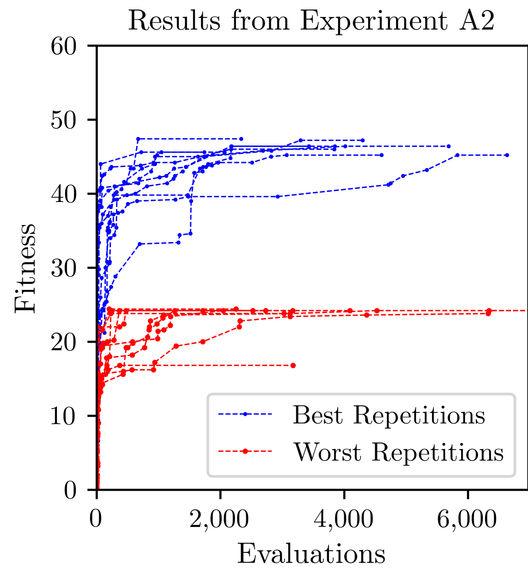
We can see that the heuristics returned from Experiment A2 have a higher fitness than those returned from Experiment A1. This is clear from the data presented in Figure 7.1a and the statistics shown in Table 7.1. This result was expected, as Experiment A1 considers a subset of the heuristics under consideration in Experiment A2. A heuristic that is a local optima under Experiment A1's algorithm may have a fitter neighbour under Experiment A2's algorithm. Under Experiment A2's algorithm, this fitter neighbour would become the candidate heuristic, which may also have fitter neighbours. On average over all repetitions, we can see that this larger search space yielded fitter heuristics.



(a) Final results from Experiments A1 and A2. Each data point represents the fitness of the final heuristic returned from that repetition of the local search algorithm. We also show an overlay of the results from the simulated local search experiment described by the triple (Local-Search-Rnd, 3, 15), originally shown in Figure 5.5c.



(b) Ten of the best and worst repetitions from Experiment A1.



(c) Ten of the best and worst repetitions from Experiment A2.

Figure 7.1: Results from Experiments A1 and A2. In Figures 7.1b and 7.1c each sequence of data points represents a single repetition.

Table 7.1: Statistical data from Experiments A1 and A2. We show the steps (number of times the candidate heuristic changes), the number of evaluations, the start, end and difference in size (s) and fitness (f) of the initial (h_1) and final heuristic (h_{final}), the MTED and SMTED between the first and final heuristic, and the number of evaluations required to find the final heuristic.

(a) Statistical data from Experiment A1.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	13.19	4	10.00	13.00	16.00	30
Evals	1,882.68	191	1,222.50	1,719.00	2,369.75	5,575
$f(h_1)$	2.65	0.00	0.00	0.50	2.06	35.80
$f(h_{final})$	34.01	9.60	30.25	35.40	39.45	42.80
Δ_f	31.36	3.20	24.85	33.70	38.60	41.40
Evals to h_{final}	870.11	3	379.00	618.50	1,126.25	4,293
$s(h_1)$	14.59	11	14.00	15.00	15.00	15
$s(h_{final})$	14.40	11	14.00	15.00	15.00	15
Δ_s	-0.19	-3	-1.00	0.00	0.00	3
MTED(h_1,h_{final})	7.51	2	6.00	7.00	9.00	12
SMTED(h_1,h_{final})	1.71	0	0.00	2.00	3.00	6

(b) Statistical data from Experiment A2.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	15.40	7	12.00	15.00	18.00	29
Evals	2,896.52	899	1,993.50	2,550.00	3,414.25	9,137
$f(h_1)$	3.25	0.00	0.00	0.60	3.55	37.20
$f(h_{final})$	38.01	16.80	37.50	40.50	41.80	47.40
Δ_f	34.76	3.80	27.65	38.50	41.00	46.40
Evals to h_{final}	1,470.94	68	533.25	1,186.50	2,063.25	6,339
$s(h_1)$	14.61	12	14.00	15.00	15.00	15
$s(h_{final})$	15.54	12	14.00	16.00	17.00	17
Δ_s	0.93	-2	0.00	1.00	2.00	4
MTED(h_1,h_{final})	9.43	5	8.00	9.00	11.00	15
SMTED(h_1,h_{final})	2.67	0	1.00	2.00	4.00	8

In Figure 7.1a we show an overlay of the results obtained from one of the simulated local search experiments described in Chapter 5. Specifically this is the simulation described by the triple (LOCAL-SEARCH-RND, 3, 15). This simulation was chosen as its methodology is similar to that used for Experiment A1. The overlay is extrapolated from the 1,000 repetitions of the simulation. We have condensed the data points so that the distribution of fitness values can be easily compared to that obtained from Experiment A1. We can see that the two experiment's fitness distributions are remarkably similar. The only difference between Experiment A1 and the simulated experiment is that the GENERATESUCCESSORS algorithm is used instead of the computationally expensive MEMOIZE-NEIGHBOURHOODS algorithm. As the experiments produced similar results, we believe that this provides strong evidence that GENERATESUCCESSORS achieves its design goals - it emulates the algorithm MEMOIZE-NEIGHBOURHOODS.

We can also see a similarity between the two distributions of fitness values obtained from Experiments A1 and A2. We would describe both distributions as plateauing at various points, before tailing upward to the next plateau. Both distributions tail upward at the highest fitness values. Like in the simulated experiments, Experiments A1 and A2 produced duplicate final heuristics. In Experiment A1 there were 79 unique final heuristics reported from all repetitions, and 80 for Experiment A2. In comparison to the results reported in Section 5.4, these proportions are quite high. However, without more data it is difficult to draw any conclusion as to whether these are typical values for the number of repetitions performed.

In Figures 7.1b and 7.1c we show how some of the repetitions of each experiment progressed. What is most striking from these examples is how quickly high-quality heuristics were created. For the majority of repetitions considered, the final heuristic was created in under 5,000 evaluations. While these repetitions did not terminate at this point, they did not create any better heuristics - in essence, they had arrived at the local optimum.

The results shown in Table 7.1 provide us with data to reinforce these observations, and afford us the opportunity to make further ones. We can see that all repetitions in both experiments terminated after less than 10,000 heuristic evaluations. To find the local optima in all repetitions required under 6,500 heuristic evaluations. However, if we look at the other statistical data, we can see that there are some outliers - both very high and very low values - that skews this evaluation data significantly. By

analysing the mean data from Experiment A2, we can see that on average just under 1,500 evaluations are required to find the local optima, and on average under 3,000 evaluations required for the repetitions to terminate. The data from Experiment A1 reported these values as under 900 and under 2,000 evaluations respectively. The large difference between these values was expected. This is because any given heuristic p will have at least as many neighbours under the LSPS algorithm described by Experiment A2 as that described by Experiment A1. Therefore, to explore p's entire neighbourhood, the LSPS algorithm described by Experiment A2 will require more heuristic evaluations than that described by Experiment A1.

In Table 7.1 we include data showing how the structure of the final heuristic changes compared to the initial heuristic. We include three metrics; the size of the heuristic, the MTED between the initial and the final heuristic, and the structured MTED (SMTED) between the initial and final heuristic. The SMTED is a form of MTED we describe as follows; given two trees a and b, the SMTED between a and b is computed as the MTED between $strip(a)$ and $strip(b)$. The function $strip$ removes all labels from a tree. The SMTED metric provides a measurement of how different the structure of two trees are to each other, while ignoring how the nodes are labelled.

We can see from the structural data that in both experiments there are examples of heuristics that grow and shrink in size, and whose MTED and SMTED between the initial and final heuristic is large. In essence as LSPS is creating effective heuristics, it is also fundamentally changing the structure of the program trees as it moves through the search space, and not necessarily just making small changes such as relabelling nodes. We note that in Experiment A2 the values regarding the structural changes are greater than those seen in Experiment A1.

In Table 7.1a, which details the statistical data from Experiment A1, we can see that the average change in the size of the heuristics is negative. Though this may seem unintuitive, when we consider the experiment's methodology, we can see why this is correct. To initialise the first heuristic, the experiment is picking from all heuristics in Language A of size ≤ 15. From the data in Table 4.2, we know that the vast majority of the chosen heuristics will be of size 15. Therefore, it would only take a small number of repetitions which have a final heuristic of size < 15 for the average change in size to become negative.

In the next subsection we present two experiments which are similar to those

described here, except that there is no limit placed on the size of the heuristics which can inhabit a neighbourhood.

7.2.2 Unbounded Experiments

In this subsection we describe the methodology and present the results from Experiments A3 and A4.

Methodology

Experiments A3 and A4 are nearly identical to Experiments A1 and A2. Instead of providing a complete description of their methodology, we give a brief description of their differences compared to other experiments.

- Experiment A3 is formulated in exactly the same manner as Experiments A1 and A2, except that the neighbourhood is not filtered after it is generated.
- Experiment A4 is formulated in exactly the same way as Experiment A3, except that the initial heuristic is created in a different way. The grow method from GP (see Section 2.5.4) is used, with a maximum depth of 4.

We consider Experiments A1 and A2 to be the realisation of the simulated experiments described in Chapter 5. The experiments described here are unconstrained versions of those experiments. In Section 4.2.1 we discussed how Language A describes a search space containing an infinite number of heuristics. As Experiments A3 and A4 do not enforce any constraints on which heuristics can inhabit a neighbourhood, they are - at least theoretically - able to navigate to any heuristic in the infinite search space described by Language A.

Experiment A4 removes the reliance that Experiments A1 to A3 had on an initialisation function that is difficult to implement without memoized data. In those experiments, we chose a heuristic with size ≤ 15 by using the enumerated data from Chapter 4. In a real-world setting this data would not be available. In addition to this, using this method of initialisation may have introduced unintended bias into the algorithm, as, in the set of heuristics with size ≤ 15, the vast majority of elements have a size of exactly 15 - therefore, heuristics of size 15 are highly likely to be chosen. For these reasons, we felt it prudent to use an alternate initialisation function with Experiment A4.

Experiments A3 and A4 were ran on the same system described in Section 4.2.2. In total Experiment A3 took 24 hours to terminate, and Experiment A4 61 hours.

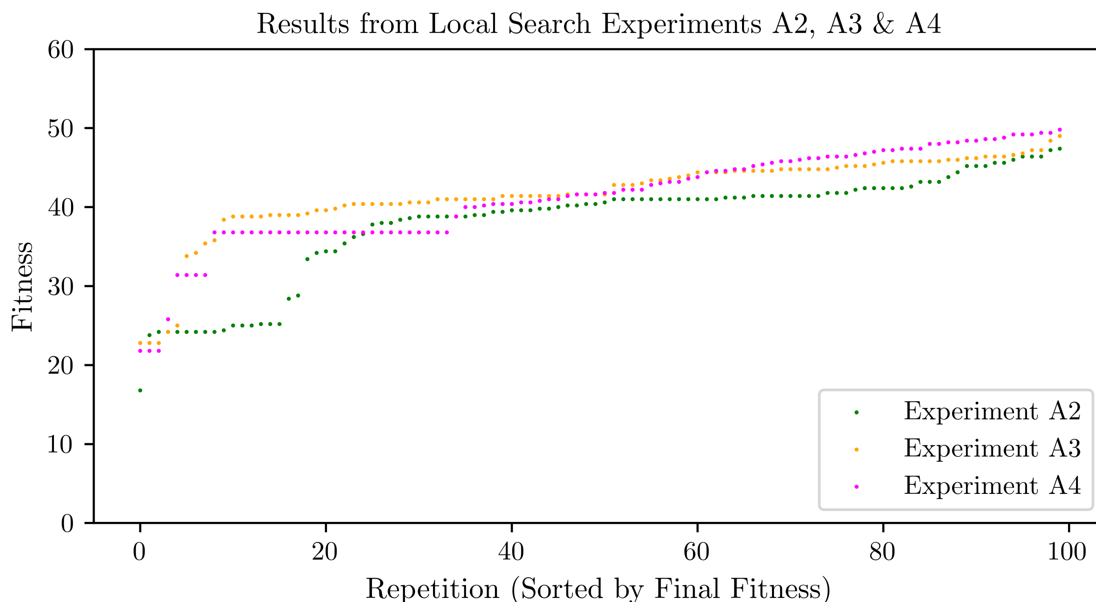
Results & Analysis

In Figure 7.2 we present three graphs; one shows the quality of the heuristics created from each repetition of each experiment, and the other two show how ten of the best and worst repetitions from each experiment progressed. In Table 7.2 we show additional statistical data pertaining to each experiment.

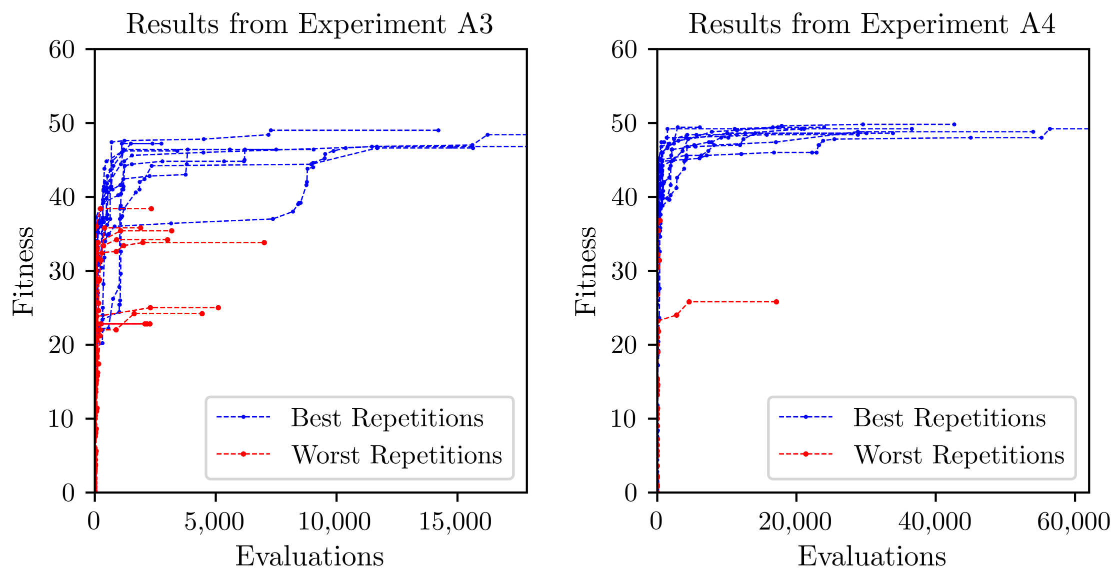
We can see from the data in Figure 7.2a that the heuristics created from Experiments A3 and A4 are generally of higher quality than those created from Experiment A2. The highest quality heuristics created also have a higher fitness than the highest quality heuristics created from exhaustive enumeration. We can see that the distribution of fitness values from both Experiments A3 and A4 is very different to the distribution from Experiment A2. In Experiment A2, there is a relatively flat plateau between fitness scores 20 and 30 encompassing the first 20 heuristics. The distribution then tails upward to the next plateau. In Experiments A3 and A4 the plateau between fitness scores 20 and 30 is much smaller in size. Experiment A4 has a large plateau between fitness scores 30 and 40 which does not appear in the other experiments. After these plateaus, each experiment's fitness distribution has an upward trajectory.

As observed previously, there is a large plateau in Experiment A4's distribution between fitness scores 30 and 40. Unlike the other plateaus we have seen, it appears to be flat. The repetitions which make up this plateau - from index 9 to 34 - all returned one of two heuristics. These heuristics had nearly identical fitness scores, which made the plateau appear flat. The number of unique heuristics returned from Experiment A3 was 82 and from Experiment A4 70. This tells us that the vast majority of the duplicate results returned from Experiment A4 are contained in this plateau.

The only difference between Experiments A3 and A4 is in the way that their local search algorithm is initialised. Experiment A3 initialises its first heuristic by picking one randomly that has size ≤ 15, while Experiment A4 uses the grow method from GP. To help us understand why these plateaus have appeared, in Figure 7.3 we present the results from Experiment A4 with additional information about the size of the initial heuristic used in that repetition. From this data, we can see that



(a) Final results from Experiments A3 and A4. Each data point represents the fitness of the final heuristic returned from that repetition of the local search algorithm. We also show the results from Experiment A2 for comparison.



(b) Ten of the best and worst repetitions from Experiment A3.

(c) Ten of the best and worst repetitions from Experiment A4.

Figure 7.2: Results from Experiments A3 and A4. In Figures 7.2b and 7.2c each sequence of data points represents a single repetition.

Table 7.2: Statistical data from Experiments A3 and A4. We show the steps (number of times the candidate heuristic changes), the number of evaluations, the start, end and difference in size (s) and fitness (f) of the initial (h_1) and final heuristic (h_{final}), the MTED and SMTED between the first and final heuristic, and the number of evaluations required to find the final heuristic.

(a) Statistical data from Experiment A3.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	19.02	8	14.00	18.00	22.25	42
Evals	5,343.31	1,184	2,524.25	3,454.00	6,739.25	24,733
$f(h_1)$	2.95	0.00	0.00	0.40	3.75	32.20
$f(h_{final})$	41.68	22.80	40.40	41.60	44.85	49.00
Δ_f	38.73	8.80	36.80	40.80	44.20	49.00
Evals to h_{final}	2,691.99	51	1,003.50	1,478.50	3,442.00	16,242
$s(h_1)$	14.50	11	14.00	15.00	15.00	15
$s(h_{final})$	17.61	13	14.00	18.00	20.00	30
Δ_s	3.11	-2	0.00	3.00	5.00	16
MTED(h_1,h_{final})	11.56	4	9.00	11.00	14.00	25
SMTED(h_1,h_{final})	4.43	0	2.00	4.00	6.00	16

(b) Statistical data from Experiment A4.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	17.86	4	10.00	16.50	24.00	43
Evals	13,481.60	151	327.75	5,578.50	22,000.75	85,500
$f(h_1)$	2.30	0.00	0.00	0.80	2.80	25.40
$f(h_{final})$	41.31	21.80	36.80	41.70	46.40	49.80
Δ_f	39.00	20.79	35.79	39.40	44.75	49.80
Evals to h_{final}	6,182.75	9	187.75	1,748.00	6,608.50	59,740
$s(h_1)$	16.03	2	3.00	17.00	26.00	40
$s(h_{final})$	19.93	7	8.00	19.50	30.00	41
Δ_s	3.90	-2	1.00	4.00	5.00	17
MTED(h_1,h_{final})	12.05	3	6.00	12.00	17.25	26
SMTED(h_1,h_{final})	4.90	0	3.00	4.50	6.00	17

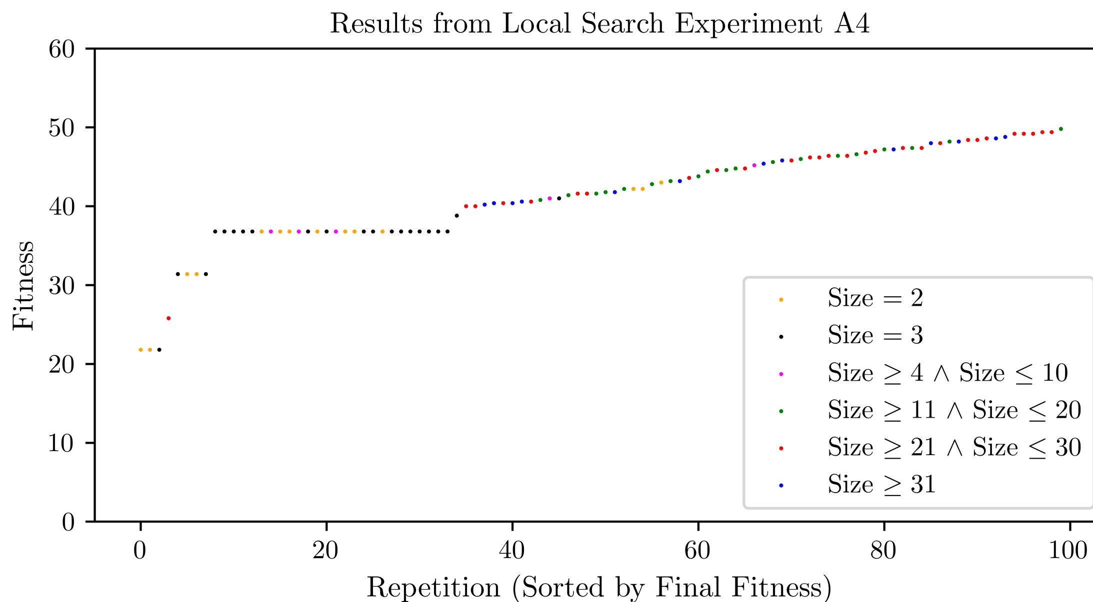


Figure 7.3: Final results from Experiment A4. Each data point represents the fitness of the final heuristic returned from that repetition of the local search algorithm. The data points are coloured according to the size of the initial heuristic of that repetition.

the majority of the heuristics in the plateaus come from repetitions which use initial heuristics with a very small size - either 2 or 3. It appears that when these small heuristics are used as a starting point, it is more likely that the final heuristic arrived at will be one of a small number of heuristics. From here, the search is evidently unable to progress further, making these plateaus appear.

The grow method to create program trees works by randomly picking a term - either a function or a terminal - then recursively populating the child nodes of that term. This process continues until the depth from the root to the current node being assigned is greater than the given depth parameter, then only terminals can be chosen. In this way, a full program tree is created. Language A has certain functions that, if chosen at any point when creating a program tree, will constrain the size of that program tree. For example, if `GetBestVar` is chosen then none of its child nodes can be functions. If it is chosen as the root of a created program tree, then, irrespective of the depth parameter used, the created program tree will be of size 3. We believe that it is due to the way in which the language is formulated that causes these small heuristics to be created, which in turn limits the effectiveness of LSPS on these repetitions.

If we look at the graphs regarding the best and worst repetitions from Experiments A3 (Figure 7.2b) and A4 (Figure 7.2c), we can see that for the small number of results considered, the number of evaluations performed far exceeds those values reported from Experiments A1 and A2 - with the exception of the worst repetitions from Experiment A4. If we consider only the best repetitions, we can see that in creating the final heuristics from Experiment A4, the number of evaluations performed far exceeds that reported from Experiment A3. We know that the largest initial heuristic used in Experiments A1 to A3 was size 15. From the data shown in Figure 7.3, we know that the size of the initial heuristics used for the best repetitions was much larger than 15, with some heuristics having an initial size ≥ 31. If the results in Section 5.3.2 are indicative for all program trees in Language A, then it stands to reason that heuristics at this size will have a far greater number of neighbours.

In Figure 7.2c we can see that, for Experiment A4, nearly all of the worst repetitions use a small number of evaluations. These data points are difficult to discern compared to those for the best repetitions. Upon further analysis, we found that all of the worst repetitions - apart from one - terminated in under 200 evaluations. We know from Figure 7.3 that the majority of these repetitions began with an initial heuristic that had a small size. Together, this tells us that those repetitions which began with a small initial heuristic found their final heuristic in a small number of evaluations.

The data in Table 7.2 shows us more detailed information regarding Experiments A3 and A4. The number of evaluations for each of the experiments far exceeds those reported from Experiments A1 and A2, with the average number of evaluations over all runs being reported at just under $5,500$ for Experiment A3 and just under $13,500$ for Experiment A4. We can see that these averages have been somewhat skewed by outliers in both experiments, however the effect is more pronounced for Experiment A4. This is illustrated by the median, and the lower and upper quartile values. This skewing of data is also seen in the values reported for the number of evaluations required to obtain the final heuristics in both experiments.

We can see that the change in the size, MTED and SMTED values reported from Experiments A3 and A4 exceed those values reported from Experiments A1 and A2. However, we are somewhat surprised that these values are not higher, and that we did not see examples of final heuristics that had much larger changes compared to their starting heuristic. The results for both experiments are relatively similar to

each other. We feel in future work that it may be beneficial to investigate the role that the size of the starting heuristic has on how the heuristic's structure changes through a repetition of LSPS. This may help provide insight into how to design more appropriate initialisation algorithms for this method of program synthesis.

The experiments described in this subsection illustrate to the reader how LSPS can be used to automatically create heuristics for our domain, without any constraints placed on the size of the created heuristics, and (for Experiment A4) no memoized data required.

7.2.3 Summary

In this section we have performed several experiments to understand how LSPS performs when used to create LS-SAT heuristics. The initial experiments can be considered a realisation of the simulated experiments described in Chapter 5, while Experiments A3 and A4 are able to navigate the entire search space of heuristics as described by Language A. We have shown how these experiments have created many high-quality heuristics using far less evaluations than the exhaustive enumeration and GP experiments described in Chapter 4. While the LSPS experiments described in this section were unable to produce heuristics with a higher fitness than those created from GP, they did create fitter heuristics than those created from exhaustive enumeration. Unlike those created from GP, the heuristics created from LSPS are human readable and relatively small. While there was a tendency for the heuristics to grow in size, this growth was far smaller than that reported from GP.

In the next section we perform a further LSPS experiment which uses the randomised neighbourhood generation algorithm described in Section 6.5.

7.3 Randomised Neighbourhood Generation

In the previous section we performed a series of experiments using LSPS and Language A to create LS-SAT heuristics. We began with Experiment A1, which emulated a simulated local search experiment described in Chapter 5. Each subsequent experiment relaxed a constraint imposed by the previous. This process culminated in Experiment A4, which required no memoized data, and imposed no constraints on the size of the created heuristics.

The overarching goal of this section is to perform an experiment that is designed

to emulate Experiment A4. This experiment, designated Experiment A5, will use the function GenerateSuccessors-Rnd (see Algorithm 2.5) to generate a heuristic's neighbours. Experiments A1 to A4 used the function GenerateSuccessors to do this. GenerateSuccessors generates the entire neighbourhood of a heuristic, whereas GenerateSuccessors-Rnd randomly generates a single neighbour. We want to analyse the difference between the results from Experiments A4 and A5, to determine how effective GenerateSuccessors-Rnd is compared to Generate-Successors.

The format of this section is as follows. In Section 7.3.1 we discuss the motivations behind the use of GenerateSuccessors-Rnd. We also discuss two issues that the use of GenerateSuccessors-Rnd introduces when used as part of an overarching LSPS algorithm. In Section 7.3.2 we describe two mechanisms to be used alongside GenerateSuccessors-Rnd, which are designed to alleviate the two issues highlighted in Section 7.3.1. In Section 7.3.3 we present Experiment A5, an LSPS experiment that uses GenerateSuccessors-Rnd. We also present our analysis on the results from that experiment. Finally in Section 7.3.4 we provide a summary of the work undertaken in this section.

7.3.1 Observations

The local search algorithms used in the simulated experiments described in Chapter 4, and those experiments described in Section 7.2, have a similar format. That is to say, they work by generating the entire neighbourhood of a candidate heuristic, then choosing heuristics randomly from that neighbourhood. To be clear, the heuristics are removed from the neighbourhood after they have been chosen. This process continues until either a fitter heuristic is found and made the new candidate, or the neighbourhood is empty and the entire algorithm terminates. This mechanism acts as a form of "early termination", and is used as the algorithm cannot progress any further from its current position. It is distinctly separate from how the local search algorithm would usually terminate - when the allocated maximum number of evaluations had been reached.

In general, local search is commonly associated with problems that represent their candidate solutions as fixed-sized arrays. Each element in the array usually represents some variable in the problem, and has a domain of possible values associated with it. For a specific problem instance, all candidate solutions are the same size. A common

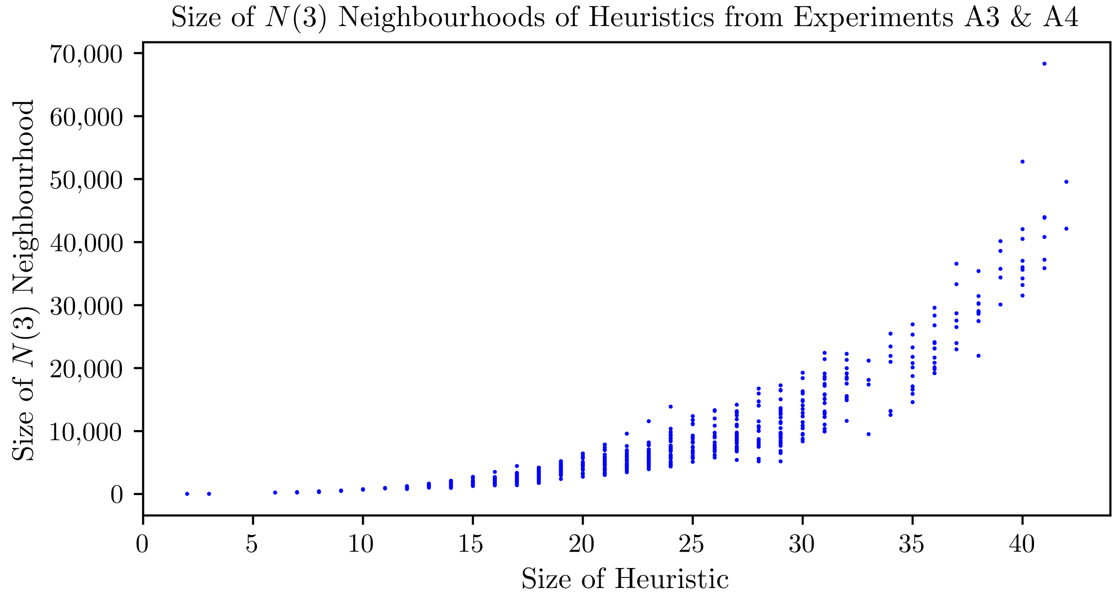


Figure 7.4: Size of the neighbourhoods of the candidate heuristics from all repetitions of Experiments A3 and A4. The neighbourhoods can be described by $N(3)$.

definition of a neighbourhood used with a fixed-sized array is that which allows any variable's value to be changed for any other valid value. Under this definition, all candidate solutions have the same number of neighbours.

Our domain does not have these characteristics. We have seen invocations of local search where the size of the candidate solution (in our domain, these are program trees or heuristics) changes. Using data collected from Experiments A3 and A4, in Figure 7.4 we present data showing the size of the $N(3)$ neighbourhood for different sized heuristics. This data suggests that the size of a heuristic's neighbourhood grows exponentially in relation to the size of the underlying program tree - at least for Language A and $N(3)$. It also shows us that all heuristics of size n do not have the same number of neighbours. In Section 5.3.2 we presented data showing this trend, however not for heuristics of such a large size.

Up to this point, the experiments described in this chapter have used the function GenerateSuccessors to generate the neighbourhood of a candidate heuristic. Assuming that the trends seen in Figure 7.4 continue, at some point we would encounter a large enough heuristic whose neighbourhood we would be unable to store, due to the computational resources required to do so. In addition to this, if we were to consider neighbourhoods described by larger neighbourhood bounds, or

neighbourhoods described by different languages, we may encounter the same problem. In Section 7.4 we perform experiments using a different definition of a neighbourhood, and in Section 7.5 we perform experiments using a larger language. In preliminary work, we attempted to use GenerateSuccessors to perform those experiments, however we found it was not possible to store the neighbourhoods created using the computational resources available to us.

In Section 6.5 we presented a randomised neighbourhood generation algorithm called GenerateSuccessors-Rnd. This algorithm works by creating an individual randomly instead of generating all neighbours. Its major advantage over GenerateSuccessors is that it can create neighbours without having to generate an entire neighbourhood. This allows us to use an arbitrarily large program tree as the candidate while using far less computational resources.

However, GenerateSuccessors-Rnd has two major disadvantages compared to GenerateSuccessors. The first of these is to do with its role in a local search algorithm. GenerateSuccessors conceptually returns the set of unique neighbours. When used as part of an LSPS algorithm, all the neighbours can be evaluated individually, and the algorithm can terminate early if no fitter neighbour found. All repetitions in Experiments A1 to A4 used this early termination mechanism. GenerateSuccessors-Rnd may return the same neighbour more than once, and cannot inform the wider local search algorithm when all neighbours have been evaluated. Therefore, if we were to substitute it for GenerateSuccessors, an invocation of local search using it would only terminate when the maximum number of evaluations had been reached. This would be undesirable for many repetitions of Experiments A1 to A4, where the heuristic that was returned was found quickly.

The second disadvantage is a more nuanced issue regarding the way in which a neighbour is chosen. If we consider a neighbourhood of size k, when choosing an element randomly from that neighbourhood, the chance that any neighbour would be chosen is $\frac{1}{k}$. It is not simple to instantiate GenerateSuccessors-Rnd in such a way so as to provide the same chance that a neighbour would be chosen.

There are several reasons for this, however we will concentrate on how the use of compound moves makes this difficult. In Section 6.7 we discussed how the simulated experiments we performed in Chapter 5 could be considered a form of VDS. In Definition 26 we introduced an alternate definition for a neighbourhood that allowed us to write $N(n)$ as $N_r(n, 1, n, n)$. Using set notation, we observe that this can be

rewritten as shown in Equation (7.1).

$$N_r(n, 1, n, n) = N_r(n, 1, 1, n) + N_r(n, 2, 2, n) + \cdots + N_r(n, n, n, n) \tag{7.1}$$

Using the shorthand $N_k(n,\ i)$ for $N_r(n,\ i,\ i,\ n)$, we can represent a neighbourhood as shown in Equation (7.2).

$$N(n) = N_k(n, 1) + N_k(n, 2) + \cdots + N_k(n, n) \tag{7.2}$$

Returning to GenerateSuccessors-Rnd, we would observe that the algorithm is written in such a way so as to allow the use of a custom function for choosing how many compound moves i to perform when generating a neighbour. If we knew how many elements are in each set $N_k(n,\ 1) \ldots N_k(n,\ n)$, we could use these values to perform a weighted pick to choose i. If we could then probe the $N_k(n,\ i)$ set uniformly, we could instantiate GenerateSuccessors-Rnd in such a way that it would be analogous to GenerateSuccessors, but without generating the entire neighbourhood.

However, the use of compound moves means that even if we knew the size of each N_k neighbourhood, we would not be able to calculate their weights correctly so that they could be used with GenerateSuccessors-Rnd. As we saw in Section 6.4.7, some program trees can only be created through using compound moves. If there are two separate edit sequences which create the same program tree that use a different number of compound moves, they will be placed in different N_k neighbourhoods. Without enumerating all neighbourhoods and checking for the existence of duplicate program trees, we would be unable to attribute weights to each N_k neighbourhood such that they could be used in the way intended.

In truth, we do not know whether the way that GenerateSuccessors picks a neighbour is a behaviour that is beneficial for LSPS in general, and therefore we do not know whether GenerateSuccessors-Rnd having a different behaviour is a disadvantage or not. However, we believe it prudent to attempt to emulate the behaviour of GenerateSuccessors as it is unbiased. If we ever desired to augment the choosing of a neighbour with some additional heuristic mechanism, we would know that if we used GenerateSuccessors (or any function that could approximate it), then no additional bias would be introduced by its use.

In the next subsection we identify mechanisms to be used with GenerateSuccessors-Rnd to alleviate the issues that we have highlighted in this subsection.

7.3.2 Using Randomised Neighbourhood Generation

In the previous subsection we highlighted two disadvantages that using GENERATE-SUCCESSORS-RND has when compared to using GENERATESUCCESSORS. In this subsection we identify additional mechanisms to be used with GENERATESUCCESSORS-RND that we believe can alleviate these issues.

Early Termination

In the LSPS algorithms that underpin Experiments A1 to A4, early termination occurred if all neighbours had been evaluated and no fitter heuristic found. Without this functionality, each of those experiments would have performed 10^8 heuristic evaluations in total, many of which would have been redundant. Using this early termination mechanism significantly reduced this number - for example Experiment A4 performed approximately 1.3×10^6 heuristic evaluations in total.

When using GENERATESUCCESSORS-RND, we cannot determine when a neighbourhood has been fully evaluated. Therefore the mechanism described in the previous paragraph is not applicable when using it. Instead, we propose an alternate early termination mechanism. It is described as follows; if no fitter neighbour has been found after evaluating e elements of the current candidate program tree's neighbourhood, then early termination occurs.

For the experiment we will perform in this section - previously designated Experiment A5 - we require an appropriate e value. Generally it would be preferable to use a value which will provide a good chance of finding the fittest heuristics, while not spending unnecessary time evaluating neighbourhoods containing no fitter neighbours.

To find an appropriate e value for Experiment A5, instead of performing computationally expensive experiments with different e values, we will perform experiments which simulate re-running Experiment A4. These simulations are designed to show how using the previously described early termination mechanism would have changed the results of Experiment A4 if it had been used as part of Experiment A4's overarching LSPS algorithm. Each simulation uses a different e value. Because we have memoized detailed results from Experiment A4, these simulations can be quickly performed, allowing us to test several different e values and find one that meets our criteria. As Experiment A5 is almost identical to Experiment A4, we believe that these simulations will provide us with an appropriate e value.

To be clear, the experiments we perform do not simulate using GenerateSuccessors-Rnd instead of GenerateSuccessors in Experiment A4's overarching LSPS algorithm. Instead they simulate augmenting Experiment A4's overarching LSPS algorithm with the previously described early termination mechanism. It is Experiment A5 that will use GenerateSuccessors-Rnd in place of GenerateSuccessors. The simulated experiments can be seen as an intermediary step between Experiments A4 and A5.

Before describing the set of simulated experiments, we must first introduce some notation. Consider the formula in Equation (7.3).

$$f(R, s_{min}) = \sum_{r \in R} \begin{cases} h(r_{final}) & \text{if } s(r_1) > s_{min} \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

The function f in Equation (7.3), when given the experiment results R and integer s_{min}, describes the sum of the heuristic quality across all repetitions of R. The experiment results R can be visualised as a set. For each repetition $r \in R$, $h(r_{final})$ represents the final heuristic's fitness from r, and $s(r_1)$ represents the size of the initial heuristic from r. We use the size parameter to omit some repetitions from f by comparing the s_{min} value to the $s(r_1)$ value.

We write $sim_{R,e}$ to refer to the results produced from simulating experiment R using the previously described early termination mechanism with a parameter value of e. The reader may assume that we next perform a set of simulations with different e values, with the goal of finding one which meets some performance criteria. Instead, the simulations take the following form; when given the parameter p (where $0 \leq p \leq 1$), the experiment results R and the integer s_{min}, we find the minimum e value that satisfies the formula in Equation (7.4).

$$f(R, s_{min}) \times p \geq f(sim_{R,e}, s_{min}) \tag{7.4}$$

All the simulations we perform use $R =$ Experiment A4 and $s_{min} = 4$. Each simulation uses a different p value. In essence, these simulations allow us to find the minimum e value which produces heuristics with a quality that is within a percentage of the original result's heuristic quality. This percentage is represented by p. In Section 7.2.2 we noted how some repetitions created small initial heuristics, and many of them arrived at the same final heuristic. The s_{min} value of 4 is used to disregard those results from these simulations.

Table 7.3: Data from simulated re-runs of Experiment A4 using the termination mechanism described in Section 7.3.1. We show data from six simulations, which we have named Experiment A4-Sx, for x values $\in \{1 \dots 6\}$. Each simulation has an associated p value, which is used to calculate the maximum e value that satisfies Equation (7.4). For each simulation we show its p value, the calculated e value, and the total number of evaluations performed over all repetitions of that simulation. We also show the total number of evaluations used over all repetitions of Experiment A4.

Name	p	e	Sum of Evals
Experiment A4	-	-	1,330,691
Experiment A4-S1	1.0	22,860	886,717
Experiment A4-S2	0.999	17,469	813,413
Experiment A4-S3	0.995	7,062	567,074
Experiment A4-S4	0.99	5,120	421,734
Experiment A4-S5	0.98	3,163	282,691
Experiment A4-S6	0.96	1,576	162,972

In Table 7.3 we show details of the six simulations performed. Each simulation has a name in the form Experiment A4-Sx for x values $\in \{1 \dots 6\}$. We show each simulation's p value, the calculated e value, and the sum of all evaluations performed in that simulation. In Figure 7.5 we show the distribution of heuristic fitness values produced from each simulation.

These results show us that a small change in p equates to a much larger change in e. For example, to obtain heuristics with a quality that is 99% of the quality of the heuristics returned from Experiment A4, an e value of just over $5,000$ is required, yet to obtain heuristics with a quality that is 98% of the quality of the heuristics returned from Experiment A4, an e value of just over $3,000$ is required.

Based on these results, we believe that an e value of $7,500$ is appropriate for our needs. The data in Table 7.3 shows us that it produces heuristics that are within 99.5% of the quality of those produced from Experiment A4. Yet, the total number of evaluations required is less than 62% of what was originally performed.

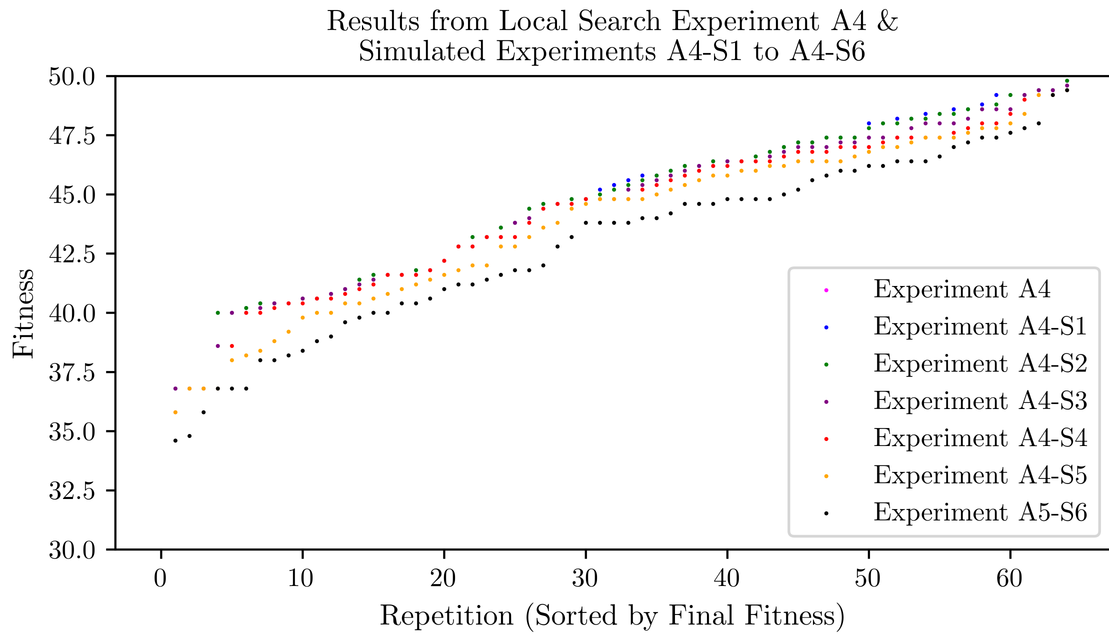


Figure 7.5: The fitness values of the final heuristics created from the simulations of Experiment A4. The simulations model re-running Experiment A4 augmented with the early termination mechanism described in Section 7.3.1. Each simulation uses a different e parameter value, which we give in Table 7.3. We only simulate the 65 repetitions from Experiment A4 whose initial heuristic was of size > 4.

Picking Neighbours Randomly

The second mechanism described in this subsection is designed to address the issue regarding how a heuristic is chosen from a neighbourhood. Specifically, we want the way that GenerateSuccessors-Rnd picks a neighbour to be as similar as possible to the method used in Experiments A1 to A4. In those experiments, after GenerateSuccessors created a neighbourhood of size k, each neighbour had a $\frac{1}{k}$ chance of being chosen.

It is difficult to create an algorithm that does this without memoizing the neighbourhood. There are many reasons for this - for example, in Section 7.3.1 we discussed how compound moves make this difficult. Another issue we have not addressed is that in Experiments A1 to A4 elements were removed from the neighbourhood after they had been chosen. GenerateSuccessors-Rnd does not take this into account. Therefore, it is not possible to use GenerateSuccessors-Rnd in such a way so that it is semantically identical to GenerateSuccessors. Instead, we try to instantiate GenerateSuccessors-Rnd so that it approximates picking an element from a neighbourhood with uniform randomness, ignoring previously selected neighbours.

To do this we use previous results to approximate the size of the N_k neighbourhoods, which we use as the weights for the number of compound moves to perform in GenerateSuccessors-Rnd, as discussed in Section 7.3.1.

To find an appropriate way of approximating the size of these neighbourhoods, we collected all candidate heuristics from Experiments A3 and A4. We separated them into subsets based on their size. For each heuristic in each subset, we generated their $N_k(3,\ i)$ neighbourhoods for $i \in \{1,\ 2,\ 3\}$. We then calculated the average sizes of each of these neighbourhoods for each subset. In Figure 7.6 we show this size data plotted against the size of the heuristics.

We can identify two clear trends from this data. Firstly, the number of neighbours in $N_k(3,\ 1)$ best resembles a polynomial function when plotted against the candidate heuristic's size. $N_k(3,\ 2)$ and $N_k(3,\ 3)$ appear to grow exponentially. In Equations (7.5) to (7.7) we show functions that best approximate the data for each $N_k(3,\ i)$ neighbourhood. In those equations the s variable is the size of the heuristic.

$$|N(3,1)_k| = 2.37s + 2.37 \tag{7.5}$$

$$|N(3,2)_k| = 28.9s \times 0.038^s \tag{7.6}$$

$$|N(3,3)_k| = 28.8s \times 0.084^s \tag{7.7}$$

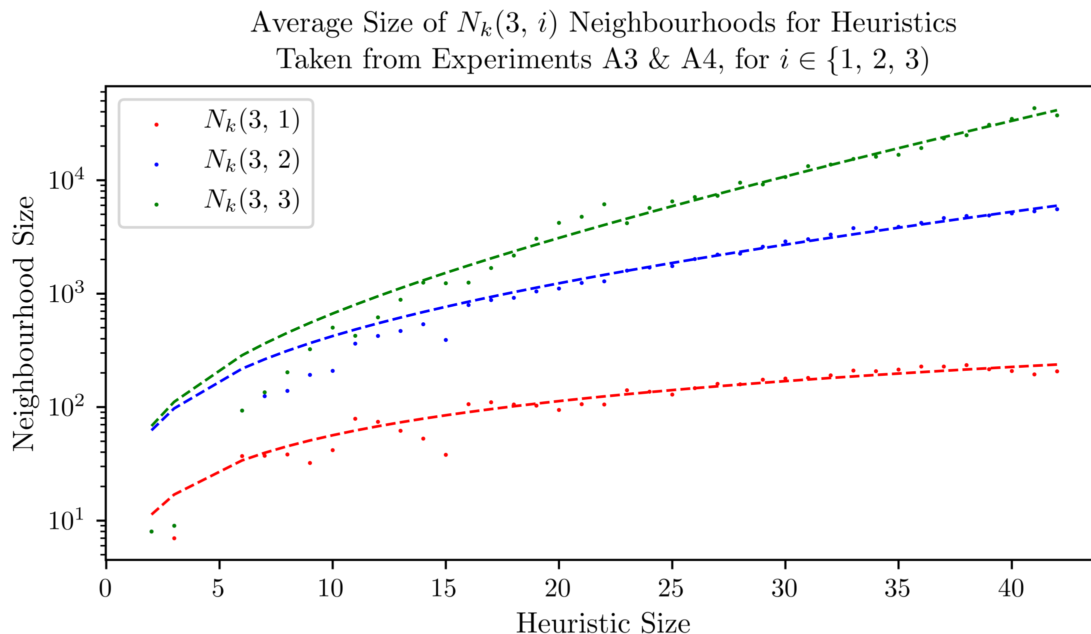


Figure 7.6: The average number of neighbours in each N_k neighbourhood for all candidate heuristics from Experiments A3 and A4. We also show lines of best fit for each neighbourhood.

It is the functions in Equations (7.5) to (7.7) that we will use to approximate the size of the N_k neighbourhoods in Experiment A5, and the rest of the experiments in this chapter.

Finally, we would like to be clear to the reader that even if we were able to calculate the exact size of the N_k neighbourhoods, using them as weights with GenerateSuccessors-Rnd would still not emulate GenerateSuccessors perfectly. This is because GenerateSuccessors-Rnd does not select an element from each N_k neighbourhood uniformly, instead it picks a point in the input program tree to identify a TPPT from, and then selects an edit sequence randomly from any that match with that TPPT. We believe that it may be possible to select an element from the N_k neighbourhood uniformly, however our attempts to create such an algorithm were unsuccessful.

In the next subsection we present Experiment A5, which uses the GenerateSuccessors-Rnd neighbourhood generation algorithm together with the mechanisms laid out in this subsection.

7.3.3 Randomised Neighbourhood Generation Experiment

In this subsection we describe the methodology and present the results from Experiment A5.

Methodology

Experiment A5 is a series of repetitions of an LSPS algorithm, used to create LS-SAT heuristics using Language A. Experiment A5's overarching algorithm is based on Local-Search-Rnd. Each repetition is initialised using the grow method from GP, with a maximum depth of 4. The neighbourhood generation algorithm GenerateSuccessors-Rnd is used to generate neighbours. The functions in Equations (7.5) to (7.7) are used by GenerateSuccessors-Rnd to decide how many compound moves to use for any given neighbour. The neighbourhood defined by $N(3)$ is used.

The heuristics are evaluated using the fitness function described in Section 3.4.1. The algorithm can terminate in one of two ways; either when $7,500$ neighbours of the current heuristic have been evaluated and no fitter neighbour found, or when a total of $100,000$ heuristics have been evaluated. Experiment A5 consists of 100 repetitions.

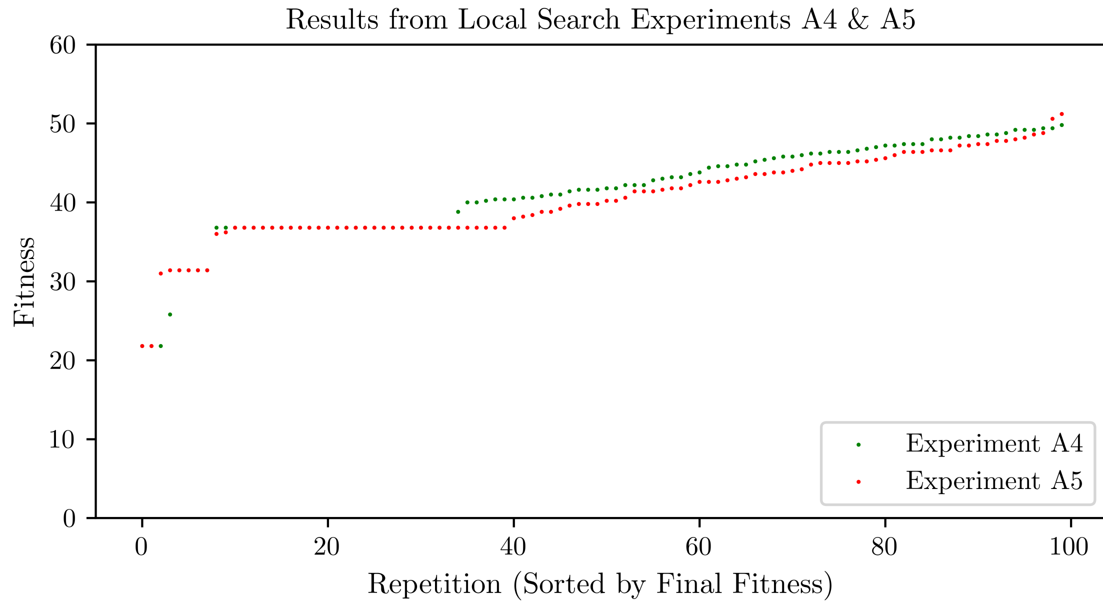
Experiment A5 is designed to be similar to Experiment A4, except that the function GenerateSuccessors-Rnd is used to generate the neighbours. We also use the two mechanisms described in the previous subsection to alleviate the two disadvantages highlighted in Section 7.3.1 regarding the use of GenerateSuccessors-Rnd.

Experiment A5 was ran on the system described in Section 4.2.2. In total, it took 45 hours to terminate.

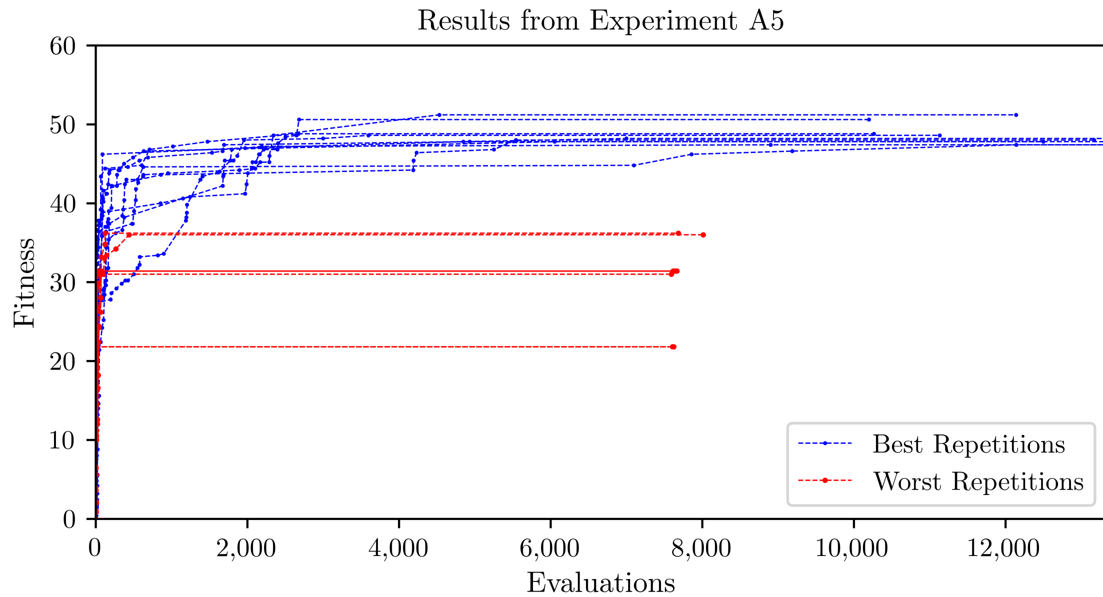
Results & Analysis

In Figure 7.7 we present two graphs; one shows the quality of the heuristics created from each repetition of the experiment, and the other shows how ten of the best and worst repetitions from the experiment progressed. In Table 7.4 we show additional statistical data pertaining to the experiment.

From the data shown in Figure 7.7a, we can see that the distribution of heuristic values reported from Experiment A5 is similar to that reported from Experiment A4. Both begin with a set of heuristics with low fitness values in a flat plateau. The remaining results trend steadily upwards. On average, the heuristics from Experiment A4 reported a higher fitness, though Experiment A5 produced the fittest heuristics.



(a) Final results from Experiment A5. Each data point represents the fitness of the final heuristic returned from that repetition of the local search algorithm. We also show the results from Experiment A4 for comparison.



(b) Ten of the best and worst repetitions from Experiment A5. Each sequence of data points represents a single repetition.

Figure 7.7: Results from Experiment A5.

Table 7.4: Statistical data from Experiment A5. We show the steps (number of times the candidate heuristic changes), the number of evaluations, the start, end and difference in size (s) and fitness (f) of the initial (h_1) and final heuristic (h_{final}), the MTED and SMTED between the first and final heuristic, and the number of evaluations required to find the final heuristic.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	15.79	4	9.00	14.50	22.00	36
Evals	9,890.45	7,503	7,717.00	8,177.50	10,984.25	23,273
$f(h_1)$	2.34	0.00	0.00	0.80	2.70	23.40
$f(h_{final})$	40.42	21.80	36.80	40.00	45.00	51.20
Δ_f	38.08	16.40	34.60	36.80	42.60	50.19
Evals to h_{final}	2,390.45	3	217.00	677.50	3,484.25	15,773
$s(h_1)$	14.56	2	3.00	14.50	23.00	39
$s(h_{final})$	18.50	7	8.00	18.00	27.00	39
Δ_s	3.94	-1	2.00	4.00	5.00	19
MTED(h_1,h_{final})	10.84	3	6.00	10.00	15.00	27
SMTED(h_1,h_{final})	4.62	0	3.00	4.00	5.00	19

This suggests to us that the LSPS algorithm underpinning Experiment A5 is generally less effective than that underpinning Experiment A4, and on average appears to create heuristics of a lower quality.

We had expected Experiment A5 to produce heuristics with a lower fitness than Experiment A4, as the neighbourhood generation algorithm used in Experiment A4 is able to systematically search all neighbours, thereby providing a guarantee that if a fitter neighbour exists it will be found. Experiment A5 provides no such guarantee, and therefore it is possible that a fitter neighbour won't be found. These results appear to reinforce our assumptions about how Experiment A5's overarching LSPS algorithm performs.

In Figure 7.7b we show the ten best and worst repetitions from Experiment A5. Compared to those from Experiment A4, Experiment A5's worst repetitions required many more evaluations to terminate, yet the quality of heuristics found is similar. We attribute the large difference in the number of heuristic evaluations to the different mechanisms that each experiment used to terminate early. In Experiment A5, the LSPS algorithm terminates early only if $7,500$ neighbours have been evaluated and no

fitter neighbour found. On further analysis, we found that the ten worst performing repetitions all arrived at a small final heuristic which had a small neighbourhood. Therefore we can conclude that, in those repetitions, a large number of unnecessary heuristic evaluations were performed. This is an unforeseen effect that introducing the early termination mechanism described in Section 7.3.2 had on the results. In future work, it may be prudent to use the approximated size of a heuristic's neighbourhood to determine the number of evaluations to perform before early termination occurs. By doing so, we may be able to reduce the number of unnecessary heuristic evaluations performed for those neighbourhoods with fewer members.

When we look at the repetitions that produced the best heuristics from Experiment A5, we can see a clear difference in the number of evaluations performed compared to the best repetitions from Experiment A4. In both experiments, we can see that the trajectory of the best repetitions is to rise to a high fitness value quickly, after which any improvements to a heuristic's fitness is small. In Experiment A5, for the repetitions highlighted, the number of evaluations performed is far less than for Experiment A4.

The data in Table 7.4 shows us more detailed information regarding Experiment A5. In regards to the data concerning the sizes of heuristics, the fitnesses of heuristics and the change in structure of the heuristics, there is little difference between the results from Experiments A4 and A5. However, across all metrics concerned with the number of evaluations performed, Experiment A5 reported substantially different values compared to those reported from Experiment A4. For example, in Experiment A4 the mean number of evaluations and evaluations required to obtain h_{final} was reported as just below $13,500$ and just over $6,000$ respectively, whereas for Experiment A5 these values were reported as just under $10,000$ and a little over $2,000$. If we look at the upper quartile value for the number of evaluations performed, Experiment A5 reported a value less than half that reported from Experiment A4, and if we look at the minimum and lower quartile values for the number of evaluations, Experiment A5 reported a substantially higher value than those reported from Experiment A4.

It stands to reason that the difference in results can be attributed to the differences in Experiment A5 compared to Experiment A4. For those repetitions that reported a final heuristic with a lower fitness, as highlighted previously, we believe that the much greater number of evaluations performed in Experiment A5 is due to the early termination mechanism. For the repetitions whose final heuristic had a higher fitness,

Experiment A5 reported that they required a substantially smaller number of heuristic evaluations. We were surprised by this result. While we had expected there to be a smaller number of evaluations performed, we had not expected such a large difference. Yet, when looking at the quality of heuristics found, it appears that the smaller number of evaluations performed did not overly effect the ability of the algorithm to find effective LS-SAT heuristics.

The experiment presented in this subsection shows the reader how the randomised neighbourhood generation algorithm can be used to navigate the search space of heuristics. While the results we have seen suggest that the heuristics created from Experiment A5 are of worse quality than those created from Experiment A4, we are satisfied that Experiment A5's LSPS algorithm is still able to effectively navigate the search space of heuristics, and the mechanisms introduced appear to work as intended.

7.3.4 Summary

In this section we have illustrated the reasoning behind the use of GenerateSuccessors-Rnd, shown several undesirable effects its use may have on the overarching LSPS algorithm, and described two mechanisms to alleviate these effects. We then described Experiment A5, and presented the results from it.

The experiment performed in this section can be seen as an extension to those performed in Section 7.2. Across all repetitions, the heuristics created from Experiment A5 were not of as high quality as those created from Experiment A4. Yet, we are satisfied that when using GenerateSuccessors-Rnd, we are still able to navigate the search space of heuristics, and find effective candidate solutions. This method of neighbourhood generation can consider neighbourhoods of much larger size than GenerateSuccessors, as it does not require storing all neighbours. In all remaining LSPS experiments performed in this chapter, we use this method of neighbourhood generation.

In the next section we present an LSPS experiment that uses neighbourhoods far larger than those used in any experiment described previously.

7.4 Using an Alternate Cost Function

In the previous section we described Experiment A5, an LSPS experiment that used GenerateSuccessors-Rnd. GenerateSuccessors-Rnd can probe an arbitrarily sized neighbourhood without having to store it. This gives it an advantage compared to GenerateSuccessors, used in Experiments A1 to A4. Using GenerateSuccessors-Rnd, we can now consider neighbourhoods of virtually limitless size.

In this section we consider an LSPS experiment which would not be possible to perform using GenerateSuccessors. The experiment detailed in this section uses an alternate definition of a neighbourhood. This definition increases the number of edit sequences associated with a specific language and neighbourhood bound. The sizes of the neighbourhoods under consideration with this definition are much larger than those considered in Experiments A1 to A5, and those experiments performed in Chapter 5. Hence, with the computational resources available to us, it is only possible to perform this experiment using GenerateSuccessors-Rnd.

The format of this section is as follows; In Section 7.4.1 we discuss the motivation behind considering an alternate way of defining a neighbourhood, before providing the definition we use in this section. In Section 7.4.2 we detail the methodology behind Experiment A6, the experiment we describe in this section, and present the results from it. Finally in Section 7.4.3 we provide a summary of the work undertaken in this section.

7.4.1 Observations

In the proceeding sections in this chapter, all of the experiments we have described have used the definition of a neighbourhood provided in Definition 23. In Definition 26 we gave an alternate definition of a neighbourhood, however we only performed experiments which used neighbourhoods that could be defined in terms of Definition 23. The neighbourhood definition in Definition 23 has been used successfully in our experiments; we have seen many examples of LSPS repetitions which have created heuristics with a high fitness. In this subsection we give our rationale behind the use of an alternate definition of a neighbourhood for use with an LSPS algorithm.

To begin, we will provide evidence which suggests that the neighbourhood definition used in Experiments A1 to A5 constricts the overarching LSPS algorithm. In all

experiments, this was the $N(3)$ neighbourhood. In Tables 7.2 and 7.4 we presented statistical data for Experiments A3 to A5. We highlight these experiments as they placed no constraints on the size of heuristic that could be created. As noted in Section 7.2.2, the change in the size of the final heuristic compared to the initial heuristic was quite small for Experiments A3 and A4, and this trend continued in Experiment A5. Of the three experiments, the highest average change in size was reported as just under 4, the upper quartile value reported as exactly 5, and the largest change in size observed was 19. We had expected there to be many more examples of repetitions which had reported a large change in heuristic size. For the reported SMTED data, the values are similar to those reported for the change in size. While the MTED data values reported are much larger, we believe this is due to the relabelling of nodes, which the other size metrics do not take into account. In general, we would state that there are few examples of repetitions which have created a final heuristic that has a substantially different structure compared to that repetition's initial heuristic - at least according to the metrics outlined here.

As to why there are not many examples of repetitions showing larger changes in heuristic size, we believe that it may be due to the way in which the neighbourhood is defined. In Table 7.5 we show statistical data pertaining to all edit sequences found in the $N(3)$ neighbourhood for Language A. To be clear to the reader, these edit sequences are comprised of edits with a cost of exactly 1, and have a cost of at most 3. In this data we show how many edit sequences correspond to a change in an input program tree's size. We calculated this by counting the number of insertions and deletions in each edit sequence. Though this data does not take into account the application of compound moves, it does show us that there are relatively few edit sequences that change the size of an input program tree by any number greater than 1. This suggests to us that, on average, the majority of the neighbours of an input program tree will be the same size, or have a size difference of 1. We believe that with fewer opportunities to make larger edits, there will be fewer repetitions which show large changes in program tree size when the final program tree is compared to the initial program tree.

The second point we would like to draw the attention of the reader to concerns specific examples of edit sequences found within neighbourhoods. In Figure 7.8 we show three PTPPTs. The PTPPT in Figure 7.8a corresponds to a specific configuration of nodes in Language A. This configuration represents the two arguments

Table 7.5: Data concerning the frequency of different types of edit sequences in $N(3)$. To compute this data, the edit sequences were separated into groups based on how they change a program tree's size. We show how many edit sequences are in each group. This data only takes into account the application of single edit sequences, not compound moves. As there is a reciprocal for every edit sequence, the number in a set that increases by n or decreases by n is identical.

Change in Size	+/-3	+/-2	+/-1	0	Total
# Edit Sequences	40	143	1,910	1,338	5,804

that the `GetOldestVar` function requires. The observations we make below are also applicable to PTPPTs that represent the arguments for the `IfRandLt`, `IfNotMinAge`, `IfVarCompare`, `IfVarCond` and `IfTabu` functions, albeit with additional typed nodes used to represent the additional arguments these functions require.

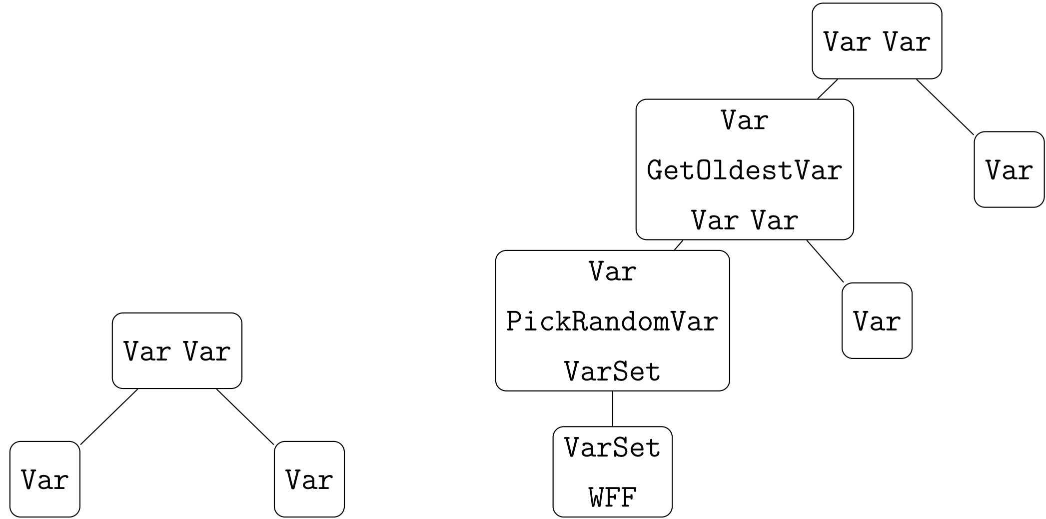
In Figures 7.8b and 7.8c we show two PTPPTs that can be created from the PTPPT in Figure 7.8a. This can be done using edit sequences which we refer to as $edit_{a,b}$ and $edit_{a,c}$ respectively. In addition, there is an edit sequence that can be used to transform the PTPPT in Figure 7.8b into the PTPPT in Figure 7.8c, which we refer to as $edit_{b,c}$. $edit_{a,b}$ and $edit_{b,c}$ have a cost of 3, while $edit_{a,c}$ has a cost of 5. $edit_{a,b}$ and $edit_{b,c}$ are in the set of edit sequences described by the $N(3)$ neighbourhood, while all three edit sequences are in the $N(5)$ neighbourhood.

$edit_{a,b}$ and $edit_{a,c}$ are representative of a specific type of edit sequence. They both add a new branch to a program tree that has a `Var` return type. These edit sequences can be repeatedly applied to a program tree to grow it indefinitely. Reciprocals of these edit sequences also exist, which delete `Var` branches.

Consider the program tree t_a, which contains a node p. Node p is the root of a configuration of nodes in t_a that is analogous to the PTPPT in Figure 7.8a. The program tree t_b is obtained by applying $edit_{a,b}$ to t_a at node p. The program tree t_c is obtained by applying $edit_{b,c}$ to t_b at node p. t_c can also be obtained by applying $edit_{a,c}$ to t_a at node p. In order to obtain t_c under $N(3)$ from t_a by applying $edit_{a,b}$ then $edit_{b,c}$, t_a, t_b and t_c's fitness values must be in a specific ordering. This ordering is shown in Equation (7.8).

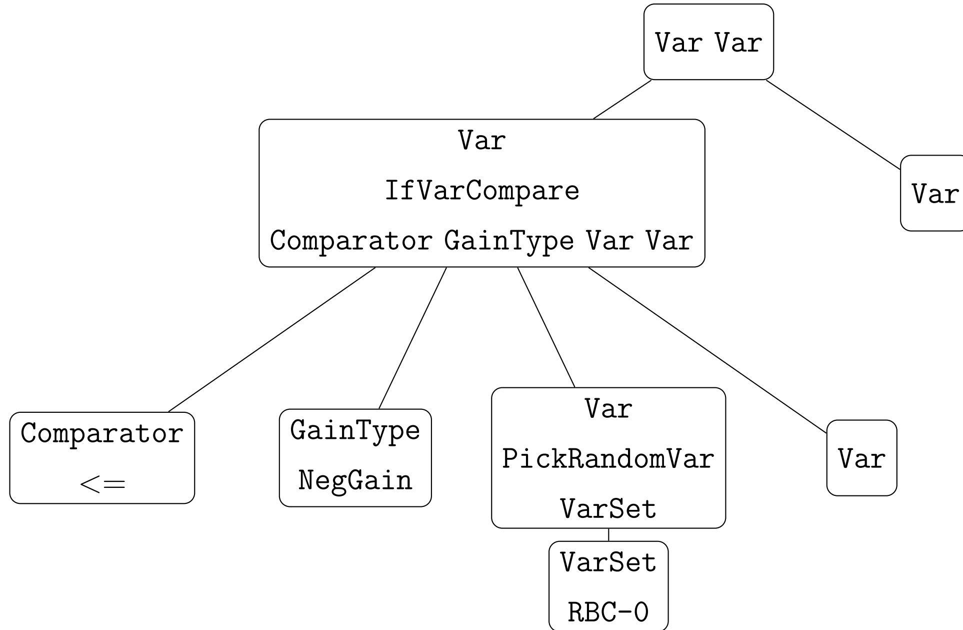
$$f(t_a) < f(t_b) < f(t_c) \tag{7.8}$$

However, if $f(t_b) < f(t_a)$ then t_c would not be obtained. While it is reasonable to



(a) A PTPPT written under Language A. It type checks, and is considered a start state PTPPT.

(b) The PTPPT after three nodes have been inserted into the PTPPT shown in Figure 7.8a. It type checks, and is considered an end state PTPPT. The edit sequence to obtain this PTPPT from the PTPPT in Figure 7.8a has a cost of 3.



(c) The PTPPT after five nodes have been inserted into the PTPPT shown in Figure 7.8a. It type checks, and is considered an end state PTPPT. The edit sequence to obtain this PTPPT from the PTPPT in Figure 7.8a has a cost of 5.

Figure 7.8: A start state PTPPT and two end state PTPPTs written using Language A.

assume that t_c could be obtained from t_a by some other sequence of edit sequences, we note that $edit_{a,b}$ and $edit_{b,c}$ are the most direct way of creating t_c from t_a under $N(3)$.

Under $N(3)$ and Language A there are many other examples of start state PTPPTs (s_1) and pairs of edit sequences ($edit_1$ and $edit_2$) with the following properties; $edit_1$ can be applied to s_1 to create s_2. $edit_2$ can be applied to s_2 to create s_3. $edit_1$ contains three insertions, one of which is an insertion of the function `GetOldestVar`. The other inserted nodes in $edit_1$ create a new `Var` branch. $edit_2$ relabels the previously inserted `GetOldestVar` function and inserts at most 2 new nodes underneath it.

The example in Figure 7.8 describes a start state PTPPT and pair of edit sequences with these properties. In that example the function `IfVarCompare` is the node `GetOldestVar` was relabelled with, but this could be substituted for other functions such as `IfNotMinAge`, `IfRandLt` or `IfTabu`.

These PTPPTs and edit sequences can be extrapolated to full program trees. However, the start, intermediary and end state program trees must have a fitness relationship which mirrors that described in Equation (7.8) in order to obtain the end state program tree in edits with the minimum cost. This tells us that there are output program trees under Language A and $N(3)$ which, in order to create from an input program tree using edits with a minimum cost, rely on a single intermediary program tree's fitness being less than the end state program tree's fitness and greater than the input program tree's fitness.

We believe that, due to the composition of the language, the $N(3)$ neighbourhood may be limiting the opportunities that the overarching LSPS algorithm has to create certain heuristics, some of which may have a high fitness value. This could mean that the algorithm is getting stuck in local optima when there are higher quality heuristics which are relatively close by, as the algorithm is unable to create them.

For these reasons, we think that it is worth investigating whether using a neighbourhood which includes more edit sequences like $edit_{a,c}$, as well as edit sequences which make larger changes to a candidate program tree, can be used with an LSPS algorithm to create higher quality heuristics. The reader may assume that we would create such a neighbourhood by increasing the neighbourhood bound. However in Table 6.3 we saw that calculating $N(4)$ required nearly 2 hours of computation time, and we believe any neighbourhoods with a larger bound would be impractical to calculate. Instead, we redefine the neighbourhood to include a cost function as

a parameter, and then instantiate a neighbourhood with a cost function designed specifically for our needs. The alternate definition for the neighbourhood is given as follows:

Definition 27 (Cost-Parameterised Neighbourhood of a Program Tree) *Given a language L, a candidate program tree h, a number representing the maximum bound of a single edit n_s, two integers representing the minimum and maximum number of compound moves n_{m_0} and n_m, a number representing the sum of the costs that all edits can have n_{sum}, and a cost function f which attributes a cost to each edit, the cost-parameterised neighbourhood $N_C(h, f, n_s, n_{m_0}, n_m, n_{sum})$ is defined as all valid program trees under L that can be obtained from h through a series of sequences of tree edits with the following properties. Each sequence of tree edits describes a transformation between two valid program trees under L. Each sequence of tree edits has a cost of at most n_s under f. There are between n_{m_0} and n_m sequences of tree edits. The total sum of the cost of all sequences of tree edits under f is at most n_{sum}. If the program tree is obvious from the context, we write $N_C(f, n_s, n_{m_0}, n_m, n_{sum})$.*

As with Definitions 23 and 26, Definition 27 does not explicitly mention the tree edits described in Section 2.6. However, it is our intention that it is used with those tree edits. Consequently, any cost function used with Definition 27 must also be defined in terms of these edits. Care must be taken to ensure that any cost function used remains consistent with the cost function requirements of the MTED metric, laid out in Section 2.6. Doing so ensures that the neighbourhood definition and the neighbourhood generation algorithm remain consistent with each other.

Instead of instantiating the cost-parameterised neighbourhood definition with a specific cost function, we use the function Cost in this chapter. It is defined in Equation (7.9).

$$\textsc{Cost}(f, edit) = \begin{cases} f(label(edit)) & \text{if } insert(edit) \vee delete(edit) \\ \max(f(label_1(edit)), & \text{if } relabel(edit) \wedge \\ \quad f(label_2(edit))) & \quad label_1(edit) \neq label_2(edit) \\ 0 & \text{otherwise} \end{cases} \tag{7.9}$$

Equation (7.9) uses several functions whose meaning may not be immediately obvious. These functions can be described as follows; *insert*, *delete* and *relabel* return *True* if the edit argument is an insertion, deletion or a relabel edit respectively, and *False*

otherwise. *label* gets the term to be inserted or deleted. $label_1$ gets the term that is to be relabelled, while $label_2$ gets the new term for the relabelled node.

Cost takes two arguments; a function f and the edit to compute the cost of, *edit*. *edit* is defined in terms of the MTED edits laid out in Section 2.6. The function f takes an element in the language L and returns an integer representing a cost. For a function f, we write $c = \textsc{Cost}(f)$ to represent partially instantiating Cost with f. We consider c to be in the correct form for use with Definition 27. In Section 6.2 we informally described a generalised cost function for use with the neighbourhood generation algorithm. The cost function shown in Equation (7.9) is designed to be analogous to that cost function.

In Equation (7.10) we show the function f_{basic}.

$$f_{basic}(term) = 1 \tag{7.10}$$

Using f_{basic}, we can define the $N(3)$ neighbourhood used throughout this thesis in terms of Definition 27. It is shown in Equation (7.11).

$$N(3) = N_C(\textsc{Cost}(f_{basic}), 3, 1, 3, 3) \tag{7.11}$$

In Equation (7.12) we show the function f_{alt}. f_{alt} uses the function $function$, which returns $True$ if its argument is a function and $False$ otherwise.

$$f_{alt}(term) = \begin{cases} 1 & \text{if } function(term) \\ 0.1 & \text{otherwise} \end{cases} \tag{7.12}$$

Using f_{alt} we can define the N_{alt} neighbourhood. This is the neighbourhood we will be using in this section. N_{alt} is shown in Equation (7.13).

$$N_{alt} = N_C(\textsc{Cost}(f_{alt}), 2.8, 1, 3, 2.8) \tag{7.13}$$

While the way that N_{alt} is constructed may appear to be unintuitive compared to the other neighbourhoods we have considered, it has been specifically designed with the observations made in this subsection in mind. Informally, in N_{alt} each edit operation that uses a function has a cost of 1, and those using a terminal a cost of 0.1. The desired effect of this cost function is that the neighbourhood will contain many edit sequences that consist of edits operating on terminals, and at most 2 that operate on functions.

In Table 7.6 we show data regarding the N_{alt} neighbourhood for Language A. Through analysing $N(3)$ and N_{alt}, we found that N_{alt} contains $5,672$ of the $5,804$

edit sequences in $N(3)$. In total, there are $1,211,108$ edit sequences in N_{alt}, one of which is $edit_{a,c}$. This is obviously a much larger number of edit sequences than $N(3)$ contains. It would not be practical to generate the entire neighbourhood of any program tree using this neighbourhood definition, instead we will have to use the randomised neighbourhood generation algorithm. The data in Table 7.6b shows us that N_{alt} contains many more edit sequences which correspond to a large change in program tree size when compared to $N(3)$. In Table 7.6a we can see that it took much longer to compute N_{alt} than $N(3)$. In the next subsection we describe Experiment A6, which is an LSPS experiment that uses the N_{alt} neighbourhood. Through analysing the results from Experiment A6, we aim to ascertain whether the additional time required to generate the edit sequences for N_{alt} can be justified by the quality of heuristics created.

7.4.2 Alternate Cost Function Experiment

In this subsection we describe the methodology and present the results from Experiment A6.

Methodology

Experiment A6 is nearly identical to Experiment A5. Instead of providing a complete description of its methodology, we give a brief description of it in terms of its differences compared to Experiment A5.

- Experiment A6 is formulated in exactly the same manner as Experiment A5, except that the neighbourhood used is the N_{alt} neighbourhood, instead of the $N(3)$ neighbourhood.

Experiment A6 was ran on the system described in Section 4.2.2. In total, it took 115 hours to terminate. Through performing Experiment A6, we aim to ascertain the effect that the N_{alt} neighbourhood has on the results. Specifically, we want to investigate the difference in the fitness and size of the created heuristics when compared to those created from other experiments.

Results & Analysis

In Figure 7.9 we present two graphs; one shows the quality of the heuristics created from each repetition of the experiment, and the other shows how ten of the best and

Table 7.6: Data concerning the neighbourhood N_{alt} for Language A. We show statistics the neighbourhood generation algorithm reported when constructing N_{alt}, as well as statistical data concerning the edit sequences in N_{alt}.

(a) Data gathered when calling CREATE-ALL-EDIT-SEQUENCES on Language A using the neighbourhoods $N(3)$ and N_{alt}. We assume that the language parameter to NEIGHBOURHOOD-GENERATION contains the cost function used. So that our definitions remain consistent, the n_s parameter in Definition 27 is analogous to the n_{max} parameter in CREATE-ALL-EDIT-SEQUENCES. For the language used (L), we show the number of functions (F) and terminals (T) in that language. In brackets are the number of terms with a unique type signature in each set. For each call to CREATE-ALL-EDIT-SEQUENCES, as described by the n_s value and cost function, we show the size of the set of TPPT start states, the total number of TPPTs for which all the edit sequences were found (we refer to these as problems solved), and the time taken to compute the edit sequences.

L	F	T	n_s	Cost Function	Start States	Problems Solved	Time taken (ms)
A	9 (8)	26 (6)	3	COST(m_{basic})	38,016	526,630	29,685
			2.8	COST(m_{alt})	49,931	60,259,738	1,012,797

(b) Data concerning the frequency of different types of edit sequences in N_{alt} for Language A. To compute this data, the edit sequences were separated into groups based on how they change a program tree's size. We show how many edit sequences are in each group. This data only takes into account the application of single edit sequences, not compound moves. As there is a reciprocal for every edit sequence, the number in a set that increases by n or decreases by n is identical.

Change in Size	+/-7	+/-6	+/-5	+/-4	+/-3
# Edit Sequences	82,944	45,090	17,568	1,548	2,560
Change in Size	+/-2	+/-1	0	Total	
# Edit Sequences	34,533	299,354	243,914	1,211,108	

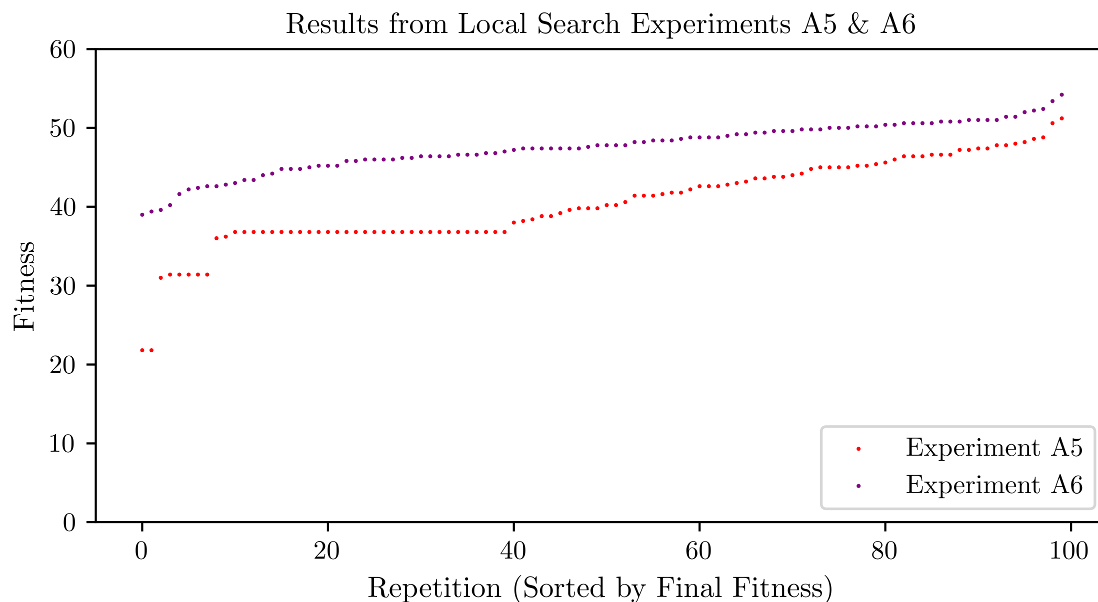
Table 7.7: Statistical data from Experiment A6. We show the steps (number of times the candidate heuristic changes), the number of evaluations, the start, end and difference in size (s) and fitness (f) of the initial (h_1) and final heuristic (h_{final}), the MTED and SMTED between the first and final heuristic, and the number of evaluations required to find the final heuristic.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	39.62	12	27.00	36.00	47.00	88
Evals	24,917.88	8,797	13,568.00	20,752.00	30,948.00	72,931
$f(h_1)$	3.06	0.00	0.00	0.80	2.60	32.00
$f(h_{final})$	47.54	39.00	46.00	47.80	50.00	54.20
Δ_f	44.48	18.60	43.00	46.00	48.50	54.20
Evals to h_{final}	17,417.88	1,297	6,068.00	13,252.00	23,448.00	65,431
$s(h_1)$	16.53	2	3.00	17.50	25.00	40
$s(h_{final})$	73.20	20	47.75	66.00	87.00	201
Δ_s	56.67	5	34.75	48.00	70.75	182
MTED(h_1,h_{final})	63.43	16	40.00	56.00	77.50	184
SMTED(h_1,h_{final})	57.45	9	35.75	49.00	70.75	182

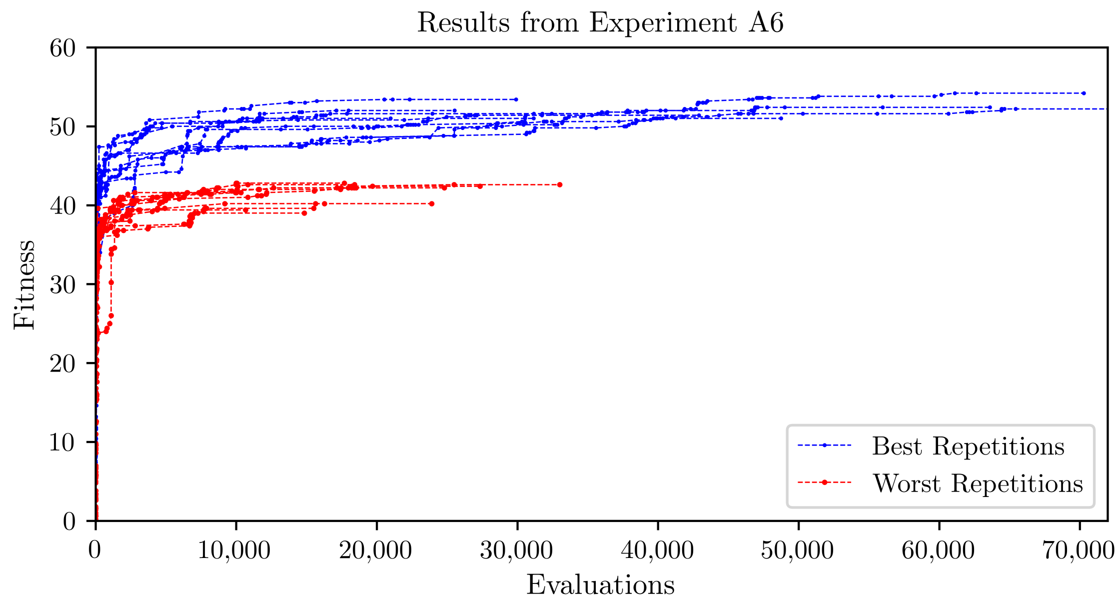
worst repetitions from the experiment progressed. In Table 7.7 we show additional statistical data pertaining to the experiment.

In the results shown in Figure 7.9a, we can immediately see that the distribution of fitness values reported from Experiment A6 is strikingly different to those reported from Experiment A5. We can see that the minimum, maximum and average fitness value of the heuristics obtained from Experiment A6 are greater than those obtained from Experiment A5. In addition to this, there are no duplicate heuristics in the results obtained from Experiment A6, whereas from Experiment A5 there were 33.

In Experiments A4 and A5 we saw large plateaus of heuristics concentrated around relatively low fitness values. This feature has only appeared in experiments which used the grow method to initialise their heuristics. As discussed in Section 7.2.2, we believe this feature to be a consequence of two things; the grow method creating a large number of very small heuristics, and the LSPS algorithm being unable to move beyond this area of the search space. In Experiment A6 we can see that there is no large plateau. When compared to Experiments A4 and A5, we can confirm that a similar proportion of repetitions from Experiment A6 were initialised with



(a) Final results from Experiment A6. Each data point represents the fitness of the final heuristic returned from that repetition of the local search algorithm. We also show the results from Experiment A5 for comparison.



(b) Ten of the best and worst repetitions from Experiment A6. Each sequence of data points represents a single repetition.

Figure 7.9: Results from Experiment A6.

heuristics of a small size. We surmise that, due to the use of the N_{alt} neighbourhood, repetitions from Experiment A6 which were initialised with a small heuristic were able to escape the area of the search space that repetitions in Experiments A4 and A5 could not, which in turn increased the overall quality of the heuristics created.

In Figure 7.9b we show how the ten best and worst repetitions from Experiment A6 progressed. This data is very different to the ten best and worst repetitions reported from Experiment A5. The majority of the final heuristics from the worst repetitions in Experiment A5 are contained within a large plateau of low-quality heuristics. In those repetitions the algorithm quickly created a relatively high-quality heuristic, but after several hundred evaluations the final heuristic was obtained and no further improvement was made. In the results for Experiment A6, the worst repetitions also create relatively high-quality heuristics, but not as quickly as in Experiment A5. However, improvements continue to be made, and the repetitions do not terminate as early as those shown for Experiment A5. A similar comparison can be made using the best repetitions from both experiments, but the difference between those results is more pronounced. Experiment A6 reported some repetitions that continued to find fitter heuristics after $50,000$ evaluations.

Table 7.7 shows us detailed statistical data regarding Experiment A6. This data backs up the observations made so far in this subsection. For every statistical metric considered, the fitness values reported from Experiment A6 are higher than those reported from every other experiment. This trend is seen in other values reported from Experiment A6 - for example, the number of steps, evaluations to obtain the final heuristic and the total number of evaluations performed are higher than those reported from any other experiment. It is worth noting that in Experiment A4 the reported values concerning the total number of evaluations are far closer to those reported from Experiment A6. However, in that experiment all neighbours were evaluated before termination occurred, and many unnecessary heuristic evaluations were performed.

On average, the heuristics created from Experiment A6 are far larger than those created from any other experiment. We can also see that the values reported for the difference in size, MTED and SMTED between the first and final heuristic are far larger than those reported from all other experiments. It is worth noting that the heuristics created from Experiment A6 are somewhat similar in size to those created from the GP experiments described in Section 4.3. One heuristic created

from Experiment A6 contains over 200 terms. While it is difficult to judge a heuristic arbitrarily, we would be surprised if it is as easily understood as those heuristics created from experiments described in previous sections. In the experiments described in this chapter we have made no attempt to control the size of the created heuristics. However in future work when using the N_{alt} neighbourhood, it may be necessary to do so in order to create heuristics that can be easily understood.

From the results presented in this subsection, we would state that the amount of time taken to create the N_{alt} neighbourhood is justified. However, the use of the N_{alt} neighbourhood does have some disadvantages; the number of evaluations required is far greater than in other experiments, and some of the created heuristics are very large in comparison to those created using the $N(3)$ neighbourhood.

The experiment described in this subsection illustrates how an alternate neighbourhood definition can be used to augment the LSPS algorithm to navigate the search space of heuristics in a more effective manner. The results have shown us that the heuristics created from Experiment A6 are of higher quality than those created from any other experiment. They are also far larger than those created from any other experiment.

7.4.3 Summary

In this section we have described an alternate neighbourhood definition, and introduced the N_{alt} neighbourhood. This neighbourhood contains a far greater number of edit sequences than the neighbourhood previously considered.

The results from Experiment A6 showed that the heuristics created using N_{alt} were of higher quality than those created from $N(3)$. We are satisfied that the additional time taken to create the N_{alt} neighbourhood is justified. The results in this section illustrate to the reader the potential of the algorithms described in Chapter 6. We can consider very large numbers of edit sequences, which can be used to navigate the solution space and, for Language A, create high-quality solutions.

In the next section we perform LSPS experiments using an alternate language, Language B.

7.5 Using an Alternate Language

In the previous sections we described experiments that used Language A and LSPS to create LS-SAT heuristics. In Chapter 3 we described the components that make up Language A. In that chapter we described many other components for building LS-SAT heuristics which have so far not been used in this thesis.

In this section we perform experiments using a different language. This language is called Language B. Like Language A, Language B is used to create LS-SAT heuristics. It is a larger language than Language A, and contains terms that correspond to more modern LS-SAT heuristic mechanisms than those used in Language A. We perform three program synthesis experiments using Language B; two LSPS experiments and one GP experiment.

The motivation behind the experiments we describe in this section is twofold; firstly, we want to ascertain whether the systems we have built are able to automatically create heuristics more effective than those hand-crafted by human experts. We believe that utilising a language which contains constructs used in modern heuristic design should provide us with the best chance of achieving this goal. Secondly, we want to know how the systems we have built for LSPS behave when given a vastly different language, and whether the observations we have made from the LSPS experiments using Language A are also seen when using Language B.

The format of this section is as follows. In Section 7.5.1 we introduce Language B, the language we use in this section. In Section 7.5.2 we present the methodology and results from the GP experiment performed using Language B. In Section 7.5.3 we present the methodology and results from Experiments B1 and B2, two LSPS experiments that use Language B. Finally in Section 7.5.4 we provide a summary of the work undertaken in this section.

7.5.1 Language B

In this subsection we present Language B, the language we use in this section to create LS-SAT heuristics. In Table 7.8 we show the terms used in Language B.

One of the motivations behind the experiments described in this section is to automatically create more effective LS-SAT heuristics. When designing Language B, we chose terms which we believed would help us to achieve this goal. The majority of the terminals and functions we chose have been used in previously existing LS-SAT

Table 7.8: Language B. All terms shown here are contained within the language, in addition to all terms in Language A except the following: the `Integer` terms -2, -1, $1 \ldots 5$, all `Age` terms, and the functions `IfTabu` and `IfNotMinAge`. In regards to the GP experiments in this chapter, terms with a grey background are in the terminal set, and those with a white background in the function set.

Type Signature	Terms
VarSet → GainType → GainType → Var	{GetBestVar2}
VarSet → GainType → Var	{GetBestVarAge}
VarSet → Var	{PickOldest}
Var → Var	{UpdatePAWS}
Maybe Var → Var → Var	{IfIsNull}
Comparator → GainType → Integer → VarSet → Maybe VarSet	{Filter}
Maybe VarSet → Maybe Var	{PickRandomM}
Maybe VarSet → GainType → Maybe Var	{GetBestVarM, GetBestVarAgeM}
VarSet → VarProb → List VarProb → Var	{WeightedVarPick}
FloatingPoint → GainType → VarProb	{ExponentFunction, Polynomial, PolynomialNegative}
VarProb → List VarProb → List VarProb	{NextElement}
GainType	{SubPosGain, SubNegGain, SubNetGain, PosGain_WA, NegGain_WA, NetGain_WA, SubPosGain_WA, SubNegGain_WA, SubNetGain_WA}
VarSet	{CONF, RBC-1, RBC_WA-0, RBC_WA-1}
Maybe VarSet	{DecrVars, DecrVars_WA, SubDecrVars, SubDecrVars_WA}
Probability	{Adapt}
Comparator	$\{>, \geq\}$
FloatingPoint	$\{0.2, 0.4 \ldots 3.8, 4.0\} - \{0.0\}$
List VarProb	{EndList}

heuristics. The reader is referred to Chapters 2 and 3 for more information regarding their meaning and examples of their previous use.

There are some functions and terminals which were included in Languages A and A1 which have been omitted from Language B. These are `IfNotMinAge`, `IfTabu`, `WFF`, all of the terminals with an `Integer` type signature (except 0), and all terminals with an `Age` type signature.

`IfTabu` was not included as many of the previously created heuristics which used this function did not perform well on larger problem instances. We believe this to be due to the much larger number of heuristic evaluations performed when solving these larger problems. Over time, we think this changes how the heuristics which use the `IfTabu` function operate. Due to it not being included, all terminals which have an `Age` type signature were also omitted.

The `WFF` terminal was omitted due to the results from the exhaustive enumeration experiment performed using Language A1. The heuristics which used the `WFF` terminal were generally less effective than those which only used the `RBC-0` terminal. We also know that including it could cause a program synthesis technique to create a subtree in the form `GetBestVar` {`WFF`, g}, where g is a gain type metric. Subtrees like this would require the overarching local search algorithm to maintain a partial ordering of all variables according to g. For large problem instances, this is computationally expensive to maintain, thus increasing the time taken for an iteration of local search to evaluate, and potentially reducing the overall effectiveness of the heuristic.

We decided to omit the majority of the terminals with an `Integer` type signature due to the results from Experiment A6. We noted that some of the created heuristics contained many combinations of terms using the `IfVarCompare` function together with different terms with an `Integer` type signature. In Section 7.6 we discuss this in greater detail, and why these combinations of terms are undesirable. To stop these combinations of terms being created in the results for the experiments which use Language B, we omitted the majority of the terms with an `Integer` type signature from the language.

However, we did include the `Integer` terminal 0. Both the functions `Filter` and `IfVarCompare` require a terminal with an `Integer` type in order to be used. In the examples of hand-crafted heuristics seen which use the `IfVarCompare` or `Filter` terminal, the `Integer` 0 has been used. We also included additional comparators, to increase the number of expressions that could be created using these functions.

We also decided to omit the `IfNotMinAge` function. This function takes three arguments; vs (which has a type signature of `VarSet`), v_1 and v_2 (which both have the type signature `Var`). `IfNotMinAge` checks to see if v_1 has the minimum age among the variables in vs. If it does, v_2 is returned, else return v_1. In the results from the exhaustive enumeration experiments performed using Language A1, we found evidence of this terminal being used in heuristics which had little semantic meaning. For example, given an arbitrary gain type g and variable v_2, a heuristic in the form `IfNotMinAge` {`WFF`, `GetBestVar` {`RBC-0`, g}, v_2} has little meaning. It is difficult to know what the intention is behind comparing the AGE of a variable obtained from `RBC-0` to the AGE of the variables in `WFF`. Language B contains many more terminals with a `VarSet` type signature, and allows variables to be created with a `Maybe VarSet` type signature. Due to this, we found little reason to include `IfNotMinAge` in Language B.

In Section 7.5.3 we present the results from the experiments performed using LSPS on Language B. We will describe two experiments in that subsection, one using the $N(3)$ neighbourhood and the other using the N_{alt} neighbourhood. In Table 7.9 we present data pertaining to the construction of the edit sequences in those neighbourhoods, and in Table 7.10 we present data concerning the frequency of different types of edit sequences in those neighbourhoods.

From the data in Table 7.9, we can see that the reported time taken to construct the edit sequences for the N_{alt} neighbourhood is somewhat similar to that reported for the N_{alt} neighbourhood for Language A. This data is quite surprising, as the number of subproblems solved for Language B is around 1.5 times that reported for Language A. Concerning the $N(3)$ neighbourhood, the number of subproblems solved for Language B is more than 20 times that reported for Language A, however the time taken is less than double. Together, this data suggests that the number of subproblems solved does not necessarily correlate with the time taken. In future work, we believe it may be useful to investigate this further using different neighbourhoods and languages, in order to more accurately determine what the relationship is between the size of the language, the number of subproblems solved and the time taken.

In Table 7.10 we show the frequency data for the edit sequences contained in the $N(3)$ and N_{alt} neighbourhoods for Language B. We can see there are significant differences when comparing this data to that reported for Language A in Table 7.5. The $N(3)$ neighbourhood for Language B contains around 10 times as many edit

Table 7.9: Data gathered when calling CREATE-ALL-EDIT-SEQUENCES on Language B using the neighbourhoods $N(3)$ and N_{alt}. We assume that the language parameter to NEIGHBOURHOOD-GENERATION contains the cost function used. So that our definitions remain consistent, the n_s parameter in Definition 27 is analogous to the n_{max} parameter in CREATE-ALL-EDIT-SEQUENCES. For the language used (L), we show the number of functions (F) and terminals (T) in that language. In brackets are the number of terms with a unique type signature in each set. For each call to CREATE-ALL-EDIT-SEQUENCES, as described by the n_s value and cost function, we show the size of the set of TPPT start states, the total number of TPPTs for which all the edit sequences were found (we refer to these as problems solved), and the time taken to compute the edit sequences.

L	F	T	n_s	Cost Function	Start States	Problems Solved	Time taken (ms)
B	21 (15)	54 (7)	3	$\text{COST}(m_{basic})$	$31,281$	$11,438,377$	$55,250$
			2.8	$\text{COST}(m_{alt})$	$56,698$	$93,907,713$	$1,187,167$

Table 7.10: Data concerning the frequency of different types of edit sequences in $N(3)$ and N_{alt} for Language B. To compute this data, the edit sequences in each neighbourhood were separated into groups based on how they change a program tree's size. We show how many edit sequences are in each group. This data only takes into account the application of single edit sequences, not compound moves. As there is a reciprocal for every edit sequence, the number in a set that increases by n or decreases by n is identical.

(a) Frequency data for the $N(3)$ neighbourhood for Language B.

Change in Size	+/-3	+/-2	+/-1	0	Total
# Edit Sequences	324	5,230	5,260	9,200	30,828

(b) Frequency data for the N_{alt} neighbourhood for Language B.

Change in Size	+/-8	+/-7	+/-6	+/-5	+/-4
# Edit Sequences	2,592,000	3,866,400	1,821,600	2,191,200	2,827,128
Change in Size	+/-3	+/-2	+/-1	0	Total
# Edit Sequences	1,653,844	991,390	3,336,668	10,777,082	49,337,542

sequences than the $N(3)$ neighbourhood for Language A. It also contains a greater number of edit sequences that make a large change to an input program tree's size.

The data for the N_{alt} neighbourhood is the most surprising. It contains nearly 5×10^8 edit sequences. This is over 40 times the number of edit sequences in the N_{alt} neighbourhood for Language A. In addition to this, a much greater proportion of the edit sequences make a larger change to an input program tree's size. We believe the large difference in the number of edit sequences is due to the much greater number of terms in Language B with identical type signatures. Consider a type-based edit which inserts a terminal with a type signature of `GainType`. Under Language A this would correspond to 3 unique term-based edits, while under Language B it would correspond to 12. In general, for an arbitrary type-based edit sequence, the number of term-based edit sequences which could be extrapolated from it under Language B would be much larger than the number extrapolated under Language A.

The language given here is one which can describe a much larger number of heuristics than Languages A or A1. As we have seen, the systems built to create neighbourhoods are able to function with this much larger language. In the next subsections we use GP and LSPS with Language B to automatically create LS-SAT heuristics.

7.5.2 GP Language B Experiment

In this subsection we present the GP experiment performed using Language B.

Methodology

The methodology of the GP experiment described in this subsection is exactly the same as the experiments described in Section 4.3, except that we use Language B instead of Language A or A1. The core motivation behind performing a GP experiment using Language B is so that we are able to compare the results with those obtained from LSPS.

Results & Analysis

In Table 7.11 and Figure 7.10 we present the results from the experiment. Table 7.11 shows some general data about the population at various points in each GP repetition. The repetition which reported the heuristic with the highest fitness value was repetition

5. In Figure 7.10 we show detailed information on how repetition 5 progressed, as well as data showing how the size of the heuristics in the population changed in that repetition.

We can see from the results in Table 7.11 that the best performing heuristic from all repetitions had a reported fitness value of 65.0. This fitness value is higher than any fitness value reported from any other heuristic in this thesis. Table 7.11 also shows us that the size of the heuristics created from this GP experiment are noticeably different than those created from the GP experiments detailed in Section 4.3. Those experiments, which used Languages A and A1, had initial populations with average sizes of between 45 and 51. The GP experiment detailed in this subsection had initial populations with average sizes of between 10 and 13.

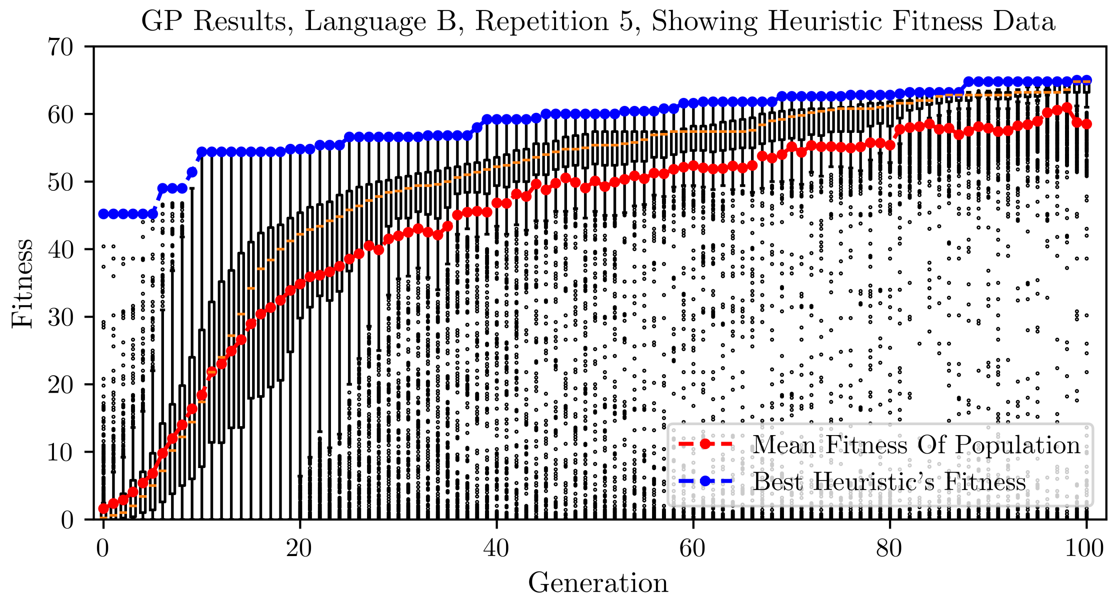
We believe these small initial average program tree sizes to be a consequence of the grow method and the terms in Language B. Language B includes a greater number of functions which, if chosen to inhabit a node at any point when using the grow method, will guarantee that the size of the subtree rooted at that node is bounded. That is to say, the subtree cannot grow indefinitely. Examples of such functions include `PickOldest`, `GetBestVar` and `GetBestVarAge`. Because of this, we believe that a larger proportion of the heuristics in the initial population were smaller than those seen in the other GP experiments, which reduced the average significantly. A similar effect was seen in some of the LSPS experiments, and discussed in Section 7.2.2.

In comparison to the results reported for the GP experiments which used Languages A and A1, the size of the heuristics created from the experiment detailed in this subsection are generally smaller. We believe this to be a consequence of the smaller initial program trees, however it could also be due to the way that Language B is formulated. The average size of the heuristics in the final generation was reported as being between 200 and 450, while for the experiments which used Languages A and A1 it was reported as being between 200 and 650.

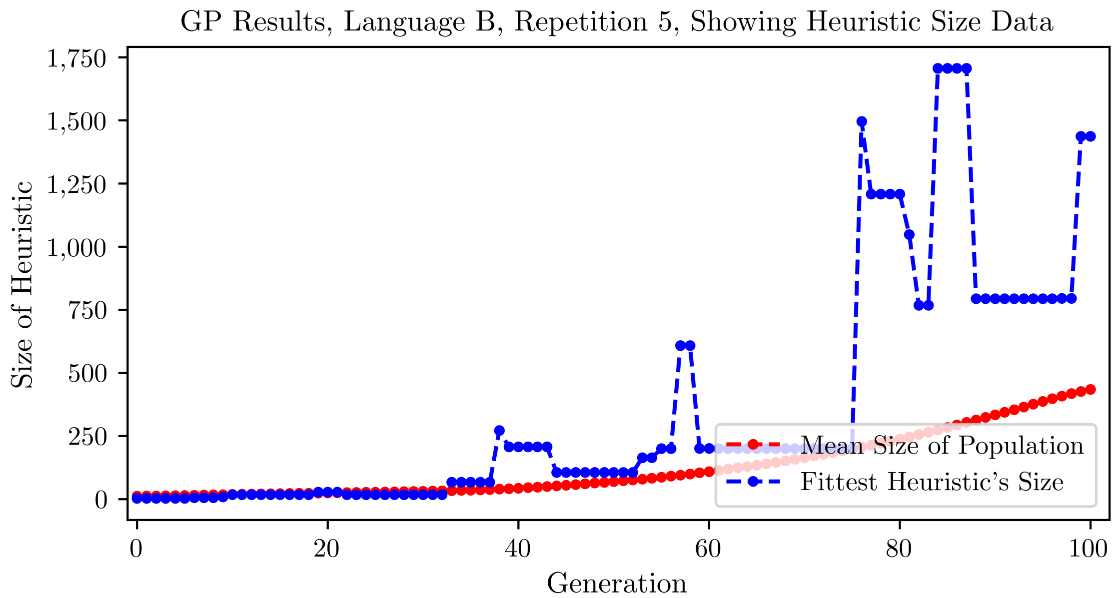
In Figure 7.10a we show how repetition 5 of the GP experiment performed using Language B progressed. This graph is similar to that shown for the best repetitions of the GP experiments performed using Languages A and A1, which we presented in Figure 4.14. However, there are some clear differences between the graphs. Firstly, we can see that the fitness of the created heuristics from Language B is higher than those created from Languages A and A1. Secondly, in the first few generations, we can see that the fitness of the population doesn't grow as quickly as in the other

Table 7.11: Statistical data pertaining to the GP experiments ran using Language B. For each repetition, we show the best heuristic's fitness, the best heuristic's size, the mean fitness of the population, and the mean size of the population for specific generations. These are the initial generation, the 25th, 50th, 75th and the 100th generation.

		Repetition 1	2	3	4	5
Initial Gen	Best Heuristic's Fitness	45.2	45.2	45.2	41.2	45.2
	Best Heuristic's Size	3	3	3	16	3
	Mean Fitness	1.49	1.55	1.61	1.82	1.59
	Mean Size	11.24	12.03	12.06	11.82	11.95
25th Gen	Best Heuristic's Fitness	56.0	59.0	56.0	56.2	56.6
	Best Heuristic's Size	55	36	45	11	18
	Mean Fitness	40.08	44.81	40.33	41.08	38.55
	Mean Size	38.02	25.75	33.9	30.99	27.84
50th Gen	Best Heuristic's Fitness	58.2	62.2	59.2	58.2	60.0
	Best Heuristic's Size	241	162	92	168	106
	Mean Fitness	48.3	51.36	48.87	47.12	50.08
	Mean Size	82.43	63.41	84.38	62.29	69.68
75th Gen	Best Heuristic's Fitness	60.8	63.8	61.2	59.8	62.6
	Best Heuristic's Size	584	810	304	159	200
	Mean Fitness	52.86	58.34	53.28	50.33	55.07
	Mean Size	228.1	148.32	188.22	134.52	198.87
100th Gen	Best Heuristic's Fitness	62.4	64.4	62.4	60.6	65.0
	Best Heuristic's Size	1,146	956	450	296	1,437
	Mean Fitness	58.41	58.94	55.99	52.96	58.52
	Mean Size	419.84	329.52	344.59	222.89	434.83



(a) Graph showing the fitness data from the 5th GP repetition. For each generation we show a boxplot detailing the distribution of fitness values in that generation, with outliers shown. We also show the mean fitness and the best heuristic's fitness.



(b) Graph showing the size data from the 5th GP repetition. We show how the mean size of the population's heuristics change as the algorithm progressed, as well as the size of the best heuristic in each generation.

Figure 7.10: Results from the 5th GP repetition performed using Language B.

GP experiments. We believe this may be an effect of the relatively small size of the program trees created in the initial generation.

In Figure 7.10b we show the graph detailing the size of the heuristics in the population as repetition 5 progressed. This graph is similar to those shown in Figure 4.15 for the GP experiments performed using Languages A and A1. However, as noted previously, the size of the heuristics for the GP experiment performed using Language B does not grow as quickly as in the experiments performed using Languages A and A1.

In Section 7.6 we present additional data pertaining to some of the heuristics created from the experiment detailed in this subsection.

From these results we would state that the use of GP with Language B has been successful. We have been able to create heuristics which have reported a higher fitness than those created using Languages A or A1. However, in future work we would suggest that alternate initialisation techniques be considered, to ascertain whether there are more appropriate choices for this language. In the next subsection we describe LSPS experiments designed to automatically create LS-SAT heuristics using Language B.

7.5.3 LSPS Language B Experiment

In this subsection we describe the methodology and present the results from the two LSPS experiments performed using Language B. We call these experiments Experiment B1 and Experiment B2.

Methodology

The methodology behind Experiments B1 and B2 is similar to the methodologies used for other experiments in this chapter. We describe them in terms of their differences compared to two previous experiments. Experiments B1 and B2 can be described as follows:

- Experiment B1 is formulated in exactly the same manner as Experiment A5, except that instead of using Language A, Language B is used.
- Experiment B2 is formulated in exactly the same manner as Experiment A6, except that instead of using Language A, Language B is used.

Both experiments were ran on the system described in Section 4.2.2. In total, Experiment B1 took 133 hours to terminate, and Experiment B2 410 hours.

Results & Analysis

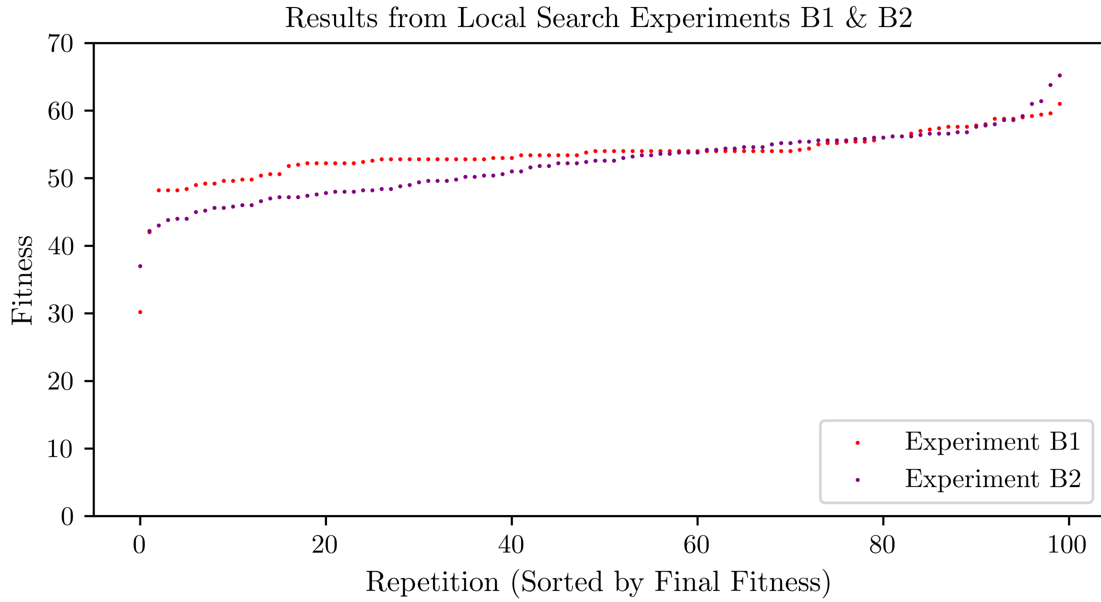
In Figure 7.11 we present three graphs; one shows the quality of the heuristics created from each repetition of both experiments, and the other two show how ten of the best and worst repetitions from each experiment progressed. In Table 7.12 we show additional statistical data pertaining to the experiments.

From the results in Figure 7.11, we can see that the minimum, maximum and average fitness value of the heuristics obtained from Experiments B1 and B2 is greater than those reported by any previous LSPS experiment. One heuristic from Experiment B2 has a fitness value of 65.2, which is greater than the highest heuristic fitness value reported from the GP experiment described in the previous subsection.

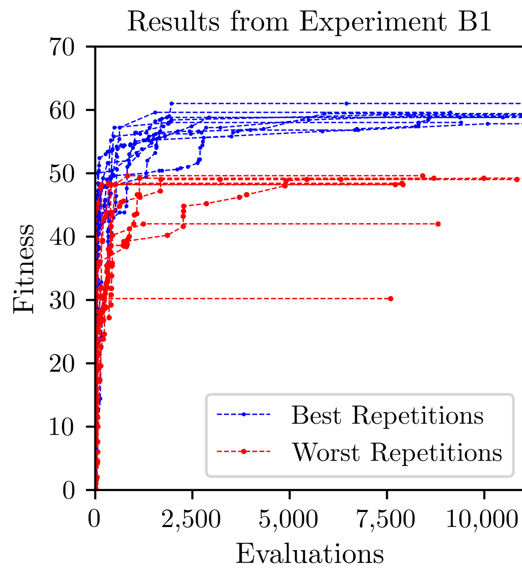
The fitness distributions reported by Experiments B1 and B2 are relatively similar to each other, however there are some subtle differences. When we consider the lowest quality heuristics from both experiments, those from Experiment B1 have a higher fitness. As we consider a greater number of repetitions, the two distributions move closer together, suggesting similar performance. Of the two experiments, Experiment B2 created the fittest heuristics.

Due to the similarity in their design, it makes sense to compare the results from Experiment A5 to Experiment B1, and from Experiment A6 to Experiment B2. Though the quality of the heuristics reported from Experiment B2 is higher than those reported from Experiment A6, the distributions are generally similar.

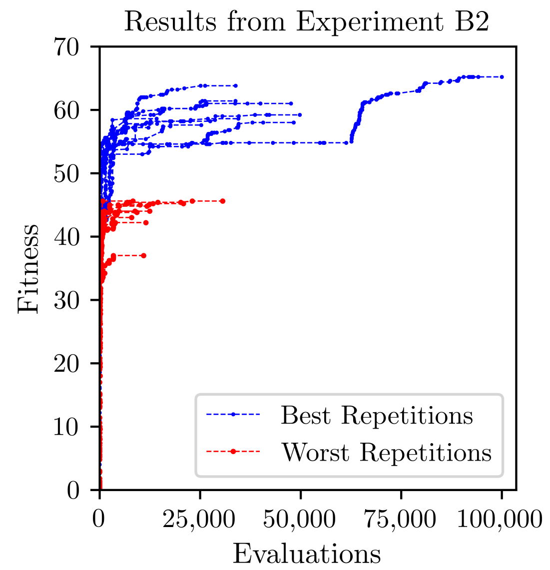
However, the same cannot be said for Experiments A5 and B1. In Experiments A4 and A5, we saw a plateau in the distributions of final heuristic fitness values. In Section 7.2.2 we surmised that its appearance was caused by the use of the grow method, and the inability of the overarching LSPS algorithm to escape from the area of the search space containing small sized heuristics. That Experiment B1 does not have a plateau is surprising. From the results in Table 7.12a, we know that a large proportion of the starting heuristics used in Experiment B1 were small. We can surmise that the adverse effect the grow method has on the size of the created heuristics is also present when using Language B. As to why there is no plateau, we believe this is due to the $N(3)$ neighbourhood for Language B containing a far greater number of edit sequences than the $N(3)$ neighbourhood for Language A, as shown in



(a) Final results from Experiments B1 and B2. Each data point represents the fitness of the final heuristic returned from that repetition of the local search algorithm.



(b) Ten of the best and worst repetitions from Experiment B1.



(c) Ten of the best and worst repetitions from Experiment B2.

Figure 7.11: Results from Experiments B1 and B2. In Figures 7.11b and 7.11c each sequence of data points represents a single repetition.

Table 7.12: Statistical data from Experiments B1 and B2. We show the steps (number of times the candidate heuristic changes), the number of evaluations, the start, end and difference in size (s) and fitness (f) of the initial (h_1) and final heuristic (h_{final}), the MTED and SMTED between the first and final heuristic, and the number of evaluations required to find the final heuristic.

(a) Statistical data from Experiment B1.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	22.78	7	14.75	20.00	30.00	48
Evals	10,586.88	7,580	7,945.00	8,901.50	12,448.75	22,085
$f(h_1)$	1.04	0.00	0.00	0.00	0.80	14.00
$f(h_{final})$	53.49	30.20	52.55	54.00	55.20	61.00
Δ_f	52.45	28.20	51.10	52.80	54.00	61.00
Evals to h_{final}	3,086.88	80	445.00	1,401.50	4,948.75	14,585
$s(h_1)$	9.29	2	3.00	4.00	14.25	38
$s(h_{final})$	19.28	7	10.00	18.00	26.00	59
Δ_s	9.99	0	6.00	8.50	13.00	27
MTED(h_1,h_{final})	16.02	4	9.00	14.00	20.25	43
SMTED(h_1,h_{final})	10.43	0	6.00	9.50	13.00	27

(b) Statistical data from Experiment B2.

	Mean	Min	Q_1	Median	Q_3	Max
Steps	47.57	8	25.75	38.50	65.00	186
Evals	24,637.49	7,712	12,864.00	18,386.50	31,810.00	100,000
$f(h_1)$	1.51	0.00	0.00	0.20	1.01	14.00
$f(h_{final})$	52.04	37.00	48.20	52.60	55.60	65.20
Δ_f	50.53	32.80	47.15	50.80	54.45	63.80
Evals to h_{final}	17,154.07	212	5364.00	10,886.50	24,310.00	94,158
$s(h_1)$	9.27	2	3.00	6.00	14.00	34
$s(h_{final})$	83.13	18	39.50	63.50	110.75	276
Δ_s	73.86	8	33.75	56.50	100.00	259
MTED(h_1,h_{final})	79.20	18	36.00	59.00	108.50	264
SMTED(h_1,h_{final})	74.06	10	33.75	57.50	100.00	259

Table 7.10a. From the data shown in Figure 7.11, evidently the greater number of edit sequences allows the LSPS algorithm to escape areas of the search space containing small sized heuristics. Under the $N(3)$ neighbourhood for Language A, it was not able to do this. In short, these results suggest that the $N(3)$ neighbourhood is better suited for navigating Language B than for navigating Language A.

The reader should note that Experiment B1 contains 17 duplicate heuristics, however they are not concentrated in one area as they were in the results for Experiments A4 and A5. Experiment B2 contains no duplicate heuristics.

When comparing the results from Experiments A5 and A6, we saw that the heuristics reported from Experiment A6 were of far higher quality. When comparing Experiments B1 and B2, there is no large difference in heuristic quality. This can be partly attributed to there being no large plateau in the results for Experiment B1. However, even when taking this into account, it appears that the $N(3)$ neighbourhood produces heuristics of a similar quality to those produced by the N_{alt} neighbourhood when using Language B.

In Figures 7.11b and 7.11c we show how the ten best and worst repetitions from Experiments B1 and B2 progressed. We can see that the repetitions under consideration from Experiment B2 took much longer to terminate than those from Experiment B1. These graphs also further illustrate the difference in heuristic quality at the extreme ends of the fitness distributions. By this we mean the best repetitions in Experiment B2 having a higher quality than the ten best from Experiment B1, and the ten worst from Experiment B2 having a lower quality than the ten worst from Experiment B1.

Due to the similarities between Experiments A5 and B1, and Experiments A6 and B2, it is natural to compare these results to those shown in Figures 7.7b and 7.9b. There are similarities in the results for each pair. Experiments A5 and B1 use a similar number of evaluations for their best repetitions. When considering the worst repetitions, the results for Experiment B1 are more varied than those for Experiment A5. This was expected, as there is no large plateau in the results for Experiment B1.

The results from Experiments A6 and B2 are strikingly similar. There are examples of repetitions in both sets of results which find fitter heuristics after tens of thousands of evaluations. Of the two, the repetitions from Experiment B2 require more evaluations to terminate. In the results for Experiment B2 we can also see an example of a repetition which used the global termination mechanism, exiting after

exactly $100,000$ heuristic evaluations.

Table 7.12 shows us detailed statistical data regarding Experiments B1 and B2. We can see that there is not much difference between the average fitness values reported from either experiment. However, when we look at the other data, we can see that our previously made observations are reinforced; that is, Experiment B2 reported heuristics with a larger range in quality than those reported from Experiment B1. This is illustrated by comparing the minimum, lower quartile, upper quartile and maximum data points for the $f(h_{final})$ statistic in each experiment.

Despite the similar average fitness scores reported, in every other data metric considered there are large differences between the experiments. The number of evaluations required is much larger in Experiment B2, as is the number of steps. The difference in size, MTED and SMTED between the first and final heuristic is also much larger in Experiment B2. In addition to this, we note that some of the created heuristics from Experiment B2 are comparable in size to some of the heuristics created from GP.

The differences in statistical data between Experiments B1 and B2 are similar to the differences we saw when we compared Experiments A5 and A6 in Section 7.4.2. If we describe the parameters and algorithm used by each experiment as that experiment's program synthesizer, then we can ask which is the best to use for creating LS-SAT heuristics. Generally, given two program synthesizers that can create heuristics with the same or similar quality, it would be preferable to use the synthesizer which requires less heuristic evaluations. To facilitate our understanding of the heuristics created, it would also be beneficial to use the synthesizer which creates the smaller heuristics.

We can justify the use of Experiment A6's synthesizer over Experiment A5's as the quality of heuristics created by Experiment A6 was much better. However it is more difficult to justify the use of Experiment B2's synthesizer over Experiment B1's. Though Experiment B2 did produce some heuristics with a higher quality than those produced by Experiment B1, on average the experiments produced heuristics of similar quality. However, Experiment B1 did this using far fewer evaluations. Based on this, we would state that Experiment B1's synthesizer is the more preferable to use.

From the results in this subsection, we can state that LSPS was able to create more effective heuristics than GP. However, we would not state that LSPS outperformed

GP. Any claim that it does would be based on a single heuristic from Experiment B2 reporting a higher fitness than the fittest heuristic produced from the GP experiment described in Section 7.5.2. The difference in fitness between these heuristics was 0.2. A better comparison between the two methods would be to consider the average fitness of the heuristics produced from each program synthesis method. Across all 100 repetitions of Experiments B1 and B2, the average fitness of the heuristics produced was 53.49 and 52.04 respectively. The average of the fittest heuristics produced from the GP experiment was 62.96. These results tell us that, on average, GP produced heuristics with the highest fitness.

Despite this, we would say that the results given in this subsection are positive. Firstly, though it may seem like a rudimentary point, they show us that LSPS is effective on a language other than Language A. The experiments described in this subsection were performed with the same parameters used in two previous experiments for Language A. The LSPS algorithm was still able to navigate the solution space effectively, and produce high-quality heuristics. We wonder if, with additional tuning for Language B, we would be able to create more effective program synthesizers.

Secondly, the experiments described in this subsection have provided us with additional data regarding the use of different neighbourhood definitions. In Section 7.4, we saw how the use of N_{alt} produced heuristics of higher quality than those created using the $N(3)$ neighbourhood. Yet in this subsection we have seen that for Language B, $N(3)$ and N_{alt} produce heuristics with a similar quality to each other. This raises the question of what type of neighbourhood definition is the best to use with a specific language. In truth, there may be neighbourhood definitions which we have not considered that produce higher quality heuristics. While we cannot determine this based on the results here, the data in this subsection does provide us with evidence that the use of neighbourhoods containing a larger number of edit sequences does not necessarily correlate with the creation of higher quality heuristics.

7.5.4 Summary

In this section we have described Language B, a language used to construct LS-SAT heuristics. We then described the GP and LSPS experiments we performed using this language, and showed the results from these experiments.

We have seen from the GP and LSPS experiments that both program synthesis techniques are able to create higher quality heuristics than those created using

Language A - at least according to the fitness function. We have also seen that, in the two LSPS experiments performed, the different neighbourhoods used had differing effects on how the LSPS algorithm progressed. By this we mean the larger heuristics that were created using Experiment B2. Despite this, the quality of heuristics created was, on average, surprisingly similar. Though LSPS was not able to outperform GP, LSPS did produce one heuristic that reported a higher fitness than the fittest heuristic found from GP.

In the next section we present examples of heuristics created from the experiments described in this chapter.

7.6 Examples of Created Heuristics

In this section we look in detail at some of the heuristics created by the experiments described in this chapter, and run some of the created heuristics on the testing set of problem instances outlined in Section 3.4.3. Each heuristic we present in this section has a name which describes its origin. For example, LS-A-3-1 is a heuristic which was produced by Experiment A3. Another example is GP-B-3, which is the heuristic with the highest fitness created from the 3rd repetition of the GP experiment described in Section 7.5.2.

The format of this section is as follows; in Section 7.6.1 we present and discuss several heuristics created from Experiments A3 to A6. In Section 7.6.2 we present and discuss heuristics created from Experiments B1 and B2, as well as show some statistical data concerning the heuristics created from the GP experiment described in Section 7.5.2. In Section 7.6.3 we present the results from running the heuristics shown in this section on the testing set outlined in Section 3.4.3. Finally in Section 7.6.4 we summarise the work in this section.

7.6.1 Heuristics Created Using Language A

In this subsection we present several heuristics created from the experiments described in this chapter which used Language A. Specifically we consider heuristics from Experiments A3 to A6. As Experiments A1 and A2 were bounded by a maximum size of 17, all the heuristics created by those experiments have been considered in Chapter 4. Therefore, we do not present any heuristics created from those experiments. In Figure 7.12 we show the heuristics we have selected from those

created by Experiments A3 to A6. All of the selected heuristics reported a high fitness value.

The design of the heuristics shown is varied, with little evidence of there being a distinct "pattern" that is indicative of a heuristic which reports a high fitness value. However, there are some similarities between the heuristics. Many of them make use of the `IfTabu` and `IfNotMinAge` terms, with some heuristics using them several times, making the resulting heuristic somewhat difficult to decipher. For example, LS-A-4-1 (shown in Figure 7.12c) contains three instances of the `IfTabu` function. Of the heuristics shown in previous chapters which use the `IfTabu` function, few of them have been effective at solving larger problem instances. However, as discussed in Section 4.4, we have previously seen evidence that suggests the fitness function is not particularly effective at identifying heuristics which perform well on larger problem instances, so it is not surprising that these heuristics have been created.

Some of the heuristics created from Experiment A6 were very different to the heuristics created from the other experiments. LS-A-6-3 (shown in Figure 7.12j) serves as an example of this. It contains several instances of the `IfVarCond` term grouped together with different terminals that have a type signature of `Integer` and `GainType`. Because of these groupings of terms, we would suggest that these heuristics are highly specialised, and appear difficult to comprehend. Though we do not show more examples here, heuristics like this were relatively common in the results from Experiment A6. Experiment A5 is identical to Experiment A6, except that it uses the $N(3)$ neighbourhood instead of the N_{alt} neighbourhood. As few of the heuristics created from Experiment A5 contained terminals in groupings we would describe as specialised, we believe that it is because of the N_{alt} neighbourhood that these types of heuristics were created by Experiment A6.

Some of the heuristics highlighted contain redundancies in their design. Specifically, LS-A-5-3, LS-A-6-1, LS-A-6-2 and LS-A-6-3 (shown in Figures 7.12g to 7.12j respectively) contain groups of terms that have no effect on which variable the heuristic picks. For example, LS-A-6-3 contains a composition of terms in the form `IfVarCond` $\{=,$ `NegGain`, $-1,\ a,\ b\}$. From the description of the `IfVarCond` function in Chapter 5, we know that the subtree denoted by b will always be returned, as the domain of the gain type representative of `NegGain` (NegGain_1) is strictly positive. In Section 4.4 we suggested two possible ways of alleviating this issue when encountered in exhaustive enumeration. One was to use methods of detecting program equivalence

```
IfVarCompare <= NegGain
 GetOldestVar
  GetBestVar RBC-0 NegGain
  IfVarCompare < NegGain
   IfTabu 5
    PickRandomVar RBC-0
    GetBestVarSnd RBC-0
      NegGain
   GetOldestVar
    GetBestVarSnd RBC-0
      NetGain
    PickRandomVar RBC-0
 GetBestVar RBC-0 PosGain
```

(a) Heuristic LS-A-3-1. Fitness value 46.6.

```
IfVarCompare < NegGain
 IfNotMinAge RBC-0
  GetBestVar RBC-0 NegGain
  IfVarCond = NegGain 5
   GetBestVarSnd RBC-0 NetGain
   GetBestVar RBC-0 PosGain
 GetOldestVar
  IfNotMinAge RBC-0
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 PosGain
  GetBestVarSnd RBC-0 NetGain
```

(b) Heuristic LS-A-3-2. Fitness value 49.0.

```
IfTabu 20
 IfTabu 50
  IfRandLt 0.7
   GetBestVar RBC-0 NegGain
   GetBestVarSnd RBC-0 NetGain
  IfTabu 10
   GetBestVar RBC-0 NetGain
   GetBestVar RBC-0 NetGain
 GetOldestVar
  GetBestVar RBC-0 NetGain
  IfTabu 5
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 PosGain
```

(c) Heuristic LS-A-4-1. Fitness value 49.2.

```
IfTabu 20
 GetBestVar RBC-0 NegGain
 IfNotMinAge RBC-0
  IfTabu 30
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 NetGain
  IfNotMinAge RBC-0
   GetBestVar RBC-0 PosGain
   GetBestVar RBC-0 PosGain
```

(d) Heuristic LS-A-4-2. Fitness value 49.4.

Figure 7.12: Ten heuristics that reported a high fitness value from Experiments A3, A4, A5 and A6.

```
IfVarCompare < NegGain
 IfVarCond = NetGain 1
  IfVarCond < PosGain 3
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 NetGain
  GetBestVar RBC-0 PosGain
 IfNotMinAge RBC-0
  IfTabu 20
   GetBestVarSnd RBC-0 NetGain
   GetBestVar RBC-0 PosGain
  IfTabu 5
   GetBestVar RBC-0 NegGain
   GetBestVarSnd RBC-0 NegGain
```

(e) Heuristic LS-A-5-1. Fitness value 48.2.

```
IfNotMinAge RBC-0
 IfTabu 10
  IfVarCompare <= NegGain
   GetBestVarSnd RBC-0 NegGain
   GetBestVar RBC-0 NegGain
  IfVarCompare < NegGain
   GetBestVar RBC-0 NetGain
   GetOldestVar
    PickRandomVar RBC-0
    PickRandomVar RBC-0
 IfVarCond = NegGain 0
  GetBestVar RBC-0 PosGain
  GetOldestVar
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 PosGain
```

(f) Heuristic LS-A-5-2. Fitness value 48.6.

```
IfNotMinAge RBC-0
 IfTabu 20
  IfVarCond = PosGain -1
   GetBestVarSnd RBC-0 NetGain
   GetBestVar RBC-0 NetGain
  GetOldestVar
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 NetGain
 IfVarCompare < NegGain
  IfNotMinAge RBC-0
   GetBestVar RBC-0 NetGain
   GetBestVar RBC-0 PosGain
  GetOldestVar
   GetBestVarSnd RBC-0 NetGain
   GetBestVar RBC-0 PosGain
```

(g) Heuristic LS-A-5-3. Fitness value 50.6.

```
IfNotMinAge RBC-0
 GetBestVar RBC-0 NegGain
 IfVarCompare < PosGain
  IfVarCond < NetGain -2
   GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 PosGain
  GetOldestVar
   GetBestVar RBC-0 PosGain
   IfVarCompare = PosGain
    IfTabu 30
     IfVarCond < PosGain -2
      PickRandomVar RBC-0
      GetBestVarSnd RBC-0 NetGain
     GetBestVarSnd RBC-0 NetGain
    GetBestVar RBC-0 PosGain
```

(h) Heuristic LS-A-6-1. Fitness value 48.6.

Figure 7.12: Ten heuristics that reported a high fitness value from Experiments A3, A4, A5 and A6. (Continued)

```
IfVarCompare < NegGain
 IfNotMinAge RBC-0
  IfVarCompare <= NegGain
   IfRandLt 0.9 { PickRandomVar RBC-0 }
    IfVarCompare < NetGain { GetBestVarSnd RBC-0 NetGain }
     PickRandomVar RBC-0
   GetBestVar RBC-0 NetGain
  GetBestVar RBC-0 PosGain
 IfTabu 30 { GetBestVarSnd RBC-0 NetGain }
  IfTabu 10 { GetBestVar RBC-0 PosGain }
  GetBestVarSnd RBC-0 PosGain
```

(i) Heuristic LS-A-6-2. Fitness value 50.6.

```
IfNotMinAge RBC-0
 IfVarCompare <= NegGain
  IfTabu 20 { GetBestVar RBC-0 NegGain }
   IfTabu 30 { GetBestVarSnd RBC-0 NetGain }
    GetBestVar RBC-0 PosGain
  IfVarCond < NetGain 2
   IfVarCond = NegGain -1 { GetBestVar RBC-0 NegGain }
    IfVarCond <= NetGain 5 { GetBestVar RBC-0 PosGain }
     IfVarCond <= PosGain 3
      IfVarCond = PosGain 1 { GetBestVarSnd RBC-0 NetGain }
       IfVarCond < NetGain 1 {GetBestVarSnd RBC-0 PosGain }
        GetBestVarSnd RBC-0
          PosGain
      IfVarCond <= NegGain 2 { GetBestVarSnd RBC-0 NetGain }
       GetBestVarSnd RBC-0 PosGain
   GetBestVar RBC-0 NetGain
 GetBestVar RBC-0 NetGain 0
```

(j) Heuristic LS-A-6-3. Fitness value 53.4.

Figure 7.12: Ten heuristics that reported a high fitness value from Experiments A3, A4, A5 and A6. (Continued)

and the other was to use a more expressive type system that prohibits comparisons like this. For LSPS, only the latter of these methods would be appropriate.

In Section 7.6.3 we present the results from running the heuristics shown here on the testing set of problem instances outlined in Section 3.4.3.

7.6.2 Heuristics Created Using Language B

In this subsection we look in greater detail at some of the heuristics created from Experiments B1, B2, and the GP experiment described in Section 7.5.2.

GP Created Heuristics

The fittest heuristics created from the GP experiment performed using Language B are too large to present here, with some heuristics containing over 1,000 terms. However, in Appendix B we present the fittest of them, GP-B-5.

Though we do not show all of the heuristics created by the GP experiment, we do present some data which provides insight into their construction. In Table 7.13 we show the number of times specific terms appear in each of the fittest heuristics created from each repetition. By presenting frequency data in this manner, we are able to see if any terms are more or less prevalent in the created heuristics.

We can see that there are some terms in Language B which are not often used. Two examples are the `SubDecrVars` and `SubDecrVars_WA` terms. When designing these two specific language components, we had no prior evidence that the mechanisms which these terms represent would be effective in LS-SAT heuristic design. Therefore, we are not particularly surprised they were seldom used. However, we were surprised that the `Filter` function was not utilised more often. In Section 2.3 we saw an example of an effective, hand-crafted heuristic that used a filtering mechanism, which the `Filter` function was based on. Because of this, we had anticipated that it would be a more commonly used component in the automatically created heuristics.

There are some terms in Language B which are used frequently in some heuristics, and hardly used in others. For example, the `WeightedVarPick` function is used 33 times in GP-B-1, 13 times in GP-B-3 but only 3 times in the other heuristics. Other terms which are used often in some heuristics but not often in others include `PickRandomVar`, `PickOldest`, `UpdatePAWS`, `PickRandomM` and `GetBestVar`. We were somewhat surprised that there were such large differences in the number of times these terms were used in each heuristic.

Table 7.13: Statistical data concerning the frequency of terms used in the fittest heuristic returned from each repetition of the GP experiments performed using Language B. We show all functions and all those terminals which are not numerical values or comparators.

Term	GP-B-1	GP-B-2	GP-B-3	GP-B-4	GP-B-5	Term	GP-B-1	GP-B-2	GP-B-3	GP-B-4	GP-B-5
PickRandomVar	12	0	1	0	16	RBC-1	35	27	1	11	37
GetBestVar	8	23	8	7	23	RBC_WA-0	9	28	5	0	13
GetBestVarSnd	15	4	2	4	25	RBC_WA-1	42	8	20	3	26
GetOldestVar	24	13	14	11	31	CONF	37	61	18	20	91
IfRandLt	62	50	15	15	30	PosGain	13	25	4	8	22
IfVarCond	40	20	18	13	77	NegGain	39	24	15	4	28
IfVarCompare	27	57	4	3	40	NetGain	34	50	14	12	21
UpdatePAWS	57	0	16	0	5	SubPosGain	8	4	2	3	32
PickOldest	17	0	4	0	9	SubNegGain	15	0	1	2	20
GetBestVarAge	53	87	20	19	65	SubNetGain	18	5	5	1	18
GetBestVar2	16	24	4	10	38	PosGain_WA	21	19	4	3	6
IfIsNull	19	34	23	11	88	NegGain_WA	5	39	4	0	33
PickRandomM	0	0	8	2	10	NetGain_WA	35	62	27	26	117
GetBestVarM	5	25	6	4	40	SubPosGain_WA	35	40	9	5	42
GetBestVarAgeM	14	9	9	5	38	SubNegGain_WA	8	8	5	7	25
Filter	1	0	0	1	3	SubNetGain_WA	15	0	5	8	26
WeightedVarPick	33	3	13	3	3	Adapt	7	1	3	7	14
NextElement	18	0	7	0	0	EndList	33	3	13	3	3
ExponentFunction	10	3	3	0	3	DecrVars	9	15	16	9	56
Polynomial	21	0	7	3	0	SubDecrVars	4	0	0	0	0
PolynomialNegative	20	0	10	0	0	DecrVars_WA	5	14	7	0	29
RBC-0	32	17	8	10	15	SubDecrVars_WA	0	5	0	1	0

The final point we would like to draw the attention of the reader to concerns how often the twelve terminals which have a `GainType` type signature were used. Terms that represent the three "base" gain type metrics were the most frequently used, then came terms which represent the dynamically weighted variants of those base metrics. We had assumed that the weighted variants would be used more frequently, as dynamic clause weighting is commonly used in some highly effective hand-crafted LS-SAT heuristics. All six terms that represent the Sub gain type variants were used far less frequently than the others. In hand-crafted heuristics these metrics have previously been used in tie-breaking mechanisms (see Section 2.3.6), and we were slightly surprised they were not more frequently used in the heuristics created from the GP experiments.

LSPS Created Heuristics

In Figure 7.13 we show four heuristics taken from the results of Experiments B1 and B2. These heuristics have been chosen as they reported some of the highest fitness values from the experiments under consideration.

We can see several similarities between the heuristics created from both experiments. All heuristics presented contain at least one term that uses dynamic weighting, and one occurrence of the `CONF` term. We can also see that heuristics created from both experiments contain several instances of terms that represent the Sub variants of the gain type metrics. We were somewhat surprised by this, as the heuristics created from GP did not use these terminals very often. Like those heuristics presented in Section 7.6.1, the heuristics from both experiments frequently used terms from a small subset of the language; for Experiment B1 the terms `GetOldestVar`, `UpdatePAWS`, `IfIsNull` and `IfVarCompare` were used most often. For Experiment B2, the frequently used terms were `IfVarCond`, `WeightedVarPick`, `IfIsNull` and `IfRandLt`. Though we do not show them here, these trends were seen in the other heuristics created from both experiments.

When we created Language B, we only included one term with an `Integer` type signature. We made this design choice with the intention of reducing the number of times the created heuristics could use the `IfVarCond` function. We did this as, when analysing the heuristics created from Experiment A6, we found that some used that function many times, and in configurations that appeared to make the heuristic highly specialised. An example of this was seen in LS-A-6-3 (shown in Figure 7.12j).

```
UpdatePAWS
 IfVarCompare > NetGain
  GetOldestVar
   GetBestVarAge RBC-0 PosGain
   IfIsNull
    GetBestVarAgeM DecrVars_WA
      SubNetGain
    GetBestVar2 RBC-0
      SubNetGain_WA NegGain
  IfIsNull
   GetBestVarAgeM DecrVars
     NegGain
   GetBestVarAge CONF NetGain_WA
```

(a) Heuristic LS-B-1-1. Fitness value 59.0.

```
UpdatePAWS
 IfIsNull
  GetBestVarM DecrVars
    SubNetGain_WA
  IfVarCompare >= NetGain_WA
   GetOldestVar
    GetBestVarAge RBC_WA-0
      NetGain_WA
    GetBestVar2 CONF
      NetGain_WA SubNetGain_WA
   PickOldest CONF
```

(b) Heuristic LS-B-1-2. Fitness value 59.6.

```
IfVarCond >= NetGain_WA 0
 IfRandLt 0.7 { GetBestVarAge CONF NetGain_WA }
  IfIsNull { GetBestVarM DecrVars SubPosGain }
   IfVarCond >= NetGain_WA 0
    IfVarCond > SubPosGain_WA 0
    GetBestVar2 CONF NegGain_WA SubNetGain
     GetBestVar2 CONF NetGain SubNegGain_WA
    GetBestVar2 CONF SubNetGain_WA NetGain_WA
 IfVarCond > SubPosGain_WA 0
  IfVarCond > SubPosGain_WA 0
   GetBestVar2 RBC-0 SubNegGain SubNegGain_WA
   IfVarCond >= SubNetGain_WA 0
    GetBestVar2 RBC-0 SubNetGain NetGain
    GetBestVar2 CONF NetGain SubNegGain_WA
  IfVarCond <= SubPosGain 0
   WeightedVarPick CONF { Polynomial 2.6 NetGain } EndList
   GetBestVar2 CONF SubPosGain_WA SubNetGain
```

(c) Heuristic LS-B-2-1. Fitness value 59.2.

Figure 7.13: Four heuristics that reported a high fitness value from Experiments B1 and B2.

```
let A =
IfVarCond > NetGain_WA 0
 IfVarCond <= SubNetGain 0
  IfVarCond >= PosGain 0
   IfVarCompare < SubPosGain
    GetBestVar2 RBC-1 SubNetGain SubPosGain_WA
    IfVarCond >= NetGain 0 { PickRandomVar RBC-0 }
     IfVarCond <= SubNetGain_WA 0
      GetBestVar2 RBC_WA-0 PosGain PosGain_WA
      GetBestVar2 RBC_WA-1 SubPosGain PosGain_WA
   GetBestVar2 RBC_WA-1 SubNegGain_WA SubNetGain
  IfVarCond > SubNetGain_WA 0
   IfVarCond >= NetGain_WA 0
    GetBestVar2 RBC-1 NetGain SubPosGain_WA
    IfVarCond = SubNegGain 0
     WeightedVarPick RBC-1 { ExponentFunction 2.0 SubNegGain_WA }
      NextElement { Polynomial 2.8 NegGain_WA }
       NextElement { ExponentFunction 3.8 SubNetGain } EndList
     IfVarCond <= PosGain_WA 0 { GetBestVar2 RBC-0 NegGain PosGain }
      GetBestVar2 CONF NetGain NetGain_WA
   IfVarCond >= SubNetGain 0
    IfVarCond < SubNetGain 0
     IfVarCond <= SubNegGain_WA 0
      IfVarCompare <= SubNegGain_WA
       GetBestVar2 RBC_WA-1 NegGain_WA NegGain
       IfIsNull { GetBestVarM DecrVars SubNegGain }
        IfVarCond >= SubNetGain 0
         GetBestVar2 CONF NegGain_WA NetGain
         IfVarCond <= SubNegGain 0 { PickRandomVar RBC_WA-1 }
          GetBestVar2 RBC_WA-0 NetGain PosGain
       PickOldest RBC-1
     GetBestVar2 RBC_WA-0 SubPosGain SubPosGain_WA
    WeightedVarPick RBC_WA-0 { Polynomial 3.2 PosGain_WA }
     NextElement { ExponentFunction 0.4 NetGain }
      NextElement { ExponentFunction 0.8 SubNegGain } EndList
 GetBestVar2 CONF NetGain_WA NegGain_WA
```

(d) Part A of the heuristic LS-B-2-2. Fitness value 65.2.

Figure 7.13: Four heuristics that reported a high fitness value from Experiments B1 and B2. (Continued)

```
UpdatePAWS
 IfRandLt 0.1
  IfRandLt 0.7
   IfIsNull
    GetBestVarAgeM SubNetGain { Filter = NetGain 0 CONF }
    IfVarCond = SubNegGain_WA 0
     IfVarCond >= NetGain_WA 0
      IfVarCond <= SubPosGain 0
       IfRandLt 0.3 { GetBestVar2 CONF SubNegGain_WA NegGain }
        IfVarCond = PosGain 0 { PickOldest CONF }
         WeightedVarPick RBC-1 { Polynomial 2.6 PosGain }
          NextElement { Polynomial 3.4 SubPosGain } EndList
       WeightedVarPick RBC-1 { Polynomial 1.4 SubPosGain_WA }
         NextElement { ExponentFunction 2.0 NegGain_WA } EndList
       PickOldest RBC_WA-0
    GetBestVar CONF NegGain
   IfVarCond > SubPosGain_WA 0
    IfVarCond > NetGain_WA 0
     IfIsNull
      GetBestVarAgeM PosGain_WA { Filter = NegGain_WA 0 RBC-1 }
      A
     IfVarCond > SubNetGain_WA 0
      IfVarCond > SubNegGain 0 { GetBestVar2 RBC-0 PosGain NetGain }
       IfVarCond > SubNegGain 0
        GetBestVar2 RBC_WA-1 SubNetGain_WA SubNegGain_WA
        GetBestVar2 RBC-0 SubNetGain_WA PosGain_WA
      GetBestVar2 RBC_WA-0 SubPosGain_WA SubPosGain
    GetBestVar2 RBC_WA-0 NegGain PosGain_WA
  GetBestVarAge CONF NetGain_WA
```

(e) Final part of the heuristic LS-B-2-2. Fitness value 65.2.

Figure 7.13: Four heuristics that reported a high fitness value from Experiments B1 and B2. (Continued)

Despite our attempts to direct the program synthesis methods away from heuristics containing many instances of `IfVarCond`, we can see that Experiment B2 still produced some heuristics like this. For example, LS-B-2-1 and LS-B-2-2 (shown in Figures 7.13c to 7.13e) contain 6 and 23 instances of the `IfVarCond` function respectively. Like Experiment A6, Experiment B2 used the N_{alt} neighbourhood. In future work we believe it may be beneficial to investigate this phenomenon further, in an attempt to identify strategies that would ensure less specialised heuristics were created when using the N_{alt} neighbourhood.

Finally, we would like to draw the attention of the reader back to the heuristic LS-B-2-2. This heuristic reported the highest fitness value of any heuristic in this thesis. It is also very large in comparison to other heuristics created from LSPS. We believe its size makes it difficult to understand how its various components work together.

However, the composition of this heuristic does have one interesting property. If we look at the second line in Figure 7.13e, we can see that a large portion of the heuristic is only evaluated 10% of the time. The other branch of this `IfRandLt` function is evaluated 90% of the time. That other branch is represented by the subtree `GetBestVarAge` {`CONF`, `NetGain_WA`}. This is a heuristic in its own right, and has a fitness value of 45. This heuristic serves as a stark reminder of the difficulty we have in understanding effective heuristic design; it appears unintuitive that a part of the heuristic that is only evaluated 10% of the time can have such a large effect on its overarching quality.

In the next subsection we present the results from running the heuristics shown here on the testing set of problem instances outlined in Section 3.4.3.

7.6.3 Testing Set Results

In this subsection we present the results from running the heuristics presented in this section on the testing set outlined in Section 3.4.3. In Table 7.14 we show the results from running the heuristics shown in Figures 7.12 and 7.13, and the heuristics created from the GP experiment detailed in Section 7.5.2, on the testing set.

Like those heuristics from the exhaustive enumeration and GP experiments in Chapter 4, all of the tested heuristics performed well on the first five subsets of problem instances. These subsets contain problems that are of a similar size to those used in the fitness function. Some of the heuristics were particularly effective on these

Table 7.14: Results from running the heuristics in Figures 7.12 and 7.13, and those created from the GP experiment using Language B on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subset of problem instances.

(a) Results from running the heuristics in Figures 7.12a to 7.12g on the testing set.

	Heuristic						
Subset Name	LS-A-3-1	LS-A-3-2	LS-A-4-1	LS-A-4-2	LS-A-5-1	LS-A-5-2	LS-A-5-3
`uf50`	99.9 0.0006	**100.0** **0.0006**	100.0 0.0006	99.9 0.0006	99.9 0.0005	99.8 0.0005	100.0 0.0006
`uf100`	99.8 0.0031	**100.0** **0.0027**	100.0 0.003	98.4 0.003	99.2 0.0022	99.6 0.0021	99.8 0.0026
`uf150`	99.2 0.0117	99.0 0.0106	**100.0** **0.0112**	99.2 0.012	100.0 0.0132	99.8 0.0105	99.8 0.0082
`uf200`	96.8 0.0247	97.0 0.0285	**99.8** **0.096**	96.8 0.0307	95.8 0.0263	96.8 0.0336	97.6 0.0278
`uf250`	**100.0** **0.0582**	98.0 0.0383	100.0 0.0776	98.2 0.0542	97.4 0.0366	96.6 0.0357	97.0 0.0271
`ufv4000`	70.0 18.9605	60.0 13.8333	36.0 13.6875	28.0 6.947	64.0 18.469	**82.0** **10.5402**	72.0 10.4821
`ufv7000`	52.0 17.1531	50.0 17.0128	6.0 3.3386	0.0 0.0	62.0 24.7351	**96.0** **26.5143**	70.0 17.9449
`ufv10000`	40.0 19.0713	26.0 20.4782	0.0 0.0	0.0 0.0	36.0 21.3834	**82.0** **29.2845**	70.0 26.8335
`ufv13000`	0.0 0.0	10.0 6.8313	0.0 0.0	0.0 0.0	4.0 2.4962	34.0 17.8564	**38.0** **18.2735**
`ufv16000`	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	**30.0** **15.2063**	0.0 0.0
`ufv19000`	10.0 9.2854	10.0 2.3986	0.0 0.0	0.0 0.0	0.0 0.0	**22.0** **7.0932**	20.0 11.9442

Table 7.14: Results from running the heuristics in Figures 7.12 and 7.13, and those created from the GP experiment using Language B on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subset of problem instances. (Continued)

(b) Results from running the heuristics in Figures 7.12h to 7.12j and 7.13 on the testing set.

	Heuristic						
Subset Name	LS-A-6-1	LS-A-6-2	LS-A-6-3	LS-B-1-1	LS-B-1-2	LS-B-2-1	LS-B-2-2
`uf50`	99.7 0.0006	99.8 0.0006	99.9 0.0008	**100.0** **0.0013**	100.0 0.0014	100.0 0.0021	100.0 0.0019
`uf100`	99.0 0.0029	99.4 0.003	99.6 0.0026	100.0 0.008	**100.0** **0.0062**	100.0 0.0095	100.0 0.0113
`uf150`	99.0 0.0108	99.6 0.011	**100.0** **0.0155**	100.0 0.0305	100.0 0.0325	100.0 0.0431	100.0 0.052
`uf200`	95.2 0.0296	96.6 0.0493	96.0 0.0248	**99.4** **0.1738**	98.8 0.1628	98.6 0.2125	98.8 0.2398
`uf250`	99.2 0.0383	99.2 0.0451	**100.0** **0.0668**	98.8 0.1263	99.6 0.2128	99.2 0.2809	99.0 0.2527
`ufv4000`	**80.0** **15.8297**	68.0 17.574	24.0 6.1259	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
`ufv7000`	**92.0** **24.7882**	58.0 21.588	8.0 7.1471	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
`ufv10000`	**78.0** **26.0185**	36.0 18.1516	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
`ufv13000`	**58.0** **27.8722**	6.0 3.7854	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
`ufv16000`	**70.0** **35.2942**	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
`ufv19000`	**70.0** **41.4527**	4.0 3.6851	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0

Table 7.14: Results from running the heuristics in Figures 7.12 and 7.13, and those created from the GP experiment using Language B on the testing set. For each problem p and heuristic h, we ran h on p five times. We report the average percentage of problems solved in each subset, and the average time (in seconds) each heuristic took to solve those problem instances. Bold typeface shows the best performing heuristic on that subset of problem instances. (Continued)

(c) Results from running the heuristics created from GP and Language B on the testing set.

	Heuristic				
Subset Name	GP-B-1	GP-B-2	GP-B-3	GP-B-4	GP-B-5
uf50	100.0 0.004	100.0 0.0035	100.0 0.0029	**100.0** **0.0027**	100.0 0.0048
uf100	100.0 0.0253	100.0 0.0277	**100.0** **0.016**	100.0 0.0178	100.0 0.0249
uf150	100.0 0.0911	98.2 0.0678	100.0 0.063	**100.0** **0.0504**	100.0 0.1339
uf200	98.4 0.305	88.2 0.2453	99.0 0.2257	**99.0** **0.1793**	97.8 0.3243
uf250	99.0 0.4082	92.0 0.273	**99.8** **0.338**	99.0 0.2439	99.0 0.4035
ufv4000	0.0 0.0	0.0 0.0	**10.0** **6.0337**	0.0 0.0	6.0 3.2716
ufv7000	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
ufv10000	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
ufv13000	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
ufv16000	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
ufv19000	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0

problem instances; for example, LS-A-3-1 was the best performing of all heuristics considered in this thesis on `uf250`. LS-A-3-2, LS-A-4-1 and LS-A-6-1 also had good general performance on these smaller instances.

When looking at the heuristic's performance on the five subsets containing smaller problem instances, we can see that the heuristics created from Language B did not perform as well as those created from Language A. Those from Language B were generally slower at solving problem instances, and unable to solve as many as those heuristics from Language A. This was seen in the results for the heuristics created from GP and LSPS. There are several terms in Language B which, when included in a heuristic, change that heuristic's update function. For example `DecrVars` and `CONF`, two terminals which represent dynamic sets of variables, require additional mechanisms to ensure the sets representing their variables are maintained correctly. These additional update mechanisms can be computationally expensive to perform. Other examples of terms which can add additional computational overhead to the overarching update function are those which use dynamic clause weighting, such as `NetGain_WA`. We believe that this is why these heuristics performed worse on the subsets of problem instances containing smaller problems when compared to the heuristics created from Language A.

We can also see that the heuristics created from Language B were unable to solve any notable number of elements in the subsets containing larger problem instances. This is true for the heuristics created from both GP and LSPS. This result is particularly disappointing, as we had hoped that a more verbose language would allow higher quality heuristics to be created that would excel at solving larger problem instances. We believe the reason these heuristics aren't particularly effective is due to the additional time they require to update their auxiliary data structures. In future work it may be worth refining Language B by removing those terms whose update mechanisms incur a large computational overhead. Such a refined language may make it easier for effective heuristics to be automatically created.

We can also see that some of the heuristics from Language A were not particularly effective on the subsets containing larger problem instances. As we discussed in Section 4.4, we do not believe our fitness function to be well suited for identifying heuristics that are effective on larger problem instances. It may be the case that the fitness function played a role in the creation of ineffective heuristics from the experiments performed using Language B.

There were some heuristics created from Language A that are highly effective on the larger problem instances. We would highlight LS-A-5-2 and LS-A-6-1 as having the best performance. Specifically, LS-A-5-2 had the best performance on `ufv4000` of all heuristics tested in this thesis, and very good performance on `ufv7000`, however on the largest problem instances it did not perform as well. LS-A-6-1 had the best performance of any heuristic on `ufv10000`, `ufv13000`, `ufv16000` and `ufv19000`. Of all the heuristics we have tested in this thesis, we would state that it appears to have the best performance.

7.6.4 Summary

In this section we have looked in detail at some of the heuristics created from the LSPS and GP experiments described in this chapter. We have also ran these highlighted heuristics on the testing set, to determine how effective they are at solving a range of different sized problem instances.

We have seen that the highest quality heuristics created from the experiments were generally different to each other, with there appearing to be no common structure or grouping of terms indicative of an effective heuristic. We also saw how some heuristics included redundancies in their design. Of particular note were the heuristics created using the N_{alt} neighbourhood, some of which appeared to be overfitted and highly specialised for the fitness function.

The performance of the heuristics on the testing set reinforced our observations from Chapter 4, in that heuristics which reported a high fitness value were not necessarily effective at solving large problem instances. We note that none of the heuristics created using Language B were particularly effective on the testing set. However, some created from Language A were. Of particular note is the heuristic LS-A-6-1, which had the best performance on the testing set of all heuristics considered in this thesis.

In the next section we summarise the work in this chapter, and discuss potential avenues of future research.

7.7 Discussions & Conclusions

In this chapter we have performed eight experiments using LSPS. The first four of these built on the work in Chapter 5, creating an LSPS algorithm which requires no

memoized data. The results from these experiments showed that LSPS is effective on our domain, and able to create high-quality LS-SAT heuristics. The remaining four LSPS experiments can be considered extensions to the initial four. These extensions utilised a randomised neighbourhood generation algorithm, an alternate neighbourhood definition and an alternate language. We have also shown examples of heuristics created through LSPS which appear to have good performance on large problem instances. In the results from running them on the testing set, some were able to solve a greater proportion of the larger problem instances than any other heuristics considered in this thesis.

The core goal of this chapter was to determine whether LSPS is a viable technique for creating LS-SAT heuristics. Based on the results we have seen, we would state categorically that it is. We have also shown that it is a flexible technique, as we have been able to use different languages, neighbourhood generation algorithms and neighbourhood definitions with it. However, we would say there is one shortcoming with its use. When analysing the results from the GP experiments performed in this thesis, the heuristics created had consistently high fitness values. Whereas for each of the LSPS experiments whose results we have reported, the created heuristics had fitness values which spanned a relatively wide range. The majority of these heuristics had fitness values below those reported by the GP experiments. In essence, the LSPS algorithm may need to be ran several times to find a high-quality solution.

The LSPS experiments that have been described in this chapter have given us insight into how the neighbourhood generation algorithm can be used as a program synthesizer. This work directly links back to one of our original goals of finding alternate methodologies for automating the creation of heuristics. Whereas Experiments A1 to A4 concentrate on creating LS-SAT heuristics using Language A, we feel that the other experiments illustrate the neighbourhood generation algorithm's flexibility. To us they also suggest that LSPS may be amenable to being deployed in different problem domains to automatically create heuristics, or to potentially solve general program synthesis problems.

There are many outstanding research questions about LSPS beyond which other domains it can be used in. Perhaps the most obvious of these are asked as a direct consequence of LSPS's similarity to generic local search. One pertinent question for our research is what effect do different initialisation functions have on the overall quality of the solutions created. The LSPS algorithms we have used in our experiments

can be described as hill-climbing algorithms, as they always move to a fitter neighbour. One potential avenue of future research could be to investigate what effect would allowing the algorithm to move to lower quality neighbours have on the quality of solutions created. We could also consider augmenting the algorithm with other generic local search mechanisms such as tabu lists or random restarts.

There are other questions that can be asked regarding the components used in LSPS. In Section 7.3 we used the randomised neighbourhood generation algorithm to generate neighbours, and continued to use it in subsequent sections. We used data gathered from previous experiments to approximate weights, which were then used with a weighted pick function to find the number of compound moves to perform. We believe that there may be ways of augmenting this algorithm to make it fairer in how it decides how many compound moves it performs. We also believe that it can be made fairer in how it decides which edit sequences to apply to an input program tree. While these changes may not improve the quality of solutions created, they would help in removing bias from the randomised neighbourhood generation algorithm. In turn, this could allow further strategies to be used to direct the neighbourhood generation algorithm in the choosing of neighbours.

In Section 6.7 we stated that, in this thesis, we do not perform experiments with neighbours that can only be obtained from the application of a single edit sequence. However, we believe it could be beneficial to understand how the results obtained from the experiments performed in this chapter would change if a different number of compound moves were used. We also feel that the results presented in this chapter raise additional research questions regarding the relationship between neighbourhood definitions and languages. We saw that for Language A, the N_{alt} neighbourhood created heuristics with a higher quality than those created using the $N(3)$ neighbourhood. However for Language B, both neighbourhoods created heuristics of a similar quality. Understanding why this is the case could help us to successfully use LSPS in other domains, as well as allow us to consider other neighbourhood definitions that could improve the quality of the created heuristics.

There is also the potential for using LSPS to augment other program synthesis methods. For example, it is entirely possible to take the heuristics obtained from GP and use LSPS on them, in an attempt to create heuristics with a higher fitness. LSPS could also be used to diversify the population in a GP run, or even as a genetic operator. One of the drawbacks of using GP is that it can suffer from bloat. We have

seen in the experiments performed that LSPS can create final solutions which are very large - in essence, it appears to have its own issues with bloat. One potential way of alleviating this is to control the size of the heuristics that can be created using LSPS. As an alternate to using a hard limit on the number of terms allowed in a heuristic, we could augment the neighbourhood generation algorithm to favour heuristics with a smaller size over larger ones, but still allow large ones to be created if it is deemed necessary to progress the search.

The suggestions made here only serve as examples of the possible areas of future research that could be performed using LSPS. While the results shown in this chapter are promising, they only provide evidence that LSPS can be effectively used in a single domain. We believe that the primary goal in any future research should be to use LSPS on other domains. By doing so, we can begin to ascertain how viable it is as a general-purpose program synthesis technique.

This concludes the final research chapter in this thesis. In the next chapter we present our conclusions to the work in this thesis, as well as highlight potential avenues of future research

Chapter 8

Conclusions

8.1 Context

Work in automated heuristic creation, usually categorised as being in generative hyper heuristics, is often focused on the use of evolutionary computation techniques to achieve its goal. To date, research has predominantly focused on genetic programming [148, 98, 27, 40, 101], with some examples of other evolutionary computation techniques being used such as grammatical evolution [53] and gene expression programming [153]. Within wider program synthesis research, there are many other techniques which can be used to automatically create programs or program fragments, however there has been little effort expended in utilising them in the domain of automated heuristic creation. The core focus of this thesis is in the use of alternate program synthesis methods to automatically create heuristics. This is a topic of particular research interest, as alternate program synthesis methods may be more effective than those used currently, and may be able to create higher quality heuristics.

This thesis has analysed how three program synthesis methods perform when used to automatically create heuristics for a local search-based SAT solver. These methods were genetic programming, exhaustive enumeration and local search program synthesis. We found that, while the heuristics created from exhaustive enumeration and local search program synthesis were of lower quality than those created from genetic programming, they were also smaller and easier to understand. These two methods also produced heuristics which, on the testing set of problem instances, were able to outperform state-of-the-art, hand-crafted heuristics.

There are several examples of previous research where the goal was to automate

the creation of LS-SAT heuristics. Fukunaga [60, 63, 61] used a bespoke GP implementation to achieve this goal, while Poli et al. [147] used a grammar-based GP approach. KhudaBukhsh et al. [100, 99] described SATenstein, a system that uses automated algorithmic configuration to automatically create LS-SAT heuristics. The language used by SATenstein contains terms which correspond to mechanisms used in modern LS-SAT heuristic design such as clause weighting. In this way, it shares some similarities with the extended language we used in Chapter 7. However, the number of heuristics that can be created by SATenstein is finite, unlike our system which can create an infinite number of heuristics.

One of the program synthesis methods used in this thesis was a new method which we called local search program synthesis. While we have used it to create heuristics, it has been described in such a way so that it can be used with any program synthesis problem whose associated language is defined in terms of a type system, and we believe it is easily augmented to work with a CFG. Within wider program synthesis research, there has been relatively little work in utilising local search-based methods. However, there are some examples, such as work by Azad and Ryan [8], who described a form of local search on program trees termed "hyper mutation". Hyper mutation works by trying to optimise the functions in a candidate program tree, in an attempt to find the combination which provides the best performance. Other researchers have described similar experiments where the algorithm was designed to optimise the combination of leaves in a program tree, rather than the functions [71, 182].

Some researchers have identified local search mechanisms similar to those used in this thesis. For example, Juárez-Smith, Castelli, and Z.-Flores [92] described a domain-specific local search mechanism for symbolic regression, which works by augmenting the candidate program tree by inserting a new root node above the previously existing one. O'Reilly [135] used hill-climbing and simulated annealing as the basis for creating program synthesizers. The author used a generalised local search mechanism based on the same tree edits used in this thesis. However, that research was based on languages that used an untyped program representation, and worked by performing random edits on the trees to ensure they were syntactically correct. Yuan and Banzhaf [178] presented a local search mechanism similar to that described in this thesis, designed to work on languages which use a type system. The authors used the same tree edits as this work, however like O'Reilly, they also used random edits to ensure that the created trees were syntactically correct.

The majority of the previous work utilising local search in program synthesis used the created techniques as part of a wider strategy. Typically local search was used to augment a genetic programming algorithm, in essence creating a memetic algorithm. For example, some authors [92] used local search as an additional genetic operator, while others [178] used it as a step in an overarching program synthesis algorithm. In our work, the local search technique was used on its own, and not as part of a wider strategy. However, we believe that LSPS is more generalised than the other local search techniques that have been used, and could be easily deployed on many other problem domains either on its own, or as part of a memetic algorithm.

8.2 Summary

8.2.1 Chapter 3

Chapter 3 detailed the way that we represent and evaluate heuristics in this thesis. It provided the necessary material for understanding the heuristics automatically created in later chapters, as well as offering a point of reference for how previously described heuristics perform on the fitness function and testing set. The DSL described can be considered an extension to those previously used for creating LS-SAT heuristics [147, 60, 63, 61]. The mechanisms and algorithms used to evaluate the heuristics may be of interest to the research community, as they are designed with efficiency in mind.

Though not the overarching goal of this thesis, we believe that the generalised mechanism for updating arbitrary gain type metrics may be of particular interest to the LS-SAT community. While efficient, the update mechanisms in the literature [15] are only described in terms of a small number of gain type metrics, whereas the update mechanism we presented is described efficiently for all gain type metrics considered in this thesis.

8.2.2 Chapter 4

In Chapter 4 we performed a series of program synthesis experiments to create LS-SAT heuristics using two languages. The second of these languages can be considered an extended version of the first. The two program synthesis methods used were exhaustive enumeration and genetic programming. The genetic programming experiments provided further evidence that it is a viable program synthesis method

for creating LS-SAT heuristics. The exhaustive enumeration experiments, while computationally expensive, provided us with data about tens of millions of heuristics. From these results, we found evidence of the clustering of heuristics in the search space. The heuristics appeared to be clustered in groups based on the order they were generated in by the exhaustive enumeration algorithm. Through our understanding of this ordering, we were able to ascertain that this clustering was categorised by the leading terms in the heuristics. We also described several examples of heuristics which appeared to outperform state-of-the-art hand-crafted heuristics on the testing set of problem instances described in Chapter 3.

The examples of clustering described in the exhaustive enumeration experiments directed our research in the subsequent chapters. In the conclusions to this chapter we reasoned that, if an algorithm were able to locate and navigate these clusters, then such an algorithm may be able to find effective heuristics far more quickly than exhaustive enumeration.

8.2.3 Chapter 5 - 7

The research described in Chapters 5 to 7 is difficult to compartmentalise, as each chapter is effectively part of a larger body of work. In effect, these three chapters describe the motivations behind, the description of, and the evaluation of LSPS. Because of this, we summarise these chapters together.

In Chapter 5 we presented empirical evidence which showed that the minimum tree edit distance metric could be used to describe the neighbourhood of a heuristic. This metric was used as it could capture the notion of clustering discussed in the previous chapter, as well as other examples of similarity observed. Through further experimentation, we found that many of these neighbourhoods contained heuristics with a higher fitness. Using the results from the exhaustive enumeration experiments described in Chapter 4, we performed a simulated local search using these neighbourhoods. In the results from these simulations, we found that this technique could consistently produce high-quality heuristics.

However, the simulated experiments relied on the use of memoized data. To apply this program synthesis technique to any candidate heuristic, we would need an algorithm that could construct the neighbourhood using only a candidate heuristic, and not rely on memoized data.

Chapter 6 described the algorithm we developed to create neighbourhoods based

on the minimum tree edit distance metric. We showed how, through the recognition of patterns in a program tree, the algorithm is able to apply pre-computed edit sequences to an input program tree to create output program trees. These output program trees are guaranteed to be syntactically correct. We then described how the pre-computed edit sequences are constructed. Throughout this chapter, we gave the descriptions of the various algorithms in terms of generic program trees and languages, rather than any of the heuristic representations used in this thesis. The algorithms were described in this manner so that they can be easily used in other program synthesis problems, not just those considered in this thesis.

In Chapter 7 we used the neighbourhood generation algorithms to perform a set of LSPS experiments. These experiments were designed to ascertain how effective LSPS is as a program synthesis technique. In total we performed eight experiments, using different languages, neighbourhood definitions, neighbourhood generation algorithms, initialisation functions and bounds on the size of the created heuristics. The results from these experiments provided strong evidence that LSPS is a viable program synthesis technique. We also showed several examples of highly effective, automatically created heuristics. One of these heuristics had the highest reported fitness of all those considered in this thesis, while another reported the best overall performance on the testing set.

8.3 Extensions & Future Work

In this section we suggest several potential avenues for future research.

8.3.1 Improved Fitness Function

The results in Chapters 4 and 7 showed us that many of the heuristics which reported a high fitness value were not effective at solving the larger problem instances in the testing set. As these heuristics were not trained on problem instances of this size, this is not surprising, however we had hoped that a heuristic's fitness value would provide some indication as to how effective it is on larger instances. While some heuristics which reported high fitness values did perform well on the larger instances, others did not. In essence, we believe the fitness function was ill-suited for accurately identifying heuristics that are effective across a range of different sized problem instances.

Other researchers have expressed a similar desire for automatically created heuris-

tics that are effective across a range of different sized problem instances [138]. In a similar way, within the LS-SAT community some researchers have described heuristics that appear to be effective on smaller problem instances, however on larger ones their performance degrades significantly [160, 161].

We believe that the systems we have developed and the data gathered through exhaustive enumeration could prove invaluable in identifying mechanisms for designing robust, accurate and computationally inexpensive fitness functions which can identify LS-SAT heuristics that are effective on a range of different sized problem instances. Such a fitness function could be used to design effective LS-SAT heuristics using the program synthesis systems described in this thesis, and could prove useful when designing fitness functions for other domains.

8.3.2 Language

In Chapter 3 we provided details regarding the DSL used throughout this thesis. In subsequent chapters we used parts of it to design languages, which were then used to automatically create LS-SAT heuristics. While we believe that the DSL we created is more verbose than those designed by other researchers, there are still several ways that it could be improved.

When designing Language B in Chapter 6, we chose not to include several of the terms we had used in Language A. This was due to our belief that doing so may create either ineffective or overfitted heuristics. We believe that these issues could be alleviated by redesigning some of the components. For example, the `IfNotMinAge` function could be re-formulated to remove its `VarSet` argument, instead being redesigned so that it can automatically determine what its `VarSet` argument should be based on the source of its v_1 argument. We believe this change would allow it to always have semantic meaning, irrespective of the contents of its child nodes.

Another example of a term that could be redesigned is the `IfTabu` function - or more specifically, the types of terminals with an `Age` type signature included in the language. Instead of including terms which represent constant Age values, we could instead include a set of terms which represent proportions of the overall Age of the local search algorithm, which may allow the heuristics to perform better on larger problem instances. In addition to this, we could experiment with adaptive Age variables, which could allow an Age value to adapt as the heuristic is running.

Apart from reformulating currently included terms, we could also introduce

mechanisms from other LS-SAT heuristics. For example, terms which represent other types of clause weighting, variable weighting (see Section 2.3.4) or different types of configuration checking (see Section 2.3.6) could be easily added to the language. Doing so may allow the program synthesizers to create higher quality heuristics.

8.3.3 Improving the LS-SAT Solver

We have used the LS-SAT solver described in Chapter 3 throughout this thesis and, while it has proven to be capable of evaluating complicated heuristics on large SAT problems, there are some improvements which could be made to its design.

One improvement that could be made is in how the solver evaluates heuristics. Currently it uses a bespoke software solution to do this. As discussed in Section 3.3, we do not think it would be practical to use a language such as C to compile and evaluate automatically created heuristics. However, recent advances in compiler technologies mean that it may be possible to create a software solution which is much more efficient than that currently used. For example, using a toolchain such as LLVM [105] could allow us to compile the heuristic function into low-level machine code. We could then call this function as part of an overarching LS-SAT solver. We believe a system designed in this manner would allow us to evaluate heuristics much more efficiently than is currently possible.

Another improvement that could be made is in how the LS-SAT solver updates the data structures required by a heuristic. Currently the system decides how to update these structures using rules based on an analysis of the heuristic, taking into account which combinations of gain types, clause weights and variable sets are used together. However, it doesn't take into consideration how the semantic meaning of a heuristic may cause a different update function to be more efficient. For example, the root of a heuristic may be in the form `IfRandLt` {0.1, `GetBestVar` {`WFF`, `NegGain`}, *b*}. If no subtree in *b* requires an ordering of all variables in the problem according to the $\textsc{NegGain}_1$, it may be more efficient to only calculate this ordering when it is required by the subtree `GetBestVar` {`WFF`, `NegGain`}. In adding this functionality, we may be able to evaluate automatically created heuristics more efficiently than we currently can. Such a system could also prove to be useful when designing hand-crafted heuristics, as it could be used to automatically identify optimisations in a heuristic's update function. We have seen little work in the literature automating this part of the heuristic design process, however we know that the efficiency of a

heuristic plays a large role in how effective that heuristic is when solving problem instances. Therefore, formulating strategies to automate this part of the heuristic design process could allow higher quality heuristics to be created.

One final improvement that could be made to the LS-SAT solver is regarding the types of problems it is designed to work with. Currently it is designed for use with SAT, but we believe that it would be relatively easy to extend it to work with MAX-SAT and Weighted MAX-SAT. Through this functionality, we may be able to automatically create effective heuristics for other domains.

8.3.4 Using Other Program Synthesis Methods

One of the research aims of this thesis was to identify alternate program synthesis techniques that could be used to automatically create heuristics. While we believe we have achieved this goal, through this work we have also identified other techniques that may be applicable to our domain.

In Section 2.5.3 we discussed several techniques which view program synthesis as a search problem. GP and exhaustive enumeration are two such techniques; GP works by sampling the leaves in a search tree, whereas exhaustive enumeration imposes an ordering on the leaves, then iterates through them using this ordering. We also considered algorithms, such as Top-Down-Search (see Algorithm 2.17) and bi-directional search, which operate on nodes in the search tree. To be clear, these nodes are partially constructed program trees or, in our domain, partially constructed heuristics. Strategies such as pruning have been used to remove branches of the search tree using these algorithms. However, as stated in Section 2.5.5, pruning is difficult to use in our domain. It would require being able to automatically reason about stochastic programs, which would require extensive expert knowledge.

There are other search techniques that could be used to reason about partially constructed heuristics. One example is Monte Carlo tree search. By sampling the fitness of the leaves descended from a particular node n, a numerical value could be calculated to represent the effectiveness of the heuristics underneath n. By applying this strategy to multiple nodes, an ordering could be constructed. An overarching algorithm could then use this ordering to direct the search to areas where it believes effective heuristics reside. We note that the results in Section 4.2.3, which showed that the leading terms in a heuristic could be used to ascertain that heuristic's effectiveness, suggest that this strategy may be a viable alternative to the other

methods we have considered.

As a technique to create heuristics, we believe that LSPS is worthy of further research. One potential extension to the LSPS algorithm could be to use some strategy to order a heuristic's neighbours, instead of doing so randomly. For example, a numerical value could be assigned to a heuristic through some computationally inexpensive analysis. These values could be used to order a neighbourhood and direct the search - in essence we would consider such a strategy to be analogous to utilising a heuristic method on the search space of heuristics. By leveraging this with a technique such as Monte Carlo tree search, we may be able to design algorithms that can quickly navigate the search space of heuristics in a controlled manner, and refine any promising heuristics found. Such algorithms could provide an alternative to those methods currently used, which, as we have discussed, are predominantly based on evolutionary computation.

The research we have conducted in this thesis could provide a major advantage when designing such algorithms. The heuristics gathered from the exhaustive enumeration experiments described in Chapter 4 could be used to perform computationally inexpensive simulations of any new algorithms, in a similar way to how the simulations described in Chapter 5 were used to ascertain the effectiveness of LSPS.

8.4 Final Remarks

To summarise, there are three core achievements of this thesis. Firstly, and perhaps most importantly, we have presented a new, novel, generalised method of program synthesis called local search program synthesis. Secondly, we have illustrated how two previously unused program synthesis methods, exhaustive enumeration and the aforementioned local search program synthesis, are able to create effective heuristics for our target domain. Finally, we have created effective, previously unknown LS-SAT heuristics, some of which outperform state-of-the-art, hand-crafted heuristics. These three core achievements mirror the three subdisciplines that intersect within the contents of this thesis; that of program synthesis, automated heuristic creation and SAT.

Bibliography

[1] Takuya Akiba et al. *Calibrating research in program synthesis using 72,000 hours of programmer time*. Tech. rep. MSR, 2013.

[2] Tatsuya Akutsu. "Tree edit distance problems: Algorithms and applications to bioinformatics". In: *IEICE Transactions on Information Systems* 93-D.2 (2010), pp. 208–218.

[3] Sam D. Allen et al. "Evolving reusable 3D packing heuristics with genetic programming". In: *Proceedings of the 11th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'09)*. Montreal, Canada, July 2009, pp. 931–938.

[4] Rajeev Alur et al. "Syntax-guided synthesis". In: *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design, (FMCAD '14)*. Portland, USA, Oct. 2013, pp. 1–8.

[5] Anbulagan et al. "Old resolution meets modern SLS". In: *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'05)*. Pittsburgh, USA, July 2005, pp. 354–359.

[6] Gilles Audemard and Laurent Simon. "GUNSAT: A greedy local search algorithm for unsatisfiability". In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence, (IJCAI 2007)*. Hyderabad, India, Jan. 2007, pp. 2256–2261.

[7] Nikolaus Augsten et al. "Efficient top-k approximate subtree matching in small memory". In: *IEEE Transactions on Knowledge and Data Engineering* 23.8 (2011), pp. 1123–1137.

[8] Raja Muhammad Atif Azad and Conor Ryan. “A simple approach to lifetime learning in genetic programming-based symbolic regression”. In: *Evolutionary computation* 22.2 (2014), pp. 287–317.

[9] Mohamed Bahy Bader-El-Den and Riccardo Poli. “Generating SAT local-search heuristics using a GP hyper-heuristic framework”. In: *Proceedings of the Evolution Artificielle, 8th International Conference on Artificial Evolution (EA’07)*. Tours, France, Oct. 2007, pp. 37–49.

[10] Adrian Balint and Andreas Fröhlich. “Improving stochastic local search for SAT with a new probability distribution”. In: *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2010)*. Edinburgh, UK, July 2010, pp. 10–15.

[11] Adrian Balint and Norbert Manthey. “SparrowToRiss 2018”. In: *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*. Oxford, UK, July 2018, pp. 38–39.

[12] Adrian Balint and Uwe Schöning. “Choosing probability distributions for stochastic local search and the role of make versus break”. In: *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2012)*. Trento, Italy, June 2012, pp. 16–29.

[13] Adrian Balint and Uwe Schöning. “probSAT”. In: *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*. Helsinki, Finland, July 2013, p. 70.

[14] Adrian Balint and Uwe Schöning. “probSAT and pprobSAT”. In: *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Vienna, Austria, July 2014, p. 63.

[15] Adrian Balint et al. “Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses”. In: *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2014)*. Vienna, Austria, July 2014, pp. 302–316.

[16] Márcio Porto Basgalupp, Rodrigo Coelho Barros, and Tiago Barabasz. “A grammatical evolution based hyper-heuristic for the automatic design of split criteria”. In: *Proceedings of the 2014 ACM Annual Genetic and Evolutionary Computation Conference, (GECCO’14)*. Vancouver, Canada, July 2014, pp. 1311–1318.

[17] Daniel Le Berre and Laurent Simon. "Preface to the Special Volume on the SAT 2005 Competitions and Evaluations". In: *Journal on Satisfiability, Boolean Modeling and Computation* 2.1-4 (2006).

[18] A. Biere and M. Heule. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-586-03929-5.

[19] Philip Bille. "A survey on tree edit distance and related problems". In: *Theoretical Computer Science* 337.1-3 (2005), pp. 217–239.

[20] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. "Program verification as Satisfiability modulo theories". In: *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT '12)*. Manchester, UK, June 2012, pp. 3–11.

[21] M.F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer US, 2007. ISBN: 978-0-387-31030-5.

[22] Forrest Briggs and Melissa O'Neill. "Functional genetic programming and exhaustive program search with combinator expressions". In: *International Journal of Knowledge-Based and Intelligent Engineering Systems* 12.1 (2008), pp. 47–68.

[23] E.K. Burke and G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer US, 2006. ISBN: 978-0-387-28356-2.

[24] Edmund K Burke et al. "A graph-based hyper-heuristic for educational timetabling problems". In: *European Journal of Operational Research* 176.1 (2007), pp. 177–192.

[25] Edmund K Burke et al. "Automatic heuristic generation with genetic programming: Evolving a jack-of-all-trades or a master of one". In: *Proceedings of the 9th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'07)*. London, UK, July 2007, pp. 1559–1565.

[26] Edmund K Burke et al. "Exploring hyper-heuristic methodologies with genetic programming". In: *Computational Intelligence* (2009), pp. 177–201.

[27] Edmund K. Burke, Matthew R. Hyde, and Graham Kendall. "Evolving bin packing heuristics with genetic programming". In: *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN 2006)*. Reykjavik, Iceland, Sept. 2006, pp. 860–869.

[28] Edmund K. Burke et al. "Hyper-heuristics: A survey of the state of the art". In: *Journal of the Operations Research Society* 64.12 (2013), pp. 1695–1724.

[29] Edmund K. Burke et al. "Hyper-Heuristics: An emerging direction in modern search technology". In: *Handbook of Metaheuristics*. Ed. by Fred W. Glover and Gary A. Kochenberger. Vol. 57. International Series in Operations Research & Management Science. Kluwer / Springer, 2003.

[30] Andrew Burnett and Andrew Parkes. "Systematic search for local-search SAT heuristics". In: *Proceedings of the 6th International Conference on Metaheuristics and Nature Inspired Computing, (META '16)*. Marrakech, Morocco, June 2016, pp. 268–270.

[31] Shaowei Cai. *Faster implementation for walksat*. Tech. rep. Queensland Research Lab, NICTA, Australia, 2013.

[32] Shaowei Cai, Chuan Luo, and Kaile Su. "CCASat: Solver description". In: *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*. Trento, Italy, July 2013, pp. 13–14.

[33] Shaowei Cai, Chuan Luo, and Kaile Su. "CCgscore in SAT Competition 2014". In: *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Vienna, Austria, July 2014, pp. 19–20.

[34] Shaowei Cai, Chuan Luo, and Haochen Zhang. "From decimation to local search and back: A new approach to MaxSAT." In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence, (IJCAI 2017)*. Melbourne, Australia, Aug. 2017, pp. 571–577.

[35] Shaowei Cai and Kaile Su. "Comprehensive score: Towards efficient local search for SAT with long clauses". In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*. Beijing, China, Aug. 2013, pp. 489–495.

[36] Shaowei Cai and Kaile Su. "Configuration checking with aspiration in local search for SAT". In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. Toronto, Canada, July 2012, pp. 434–440.

[37] Shaowei Cai and Kaile Su. “Local search for Boolean Satisfiability with configuration checking and subscore”. In: *Artificial Intelligence* 204 (2013), pp. 75–98.

[38] Shaowei Cai and Kaile Su. “Local search with configuration checking for SAT”. In: *IEEE 23rd International Conference on Tools with Artificial Intelligence, (ICTAI 2011)*. Boca Raton, USA, Nov. 2011, pp. 59–66.

[39] Shaowei Cai, Kaile Su, and Abdul Sattar. “Local search with edge weighting and configuration checking heuristics for minimum vertex cover”. In: *Artificial Intelligence* 175.9-10 (2011), pp. 1672–1696.

[40] Shelvin Chand et al. “On the use of genetic programming to evolve priority rules for resource constrained project scheduling problems”. In: *Information Sciences* 432 (2018), pp. 146–163.

[41] Salman Cheema et al. “A practical framework for constructing structured drawings”. In: *Proceedings of the 19th International Conference on Intelligent User Interfaces, (IUI 2014)*. Haifa, Israel, Feb. 2014, pp. 311–316.

[42] Peter C. Cheeseman, Bob Kanefsky, and William M. Taylor. “Where the really hard problems are”. In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence, (IJCAI ’91)*. Sudney, Australia, Aug. 1991, pp. 331–340.

[43] Jingchao Chen. “Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques”. In: *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2009)*. Swansea, UK, June 2009, pp. 298–311.

[44] Edmund M. Clarke et al. “Bounded model checking using Satisfiability solving”. In: *Formal Methods in System Design* 19.1 (2001), pp. 7–34.

[45] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. Shaker Heights, USA, May 1971, pp. 151–158.

[46] Peter I. Cowling, Graham Kendall, and Eric Soubeiga. “A hyperheuristic approach to scheduling a sales summit”. In: *Proceedings of the 3rd International Conference on Practice and Theory of Automated Timetabling, (PATAT 2000)*. Konstanz, Germany, Aug. 2000, pp. 176–190.

[47] Wallace B Crowston, Fred Glover, Jack D Trawick, et al. *Probabilistic and parametric learning combinations of local job shop scheduling rules*. Tech. rep. Cambridge Institution of Technology, 1963.

[48] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. "Qlose: Program repair with quantitative objectives". In: *Proceedings of the 28th International Conference on Computer Aided Verification, (CAV 2016)*. Toronto, Canada, July 2016, pp. 383–401.

[49] Martin Davis, George Logemann, and Donald W. Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[50] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *Journal of the ACM* 7.3 (1960), pp. 201–215.

[51] Erik D. Demaine et al. "An optimal decomposition algorithm for tree edit distance". In: *ACM Transactions on Algorithms* 6.1 (2009), 2:1–2:19.

[52] John H. Drake et al. "A genetic programming hyper-heuristic for the multidimensional knapsack problem". In: *Kybernetes* 43.9/10 (2014), pp. 1500–1511.

[53] Iztok Fajfar, Árpád Bürmen, and Janez Puhan. "Grammatical evolution as a hyper-heuristic to evolve deterministic real-valued optimization algorithms". In: *Genetic Programming and Evolvable Machines* 19.4 (2018), pp. 473–504.

[54] Lei Fang and Michael S. Hsiao. "A new hybrid solution to boost SAT solver performance". In: *2007 Design, Automation and Test in Europe Conference and Exposition*. Nice, France, Apr. 2007, pp. 1307–1313.

[55] Cândida Ferreira. "Gene expression programming: A new adaptive algorithm for solving problems". In: *Complex Systems* 13.2 (2001), pp. 87–129.

[56] Henry Fisher. "Probabilistic learning combinations of local job-shop scheduling rules". In: *Industrial Scheduling* (1963), pp. 225–251.

[57] Peter A Flach. "The logic of learning: A brief introduction to inductive logic programming". In: *Proceedings of the CompulogNet Area Meeting on Computational Logic and Machine Learning*. Manchester, UK, June 1998, pp. 1–17.

[58] Pierre Flener. "Inductive logic program synthesis with DIALOGS". In: *Inductive Logic Programming, 6th International Workshop, ILP-96, Selected Papers*. Stockholm, Sweden, Aug. 1996, pp. 175–198.

[59] Pierre Flener and Ute Schmid. "An introduction to inductive programming". In: *Artificial Intelligence Review* 29.1 (2008), pp. 45–62.

[60] Alex S. Fukunaga. "Automated discovery of composite SAT variable-selection heuristics". In: *Proceedings of the 18th National Conference on Artificial Intelligence and the 14th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'02)*. Edmonton, Canada, Aug. 2002, pp. 641–648.

[61] Alex S. Fukunaga. "Automated discovery of local search heuristics for Satisfiability testing". In: *Evolutionary Computation* 16.1 (2008), pp. 31–61.

[62] Alex S. Fukunaga. "Efficient implementations of SAT local search". In: *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2004)*. Vancouver, Canada, May 2004.

[63] Alex S. Fukunaga. "Evolving local search heuristics for SAT using genetic programming". In: *Proceedings of the 6th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'04)*. Seattle, USA, June 2004, pp. 483–494.

[64] Oliver Gableske. "Dimetheus". In: *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*. Bordeaux, France, July 2016, pp. 37–38.

[65] M. Gendreau and J.Y. Potvin. *Handbook of Metaheuristics*. International Series in Operations Research & Management Science. Springer International Publishing, 2018. ISBN: 978-3-319-91086-4.

[66] Ian Gent and Toby Walsh. *The enigma of SAT hill-climbing procedures*. Tech. rep. 605. Department of AI, University of Edinburgh, 1992.

[67] Ian P. Gent and Toby Walsh. "The SAT phase transition". In: *Proceedings of the Eleventh European Conference on Artificial Intelligence*. Amsterdam, the Netherlands, Aug. 1994, pp. 105–109.

[68] Ian P. Gent and Toby Walsh. "Towards an understanding of hill-climbing procedures for SAT". In: *Proceedings of the 11th National Conference on Artificial Intelligence and the 5th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'07)*. Washington, USA, July 1993, pp. 28–33.

[69] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. "Answer set programming based on propositional Satisfiability". In: *Journal of Automated Reasoning* 36.4 (2006), pp. 345–377.

[70] Fred W. Glover. "Tabu search - part I". In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206.

[71] Mario Graff, Rafael Peña, and Aurelio Medina. "Wind speed forecasting using genetic programming". In: *Proceedings of the IEEE Congress on Evolutionary Computation, (CEC 2013)*. Cancun, Mexico, June 2013, pp. 408–415.

[72] Jun Gu. "Efficient local search for very large-scale Satisfiability problems". In: *SIGART Bulletin* 3 (1992), pp. 8–12.

[73] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2011)*. Austin, USA, Jan. 2011, pp. 317–330.

[74] Sumit Gulwani. "Dimensions in program synthesis". In: *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. Hagenberg, Austria, July 2010, pp. 13–24.

[75] Sumit Gulwani, William R. Harris, and Rishabh Singh. "Spreadsheet data manipulation using examples". In: *Communications of the ACM* 55.8 (2012), pp. 97–105.

[76] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program synthesis". In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119.

[77] Sumit Gulwani et al. "Inductive programming meets the real world". In: *Communications of the ACM* 58.11 (2015), pp. 90–99.

[78] Djamal Habet, Donia Toumi, and André Abrame. "Ncca+: Configuration checking and Novelty+ like heuristic". In: *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*. Helsinki, Finland, July 2013, pp. 61–62.

[79] Brad Harvey, James A. Foster, and Deborah A. Frincke. “Towards byte code genetic programming”. In: *Proceedings of the 1st ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'99).* Orlando, USA, July 1999, p. 1234.

[80] Timon Hertli. “3-SAT faster and simpler - unique-SAT bounds for PPSZ hold in general”. In: *SIAM Journal on Computing* 43.2 (2014), pp. 718–729.

[81] Edward A Hirsch and Arist Kojevnikov. “UnitWalk: A new SAT solver that uses local search guided by unit clause elimination”. In: *Annals of Mathematics and Artificial Intelligence* 43.1 (2005), pp. 91–111.

[82] Nhu Binh Ho and Joc Cing Tay. “Evolving dispatching rules for solving the flexible job-shop problem”. In: *Proceedings of the 2005 IEEE Congress on Evolutionary Computation, (CEC 2005).* Edinburgh, UK, Sept. 2005, pp. 2848–2855.

[83] H.H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications.* The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2005. ISBN: 978-1-558-60872-6.

[84] Holger H. Hoos. “An adaptive noise mechanism for WalkSAT”. In: *Proceedings of the 18th National Conference on Artificial Intelligence and the 14th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'02).* Edmonton, Canada, July 2002, pp. 655–660.

[85] Holger H. Hoos. “On the run-time behaviour of stochastic local search algorithms for SAT”. In: *Proceedings of the 16th National Conference on Artificial Intelligence and the 11th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'99).* Orlando, USA, July 1999, pp. 661–666.

[86] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. “Scaling and probabilistic smoothing: Efficient dynamic local search for SAT”. In: *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, (CP 2002).* Ithaca, USA, Sept. 2002, pp. 233–248.

[87] Abdelraouf Ishtaiwi et al. “Neighbourhood clause weight redistribution in local search for SAT”. In: *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, (CP 2005).* Sitges, Spain, Oct. 2005, pp. 772–776.

[88] Domagoj Jakobovic, Leonardo Jelenkovic, and Leo Budin. “Genetic programming heuristics for multiple machine scheduling”. In: *Proceedings of the 10th European Conference on Genetic Programming, (EUROGP 2007)*. Valencia, Spain, Apr. 2007, pp. 321–330.

[89] Mikolás Janota and Inês Lynce, eds. *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing, (SAT 2019)*. Lisbon, Portugal, July 2019.

[90] S.L.P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice/Hill International, 1987. ISBN: 978-0-134-53333-9.

[91] Terry Jones and Stephanie Forrest. “Fitness distance correlation as a measure of problem difficulty for genetic algorithms”. In: *Proceedings of the 6th International Conference on Genetic Algorithms*. Pittsburgh, USA, July 1995, pp. 184–192.

[92] Perla S. Juárez-Smith, Mauro Castelli, and Emigdio Z.-Flores. “Local search is underused in genetic programming”. In: *Genetic Programming Theory and Practice XIV*. Ann Arbor, USA, May 2016, pp. 119–137.

[93] Richard M. Karp. “Reducibility among combinatorial problems”. In: *Proceedings of a Symposium on the Complexity of Computer Computations*. New York, USA, Mar. 1972, pp. 85–103.

[94] Susumu Katayama. “An analytical inductive functional programming system that avoids unintended programs”. In: *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, (PEPM 2012)*. Philadelphia, USA, Jan. 2012, pp. 43–52.

[95] Susumu Katayama. “Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening”. In: *Proceedings of the 10th Pacific Rim International Conference on Artificial Intelligence: Trends in Artificial Intelligence, (PRICAI 2008)*. Hanoi, Vietnam, Dec. 2008, pp. 199–210.

[96] Susumu Katayama. “Power of brute-force search in strongly-typed inductive functional programming automation”. In: *Proceedings of the 8th Pacific Rim International Conference on Artificial Intelligence: Trends in Artificial Intelligence, (PRICAI 2004)*. Auckland, New Zealand, Aug. 2004, pp. 75–84.

[97] Henry A. Kautz and Bart Selman. "Planning as Satisfiability". In: *Proceedings of the 10th European conference on Artificial intelligence, (ECAI '92)*. Vienna, Austria, Aug. 1992, pp. 359–363.

[98] Robert E. Keller and Riccardo Poli. "Linear genetic programming of parsimonious metaheuristics". In: *Proceedings of the 2007 IEEE Congress on Evolutionary Computation, (CEC 2007)*. Singapore, Sept. 2007, pp. 4508–4515.

[99] Ashiqur R. KhudaBukhsh et al. "SATenstein: Automatically building local search SAT solvers from components". In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence, (IJCAI 2009)*. Pasadena, USA, July 2009, pp. 517–524.

[100] Ashiqur R. KhudaBukhsh et al. "SATenstein: Automatically building local search SAT solvers from components". In: *Artificial Intelligence* 232 (2016), pp. 20–42.

[101] Emmanuel Kieffer et al. "Tackling large-scale and combinatorial bi-level problems with a genetic programming hyper-heuristic". In: *IEEE Transactions on Evolutionary Computation* 24.1 (2020), pp. 44–56.

[102] J.R. Koza, J.R. Koza, and J.P. Rice. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book. Bradford, 1992. ISBN: 978-0-262-11170-6.

[103] John R. Koza. "Human-competitive results produced by genetic programming". In: *Genetic Programming and Evolvable Machines* 11.3-4 (2010), pp. 251–284.

[104] Tracy Larrabee. "Test pattern generation using Boolean Satisfiability". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.1 (1992), pp. 4–15.

[105] Chris Lattner and Vikram S. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization, (CGO 2004)*. San Jose, USA, Mar. 2004, pp. 75–88.

[106] Claire Le Goues et al. "Genprog: A generic method for automatic software repair". In: *IEEE Transactions on Software Engineering* 38.1 (2011), pp. 54–72.

[107] Chu Min Li and Wenqi Huang. "Diversification and determinism in local search for Satisfiability". In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2005)*. St. Andrews, UK, June 2005, pp. 158–172.

[108] Chu Min Li, Wanxia Wei, and Harry Zhang. "Combining adaptive noise and look-ahead in local search for SAT". In: *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2007)*. Lisbon, Portugal, May 2007, pp. 121–133.

[109] Chumin Li, Chong Huang, and Xu Ruchu. "Balance between intensification and diversification: a unity of opposites". In: *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Vienna, Austria, July 2014, pp. 10–11.

[110] Jian Lin, Lei Zhu, and Kaizhou Gao. "A genetic programming hyper-heuristic approach for the multi-skill resource constrained project scheduling problem". In: *Expert Systems with Applications* 140 (2020). Article no. 112915.

[111] Shen Lin and Brian W. Kernighan. "An effective heuristic algorithm for the Traveling-Salesman problem". In: *Operations Research* 21.2 (1973), pp. 498–516.

[112] Fan Long and Martin Rinard. "Automatic patch generation by learning correct code". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2016)*. St. Petersburg, USA, Jan. 2016, pp. 298–312.

[113] Jana Lovíšková. "Solving the 3-SAT problem using genetic algorithms". In: *2015 IEEE 19th International Conference on Intelligent Engineering Systems (INES)*. Bratislava, Slovakia, Sept. 2015, pp. 207–212.

[114] S. Y. Lu. "A tree-matching algorithm based on node splitting and merging". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6.2 (1984), pp. 249–256.

[115] Chuan Luo, Kaile Su, and Shaowei Cai. "Improving local search for random 3-SAT using quantitative configuration checking". In: *Proceedings of the 20th European conference on Artificial intelligence, (ECAI 2012)*. Montpellier, France, Aug. 2012, pp. 570–575.

[116] Chuan Luo et al. “Clause states based configuration checking in local search for Satisfiability”. In: *IEEE Transactions on Cybernetics* 45.5 (2014), pp. 1028–1041.

[117] Chuan Luo et al. “CSCCSat in SAT Competition 2016”. In: *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions.* Bordeaux, France, July 2016, pp. 37–38.

[118] Chuan Luo et al. “Double configuration checking in stochastic local search for Satisfiability”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence.* Québec City, Canada, July 2014, pp. 2703–2709.

[119] Chuan Luo et al. “Focused random walk with configuration checking and break minimum for Satisfiability”. In: *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, (CP 2013).* Uppsala, Sweden, Sept. 2013, pp. 481–496.

[120] Zohar Manna and Richard J Waldinger. “Toward automatic program synthesis”. In: *Communications of the ACM* 14.3 (1971), pp. 151–165.

[121] Zohar Manna and Richard J. Waldinger. “A deductive approach to program synthesis”. In: *ACM Transactions on Programming Languages and Systems* 2.1 (1980), pp. 90–121.

[122] Joao Marques-Silva. “Practical applications of Boolean Satisfiability”. In: *Proceedings of the 9th International Workshop on Discrete Event Systems, (WODES 2008).* Gothenburg, Sweden, May 2008, pp. 74–80.

[123] David A. McAllester, Bart Selman, and Henry A. Kautz. “Evidence for invariants in local search”. In: *Proceedings of the 14th National Conference on Artificial Intelligence and the 9th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'07).* Providence, USA, July 1997, pp. 321–326.

[124] James McDermott et al. “Genetic programming needs better benchmarks”. In: *Proceedings of the 14th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'12).* Philadelphia, USA, July 2012, pp. 791–798.

[125] Robert I McKay et al. “Grammar-based genetic programming: A survey”. In: *Genetic Programming and Evolvable Machines* 11.3 (2010), pp. 365–396.

[126] Aditya Krishna Menon et al. "A machine learning framework for programming by example". In: *Proceedings of the 30th International Conference on Machine Learning, (ICML 2013)*. Atlanta, USA, June 2013, pp. 187–195.

[127] W. Michiels, E. Aarts, and J. Korst. *Theoretical Aspects of Local Search*. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2007. ISBN: 978-3-540-35854-1.

[128] Julian Francis Miller and Simon Harding. "Cartesian genetic programming". In: *Proceedings of the 10th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'08)*. Atlanta, USA, July 2008, pp. 2701–2726.

[129] David G. Mitchell, Bart Selman, and Hector J. Levesque. "Hard and easy distributions of SAT problems". In: *Proceedings of the 10th National Conference on Artificial Intelligence and the 4th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'92)*. San Jose, USA, July 1992, pp. 459–465.

[130] David J. Montana. "Strongly typed genetic programming". In: *Evolutionary Computation* 3.2 (1995), pp. 199–230.

[131] Paul Morris. "The breakout method for escaping from local minima". In: *Proceedings of the 11th National Conference on Artificial Intelligence and the 5th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'93)*. Washington, USA, July 1993, pp. 40–45.

[132] S. Muggleton. *Inductive Logic Programming*. A.P.I.C. studies in data processing. Academic Press, 1992. ISBN: 978-0-125-09715-4.

[133] Hoang Duong Thien Nguyen et al. "SemFix: Program repair via semantic analysis". In: *35th International Conference on Software Engineering, (ICSE '13)*. San Francisco, USA, May 2013, pp. 772–781.

[134] U-M O'Reilly. "Using a distance metric on genetic programs to understand genetic operators". In: *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*. Orlando, USA, Oct. 1997, pp. 4092–4097.

[135] Una-May O'Reilly. "An analysis of genetic programming." PhD thesis. Carleton University, 1995.

[136] Gabriela Ochoa, Rong Qu, and Edmund K. Burke. "Analyzing the landscape of a graph based hyper-heuristic for timetabling problems". In: *Proceedings of the 11th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'09)*. Montreal, Canada, July 2009, pp. 341–348.

[137] Fernando E. B. Otero, Tom Castle, and Colin G. Johnson. "EpochX: Genetic programming in Java with statistics and event monitoring". In: *Proceedings of the 14th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'12)*. Philadelphia, USA, July 2012, pp. 93–100.

[138] Ender Özcan and Andrew J. Parkes. "Policy matrix evolution for generation of heuristics". In: *Proceedings of the 13th ACM Annual Genetic and Evolutionary Computation Conference, (GECCO'11)*. Dublin, Ireland, July 2011, pp. 2011–2018.

[139] Benjamin Paaßen. *Revisiting the tree edit distance and its backtracing: A tutorial*. 2021. arXiv: 1805.06869 [cs.DS].

[140] Christos H. Papadimitriou. "On selecting a satisfying truth assignment (extended abstract)". In: *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*. San Juan, Puerto Rico, Oct. 1991, pp. 163–169.

[141] John Park et al. "An investigation of ensemble combination schemes for genetic programming based hyper-heuristic approaches to dynamic job shop scheduling". In: *Applied Soft Computing* 63 (2018), pp. 72–86.

[142] Mateusz Pawlik and Nikolaus Augsten. "RTED: A robust algorithm for the tree edit distance". In: *Proceedings of the VLDB Endowment* 5.4 (2011), pp. 334–345.

[143] Duc Nghia Pham and Charles Gretton. "gNovelty+". In: *SAT Competition 2007*. Lisbon, Portugal, May 2007.

[144] Duc Nghia Pham and Charles Gretton. "gNovelty+ (v. 2)". In: *SAT 2009 Competitive Events Booklet - Proceedings of the 2009 International SAT Competition, 12th International Conference on Theory and Applications of Satisfiability Testing*. Swansea, UK, June 2009, pp. 9–10.

[145] Duc Nghia Pham et al. "Combining adaptive and dynamic local search for Satisfiability". In: *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 149–172.

[146] Phitchaya Mangpo Phothilimthana et al. “Scaling up superoptimization”. In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS 2016)*. Atlanta, USA, Apr. 2016, pp. 297–310.

[147] R. Poli et al. *A Field Guide to Genetic Programming*. Lulu.com, 2008. ISBN: 978-1-409-20073-4.

[148] Riccardo Poli, John R. Woodward, and Edmund K. Burke. “A histogram-matching approach to the evolution of bin-packing strategies”. In: *Proceedings of the 2007 IEEE Congress on Evolutionary Computation, (CEC 2007)*. Singapore, Sept. 2007, pp. 3500–3507.

[149] Steven Prestwich. “Random walk with continuously smoothed variable weights”. In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, (SAT 2005)*. St. Andrews, UK, June 2005, pp. 203–215.

[150] M. Privosnik. “The scalability of evolved on line bin packing heuristics”. In: *Proceedings of the 2007 IEEE Congress on Evolutionary Computation, (CEC 2007)*. Singapore, Sept. 2007, pp. 2530–2537.

[151] *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*. Oxford, UK, July 2018.

[152] Conor Ryan, J. J. Collins, and Michael O’Neill. “Grammatical evolution: Evolving programs for an arbitrary language”. In: *Proceedings of the 1st European Workshop on Genetic Programming, (EUROGP ’98)*. Paris, France, Apr. 1998, pp. 83–96.

[153] Nasser R. Sabar et al. “Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems”. In: *IEEE Transactions on Evolutionary Computation* 19.3 (2015), pp. 309–325.

[154] Eric Schkufza, Rahul Sharma, and Alex Aiken. “Stochastic superoptimization”. In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS 2013*. Houston, USA, Mar. 2013, pp. 305–316.

[155] U. Schöning and J. Torán. *The Satisfiability Problem: Algorithms and Analyses*. Mathematik für Anwendungen. Lehmanns Media, 2013. ISBN: 978-3-865-41648-3.

[156] Dale Schuurmans and Finnegan Southey. "Local search characteristics of incomplete SAT procedures". In: *Artificial Intelligence* 132.2 (2001), pp. 121–150.

[157] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. "The exponentiated subgradient algorithm for heuristic Boolean programming". In: *Proceedings of the 17th International Joint Conference on Artificial Intelligence, (IJCAI 2001)*. Seattle, USA, Aug. 2001, pp. 334–341.

[158] Sakari Seitz, Mikko Alava, and Pekka Orponen. "Focused local search for random 3-Satisfiability". In: *Journal of Statistical Mechanics: Theory and Experiment* 2005.06 (2005). Article no. P06006.

[159] Bart Selman and Henry Kautz. "Domain-independent extensions to GSAT: Solving large structured Satisfiability problems". In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence, (IJCAI '93)*. Chambery, France, Aug. 1993, pp. 290–295.

[160] Bart Selman, Henry A Kautz, and Bram Cohen. "Noise strategies for improving local search". In: *Proceedings of the 12th National Conference on Artificial Intelligence and the 6th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'94)*. Seattle, USA, Aug. 1994, pp. 337–343.

[161] Bart Selman, Hector J. Levesque, and David G. Mitchell. "A new method for solving hard Satisfiability problems". In: *Proceedings of the 10th National Conference on Artificial Intelligence and the 4th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI'92)*. San jose, USA, July 1992, pp. 440–446.

[162] Alexander Semenov and Oleg Zaikin. "Algorithm for finding partitionings of hard variants of Boolean Satisfiability problem with application to inversion of some cryptographic functions". In: *SpringerPlus* 5.1 (2016). Article no. 554.

[163] Paul Shaw. "Using constraint programming and local search methods to solve vehicle routing problems". In: *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming, (CP 98)*. Pisa, Italy, Oct. 1998, pp. 417–431.

[164] Mary Sheeran and Gunnar Stålmarck. “A tutorial on Stålmarck’s proof procedure for propositional logic”. In: *Formal Methods System Design* 16.1 (2000), pp. 23–58.

[165] Alejandro Sosa-Ascencio et al. “Grammar-based generation of variable-selection heuristics for constraint satisfaction problems”. In: *Genetic Programming and Evolvable Machines* 17.2 (2016), pp. 119–144.

[166] William M Spears. “Simulated annealing for hard Satisfiability problems.” In: *Cliques, Coloring, and Satisfiability* 26 (1993), pp. 533–558.

[167] Phillip D. Summers. “A methodology for LISP program construction from examples”. In: *Journal of the ACM* 24.1 (1977), pp. 161–175.

[168] Kuo-Chung Tai. “The tree-to-tree correction problem”. In: *Journal of the ACM* 26.3 (1979), pp. 422–433.

[169] Naoyuki Tamura et al. “Compiling finite linear CSP into SAT”. In: *Constraints* 14.2 (2009), pp. 254–272.

[170] Boxiong Tan, Hui Ma, and Yi Mei. “A hybrid genetic programming hyper-heuristic approach for online two-level resource allocation in container-based clouds”. In: *Proceedings of the 2019 IEEE Congress on Evolutionary Computation, (CEC 2019)*. Wellington, New Zealand, June 2019, pp. 2681–2688.

[171] John Thornton et al. “Additive versus multiplicative clause weighting for SAT”. In: *Proceedings of the 19th National Conference on Artificial Intelligence and the 16th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI’04)*. San Jose, USA, July 2004, pp. 191–196.

[172] Leonardo Trujillo, Enrique Naredo, and Yuliana Martínez. “Preliminary study of bloat in genetic programming with behavior-based search”. In: *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation IV*. Springer, 2013, pp. 293–305.

[173] Zhe Wu and Benjamin W. Wah. “An efficient global-search strategy in discrete lagrangian methods for solving hard Satisfiability problems”. In: *Proceedings of the 17th National Conference on Artificial Intelligence and the 12th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI’00)*. Austin, USA, July 2000, pp. 310–315.

[174] Binzi Xu et al. "Genetic programming with delayed routing for multiobjective dynamic flexible job shop scheduling". In: *Evolutionary Computation* 29.1 (2021), pp. 75–105.

[175] Lin Xu et al. "SATzilla: Portfolio-based algorithm selection for SAT". In: *Journal of Artificial Intelligence Research* 32 (2008), pp. 565–606.

[176] Lin Xu et al. "SATzilla2009: An automatic algorithm portfolio for SAT". In: *SAT 2009 Competitive Events Booklet - Proceedings of the 2009 International SAT Competition, 12th International Conference on Theory and Applications of Satisfiability Testing*. Swansea, UK, June 2009, pp. 53–56.

[177] Tina Yu and Chris Clack. "PolyGP: A polymorphic genetic programming system in Haskell". In: *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Madison, USA, July 1998, pp. 416–421.

[178] Yuan Yuan and Wolfgang Banzhaf. "Iterative genetic improvement: Scaling stochastic program synthesis". In: *CoRR* abs/2202.13040 (2022).

[179] Guo-Qiang Zeng and Yong-Zai Lu. "Survey on computational complexity with phase transitions and extremal optimization". In: *Proceedings of the 48th IEEE Conference on Decision and Control, (CDC 2009)*. Shanghai, China, Dec. 2009, pp. 4352–4359.

[180] Hongyu Zhang et al. "Bing developer assistant: Improving developer productivity by recommending sample code". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE 2016)*. Seattle, USA, Nov. 2016, pp. 956–961.

[181] Kaizhong Zhang and Dennis E. Shasha. "Simple fast algorithms for the editing distance between trees and related problems". In: *SIAM Journal on Computing* 18.6 (1989), pp. 1245–1262.

[182] Mengjie Zhang and William D. Smart. "Genetic programming with gradient descent search for multiclass object classification". In: *Proceedings of the 7th European Conference on Genetic Programming, (EUROGP 2004)*. Coimbra, Portugal, Apr. 2004, pp. 399–408.

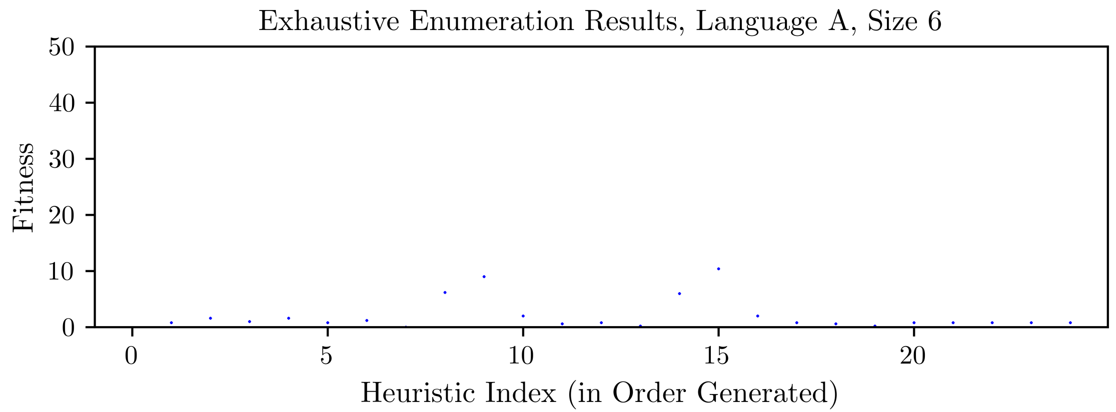
Appendices

Appendix A

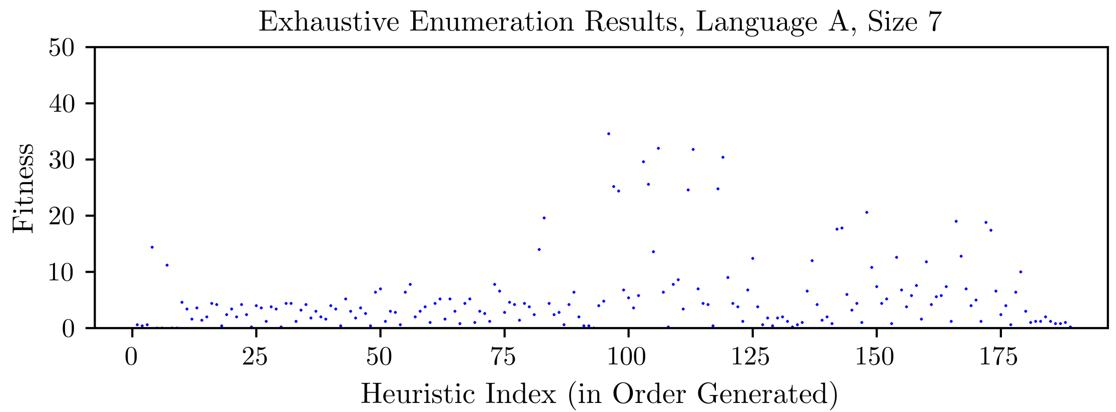
Exhaustive Enumeration Results

In this appendix we present all results from the exhaustive enumeration experiments using Language A and Language A1 in Chapter 4. We present these results as a series of graphs. The graphs contain all results for a specific size of heuristic for a specific language.

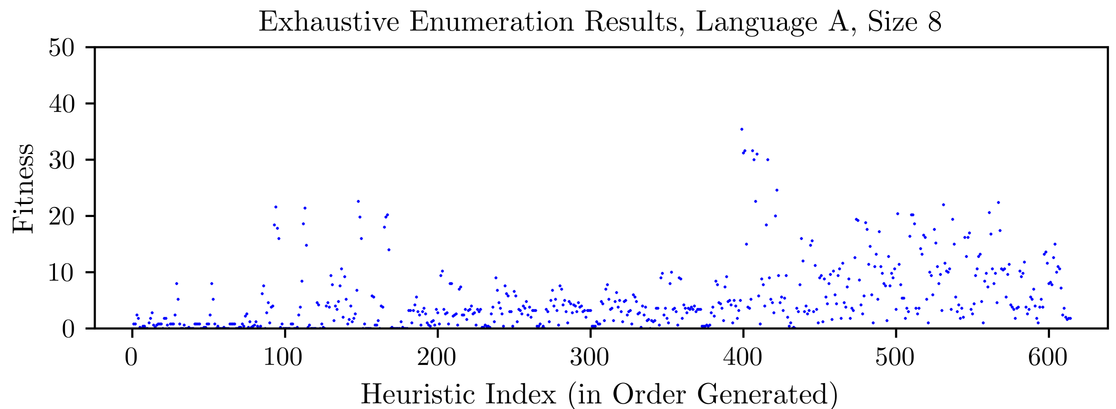
For Language A we show graphs for sizes 6-9 and 11-16 in Figure A.1. The graphs for sizes 10 and 17 are shown in Chapter 4. For Language A1 we show graphs for sizes 6-15 in Figure A.2.



(a) Exhaustive enumeration results, showing heuristics of size 6.

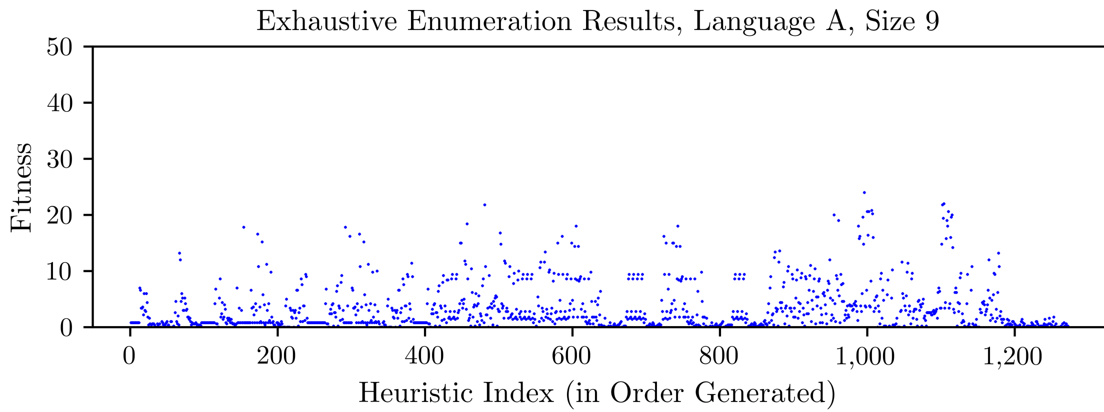


(b) Exhaustive enumeration results, showing heuristics of size 7.

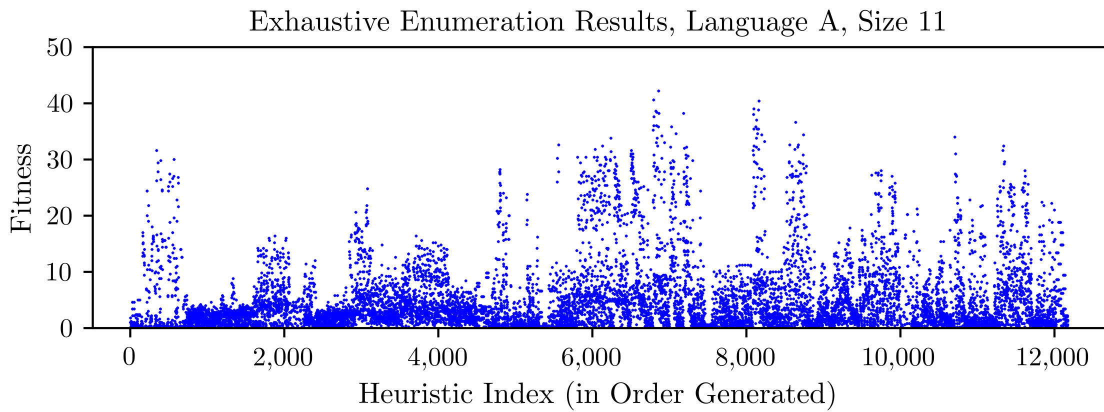


(c) Exhaustive enumeration results, showing heuristics of size 8.

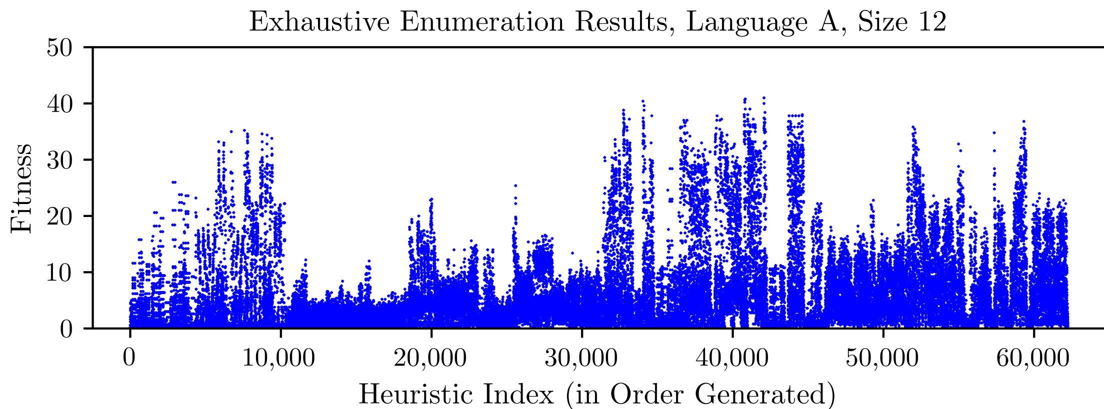
Figure A.1: Results from the exhaustive enumeration experiment performed on Language A, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic.



(d) Exhaustive enumeration results, showing heuristics of size 9.

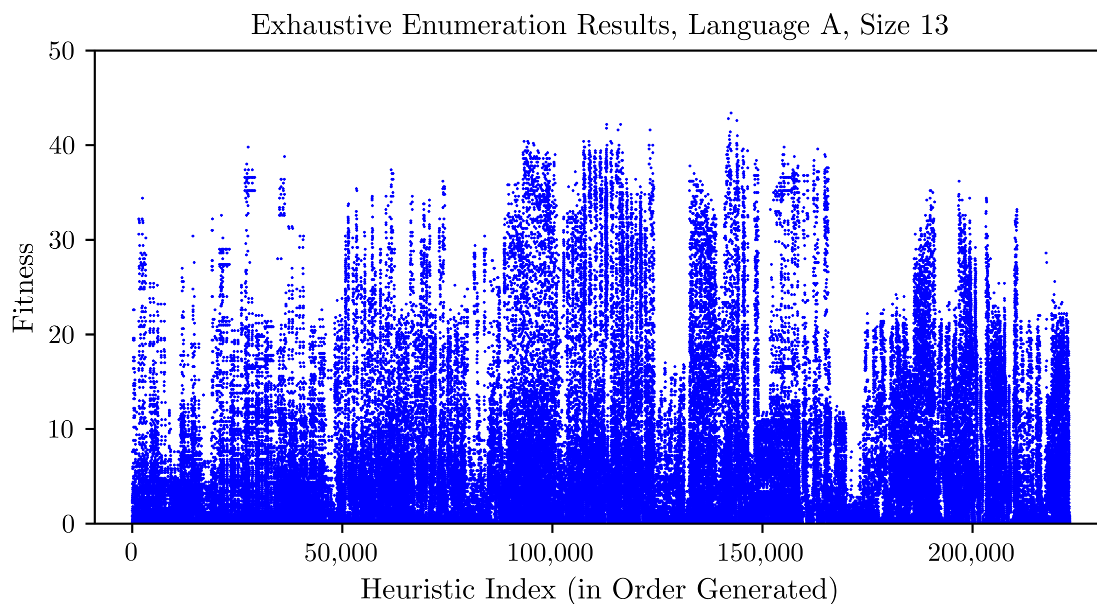


(e) Exhaustive enumeration results, showing heuristics of size 11.

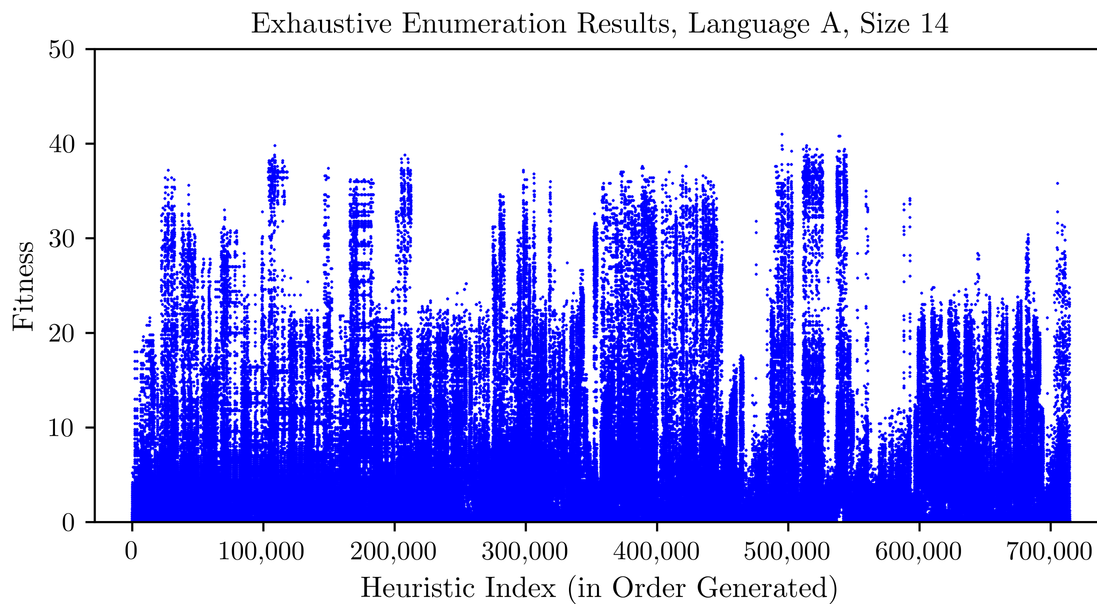


(f) Exhaustive enumeration results, showing heuristics of size 12.

Figure A.1: Results from the exhaustive enumeration experiment performed on Language A, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic. (Continued)

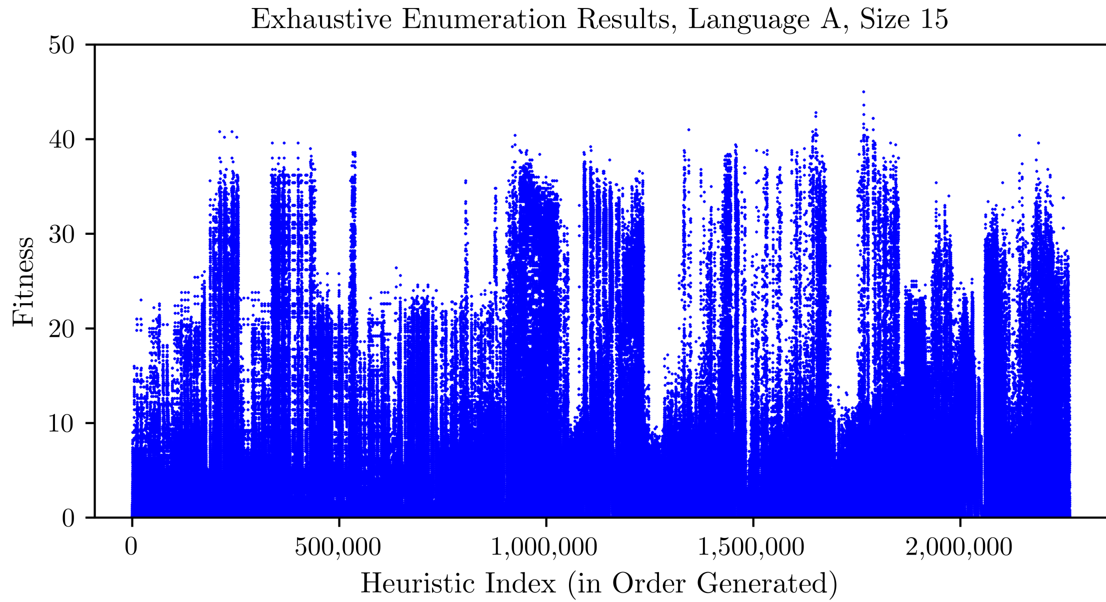


(g) Exhaustive enumeration results, showing heuristics of size 13.

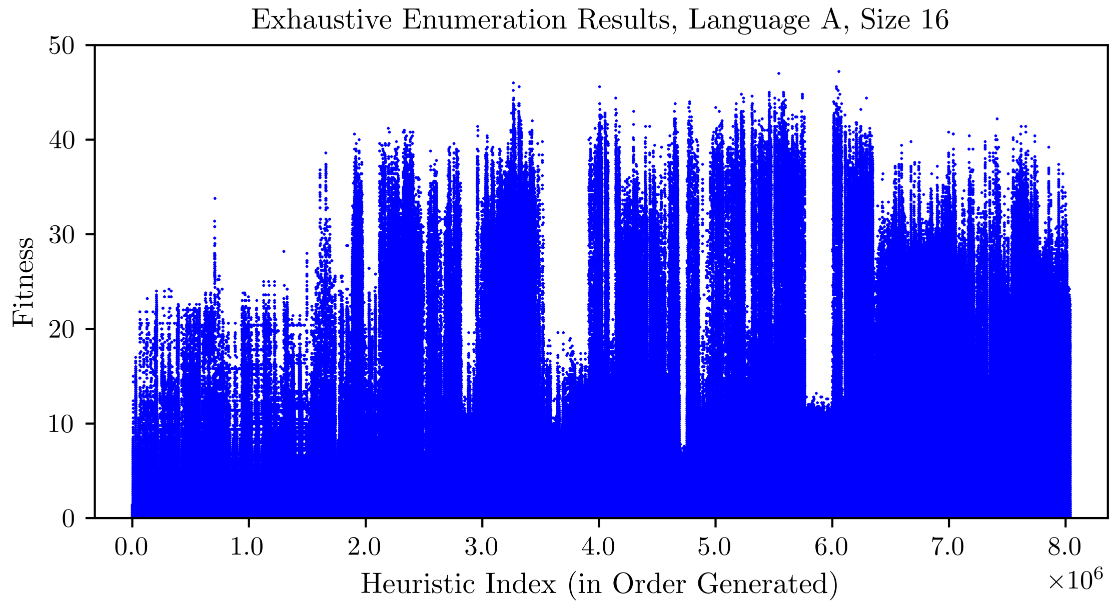


(h) Exhaustive enumeration results, showing heuristics of size 14.

Figure A.1: Results from the exhaustive enumeration experiment performed on Language A, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic. (Continued)

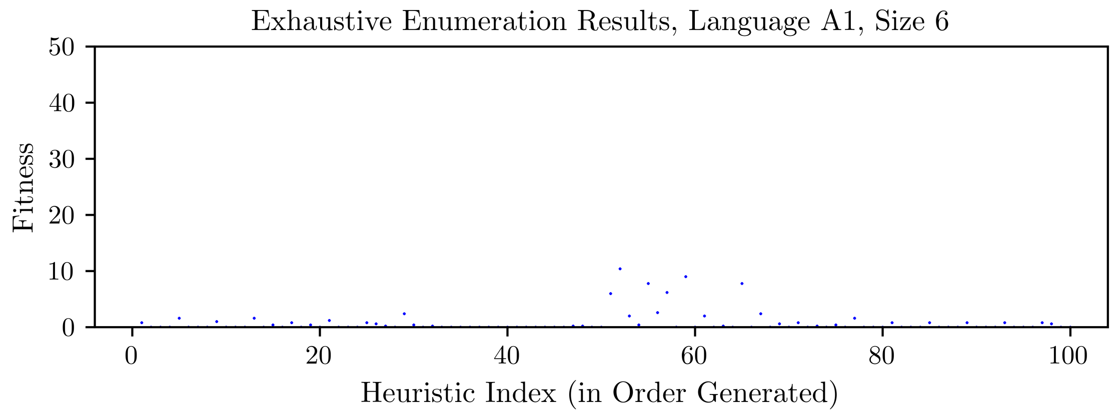


(i) Exhaustive enumeration results, showing heuristics of size 15.

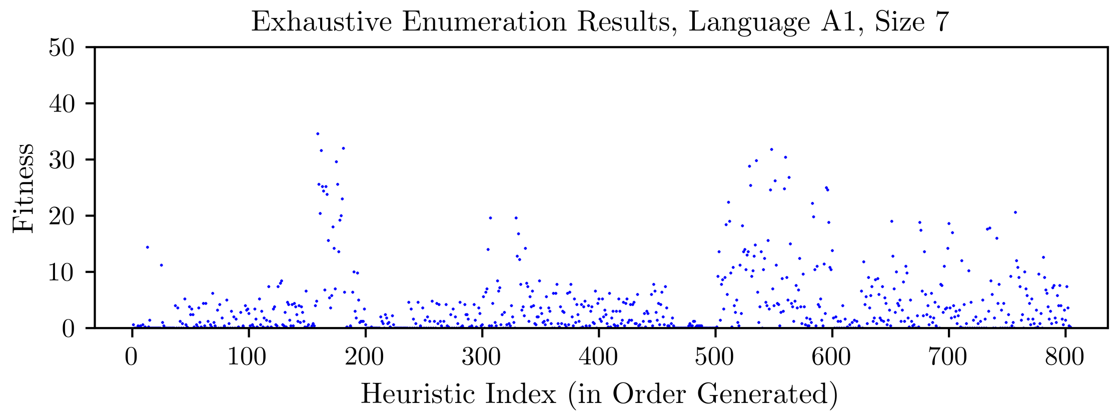


(j) Exhaustive enumeration results, showing heuristics of size 16.

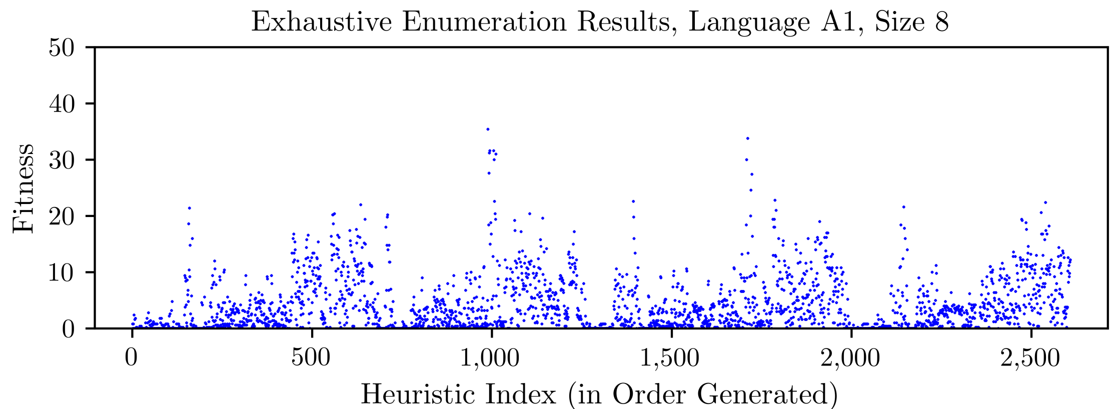
Figure A.1: Results from the exhaustive enumeration experiment performed on Language A, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic. (Continued)



(a) Exhaustive enumeration results, showing heuristics of size 6.

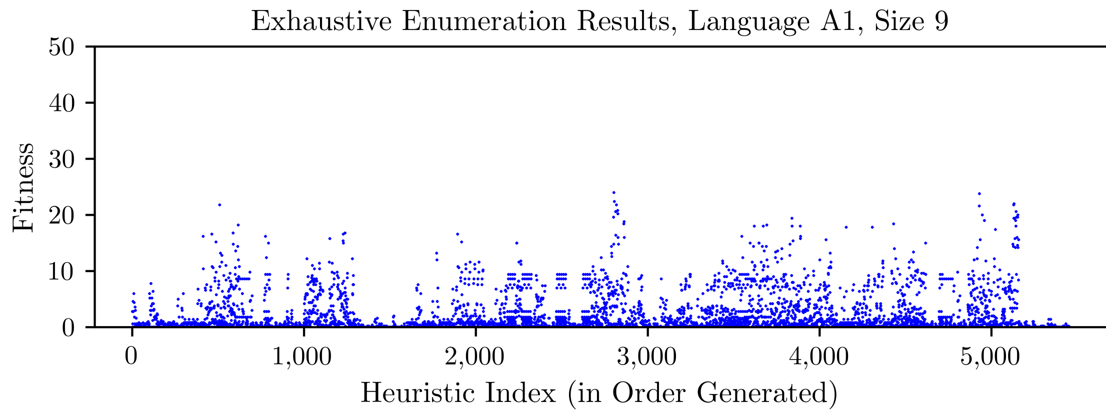


(b) Exhaustive enumeration results, showing heuristics of size 7.

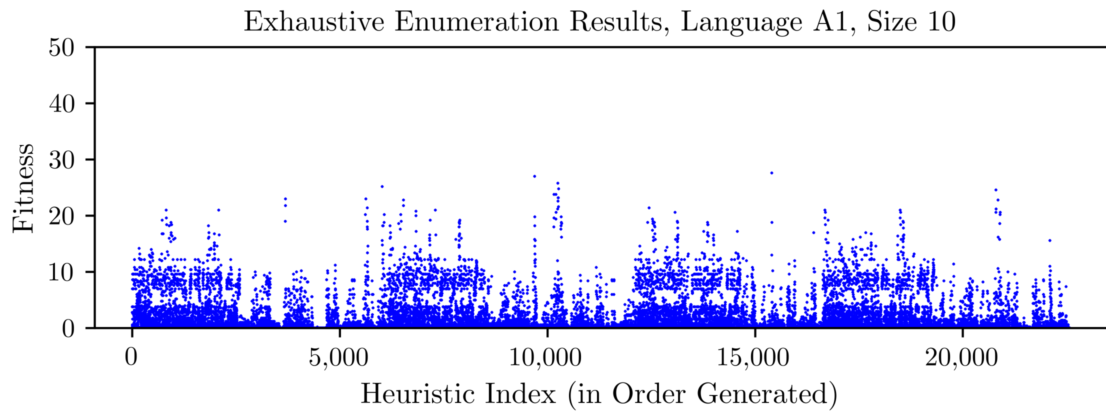


(c) Exhaustive enumeration results, showing heuristics of size 8.

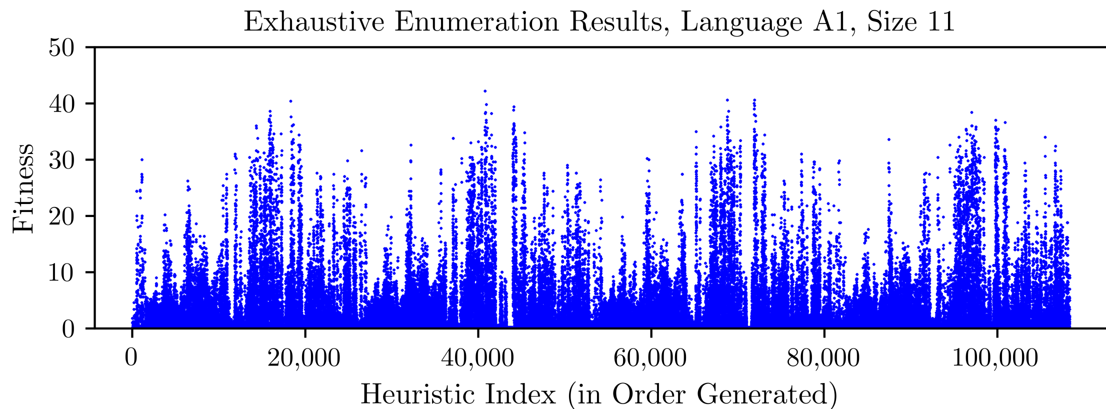
Figure A.2: Results from the exhaustive enumeration experiment performed on Language A1, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic.



(d) Exhaustive enumeration results, showing heuristics of size 9.

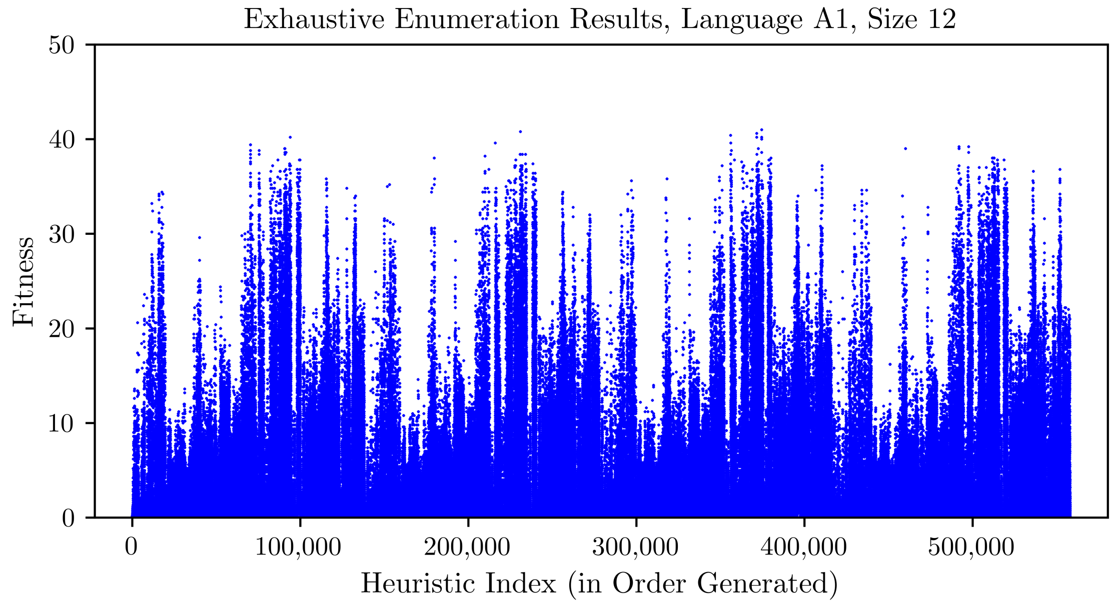


(e) Exhaustive enumeration results, showing heuristics of size 10.

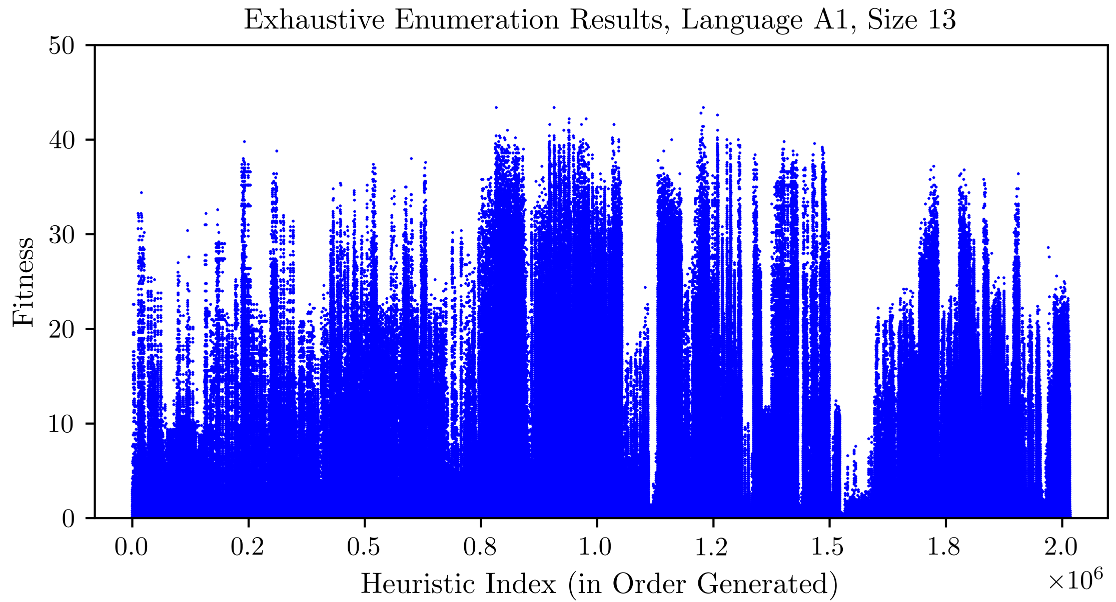


(f) Exhaustive enumeration results, showing heuristics of size 11.

Figure A.2: Results from the exhaustive enumeration experiment performed on Language A1, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic. (Continued)

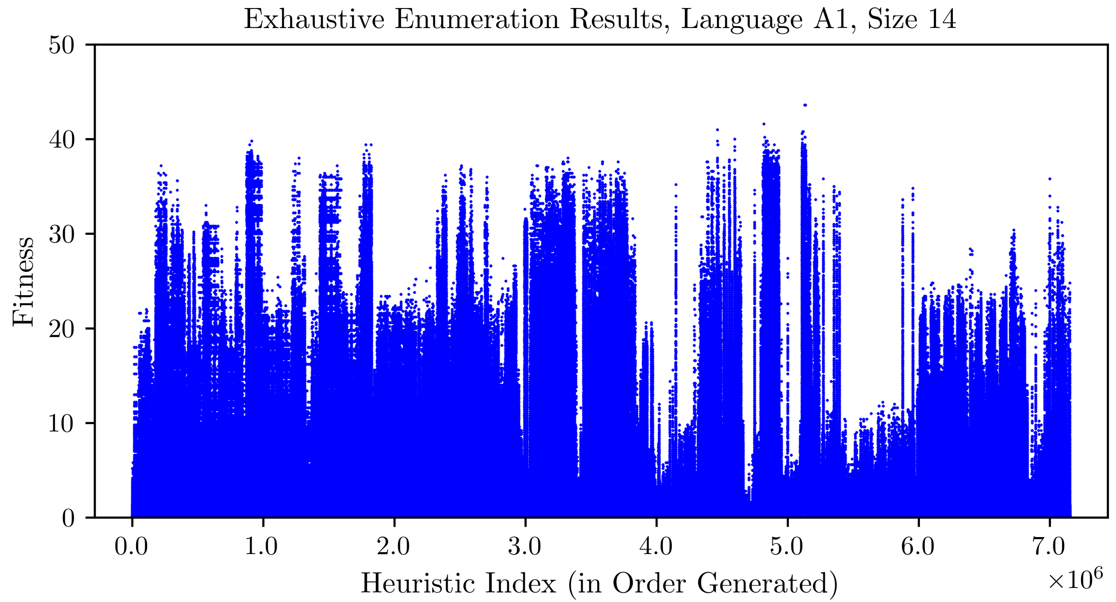


(g) Exhaustive enumeration results, showing heuristics of size 12.

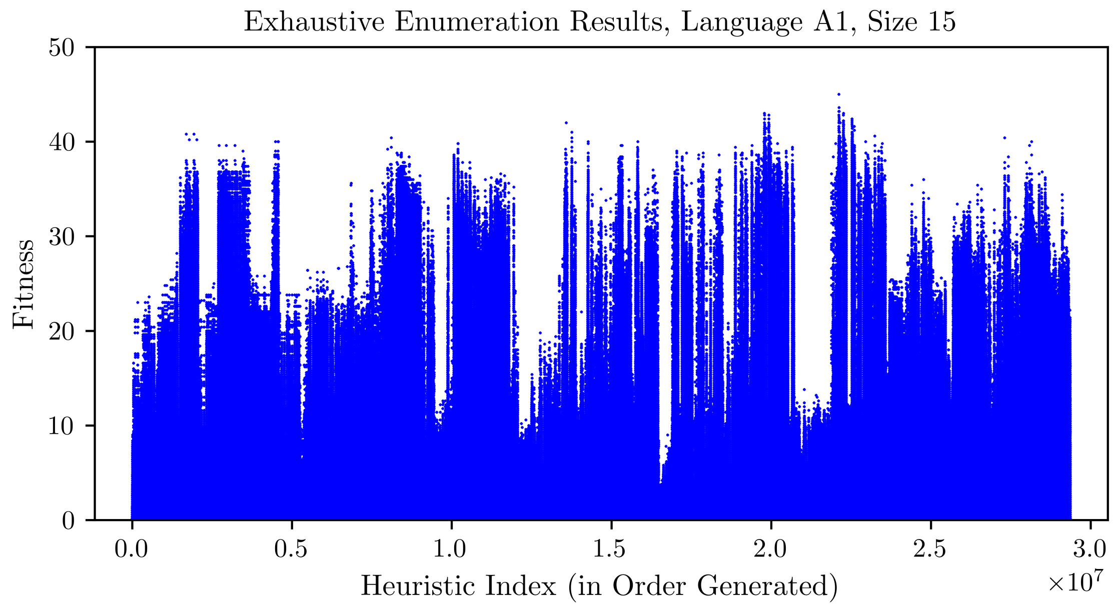


(h) Exhaustive enumeration results, showing heuristics of size 13.

Figure A.2: Results from the exhaustive enumeration experiment performed on Language A1, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic. (Continued)



(i) Exhaustive enumeration results, showing heuristics of size 14.



(j) Exhaustive enumeration results, showing heuristics of size 15.

Figure A.2: Results from the exhaustive enumeration experiment performed on Language A1, as detailed in Chapter 4. Each point in each graph represents the fitness of a heuristic. (Continued)

Appendix B

Genetic Programming Created Heuristics

In this Appendix we show three heuristics created from the genetic programming experiments detailed in Chapters 4 and 7. Each heuristic shown here is the heuristic with the highest reported fitness from one of the GP experiments performed using Language A, Language A1 and Language B. In Figure B.1 the heuristic called GP-A-5 is shown, created from the GP experiments performed using Language A. In Figure B.2 the heuristic called GP-A1-3 is shown, created from the GP experiments performed using Language A1. In Figure B.3 the heuristic called GP-B-5 is shown, created from the GP experiments performed using Language B.

```
let A =
IfVarCompare < PosGain
 IfVarCond < NegGain 4
  IfVarCond = NetGain 4
   GetBestVarSnd RBC-0 NetGain
   IfRandLt 0.9
    PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  GetBestVarSnd RBC-0 NegGain
 GetOldestVar
  IfTabu 10
   IfVarCompare = NegGain
    PickRandomVar RBC-0
    IfTabu 10
     IfTabu 20
      IfVarCond < NetGain -2
       GetBestVar RBC-0 NetGain
       GetOldestVar
        GetBestVar RBC-0 NegGain
        GetBestVarSnd RBC-0
          NegGain
      GetOldestVar
       IfVarCond < NetGain 3
        GetBestVar RBC-0 NegGain
        GetBestVar RBC-0 PosGain
       GetOldestVar
        PickRandomVar RBC-0
        PickRandomVar RBC-0
     GetBestVar RBC-0 NegGain
   IfTabu 30
    GetBestVarSnd RBC-0 NegGain
    GetBestVar RBC-0 NetGain
  IfVarCompare = NetGain
   IfNotMinAge RBC-0
    GetBestVarSnd RBC-0 PosGain
    GetBestVarSnd RBC-0 PosGain
   GetBestVar RBC-0 NegGain
```

(a) Part A of the GP-A-5 heuristic.

```
let B =
IfVarCompare < PosGain
 IfVarCond < NegGain 4
  IfVarCond = NetGain 4
   GetBestVarSnd RBC-0 NetGain
   IfRandLt 0.9
    PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  GetBestVarSnd RBC-0 NegGain
 GetOldestVar
  IfTabu 10
   IfVarCompare = NetGain
    PickRandomVar RBC-0
    IfTabu 10
     IfTabu 20
      IfVarCond < NetGain -2
       GetBestVar RBC-0 NetGain
       GetOldestVar
        GetBestVar RBC-0 NegGain
        GetBestVarSnd RBC-0
          NegGain
      IfNotMinAge RBC-0
       IfNotMinAge RBC-0
        PickRandomVar RBC-0
        IfRandLt 0.7
         PickRandomVar RBC-0
         GetBestVar RBC-0 PosGain
       GetBestVar RBC-0 PosGain
     GetBestVar RBC-0 NegGain
   IfTabu 30
    GetBestVarSnd RBC-0 NegGain
    GetBestVar RBC-0 NetGain
  IfVarCompare = NegGain
   IfNotMinAge RBC-0
    GetBestVarSnd RBC-0 PosGain
    GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 NegGain
```

(b) Part B of the GP-A-5 heuristic.

Figure B.1: The GP-A-5 heuristic. Fitness value of 57.0.

```
let C =
IfVarCond < PosGain 3
 IfRandLt 0.7
  GetOldestVar
   IfVarCompare = NetGain
    IfVarCond = PosGain 5
     IfVarCond = PosGain -2 { B }
      GetBestVarSnd RBC-0 NegGain
     IfTabu 20
      IfVarCompare = PosGain
       PickRandomVar RBC-0
       PickRandomVar RBC-0
      IfVarCompare < NetGain
       GetOldestVar
        IfRandLt 0.5
         PickRandomVar RBC-0
         IfVarCond <= NetGain -1
          PickRandomVar RBC-0
          GetBestVarSnd RBC-0
            PosGain
        IfNotMinAge RBC-0
         PickRandomVar RBC-0
         GetBestVar RBC-0 NetGain
       IfTabu 10
        IfTabu 30
         GetBestVarSnd RBC-0
           PosGain
         GetBestVar RBC-0 PosGain
        IfVarCompare <= NetGain
         GetBestVar RBC-0 PosGain
         PickRandomVar RBC-0
    GetBestVar RBC-0 NegGain
   IfTabu 40
    PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  PickRandomVar RBC-0
 GetBestVar RBC-0 NegGain
```

(c) Part C of the GP-A-5 heuristic.

```
let D =
IfNotMinAge RBC-0
 IfVarCond = NegGain 5
  IfVarCond = PosGain -2
   A
   GetBestVarSnd RBC-0 NegGain
  GetOldestVar
   IfTabu 5
    IfVarCompare = NegGain
     IfTabu 40
      IfVarCond <= NetGain -1
       PickRandomVar RBC-0
       GetBestVarSnd RBC-0
         NegGain
      PickRandomVar RBC-0
     IfTabu 40
      PickRandomVar RBC-0
      GetBestVar RBC-0 NetGain
    PickRandomVar RBC-0
   IfTabu 10
    IfRandLt 0.1
     GetBestVarSnd RBC-0 PosGain
     GetOldestVar
      IfVarCond < NetGain 3
       GetBestVar RBC-0 NetGain
       GetBestVar RBC-0 PosGain
      GetOldestVar
       PickRandomVar RBC-0
       PickRandomVar RBC-0
    IfVarCond = NegGain 2
     PickRandomVar RBC-0
     PickRandomVar RBC-0
 GetBestVarSnd RBC-0 NetGain
```

(d) Part D of the GP-A-5 heuristic.

Figure B.1: The GP-A-5 heuristic. Fitness value of 57.0. (Continued)

```
let E =
IfTabu 10
 IfNotMinAge RBC-0
  GetBestVar RBC-0 NegGain
  GetBestVarSnd RBC-0 PosGain
 IfVarCond < PosGain 4
  IfRandLt 0.7
   IfTabu 30
    IfRandLt 0.7
     GetBestVar RBC-0 NetGain
     GetBestVar RBC-0 PosGain
    IfVarCompare <= NetGain
     GetBestVarSnd RBC-0 PosGain
     GetBestVarSnd RBC-0 PosGain
   IfTabu 40
    GetBestVarSnd RBC-0 NegGain
    IfNotMinAge RBC-0
     GetBestVar RBC-0 PosGain
     GetBestVar RBC-0 PosGain
  IfNotMinAge RBC-0
   IfRandLt 0.1
    GetBestVar RBC-0 NegGain
    GetBestVarSnd RBC-0 NegGain
   IfRandLt 0.1
    IfVarCompare = NetGain
     GetBestVarSnd RBC-0 NetGain
     PickRandomVar RBC-0
    GetOldestVar
     PickRandomVar RBC-0
     PickRandomVar RBC-0
```

(e) Part E of the GP-A-5 heuristic.

```
let F =
IfRandLt 0.1
 IfVarCond < NegGain 0
  GetOldestVar
   GetBestVar RBC-0 PosGain
   GetOldestVar
    IfNotMinAge RBC-0
     E
     IfVarCompare < NetGain
      IfVarCompare < NegGain
       GetOldestVar
        PickRandomVar RBC-0
        GetBestVarSnd RBC-0
          NetGain
       PickRandomVar RBC-0
      PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  IfVarCompare = NetGain
   IfRandLt 0.3
    GetBestVarSnd RBC-0 NetGain
    GetBestVar RBC-0 PosGain
   IfRandLt 0.3
    GetBestVar RBC-0 PosGain
    GetBestVar RBC-0 NegGain
 GetOldestVar
  GetOldestVar
   PickRandomVar RBC-0
   GetOldestVar
    GetBestVarSnd RBC-0 PosGain
    GetBestVar RBC-0 NetGain
  PickRandomVar RBC-0
```

(f) Part F of the GP-A-5 heuristic.

Figure B.1: The GP-A-5 heuristic. Fitness value of 57.0. (Continued)

```
let G =
IfVarCompare = NetGain
 GetOldestVar
  IfNotMinAge RBC-0
   IfNotMinAge RBC-0
    IfRandLt 0.1
     GetBestVarSnd RBC-0 PosGain
     GetOldestVar
      IfVarCond < NetGain 2
       GetBestVar RBC-0 NegGain
       GetBestVar RBC-0 PosGain
      GetOldestVar
       PickRandomVar RBC-0
       PickRandomVar RBC-0
    IfNotMinAge RBC-0
     GetBestVarSnd RBC-0 NetGain
     PickRandomVar RBC-0
   IfRandLt 0.7
    IfRandLt 0.3
     GetBestVar RBC-0 PosGain
     GetBestVarSnd RBC-0 NegGain
    GetOldestVar
     GetBestVarSnd RBC-0 NegGain
     GetBestVarSnd RBC-0 NetGain
  GetBestVarSnd RBC-0 NetGain
 B
```

(g) Part G of the GP-A-5 heuristic.

```
let H =
IfVarCompare <= PosGain
 D
 IfNotMinAge RBC-0
  IfNotMinAge RBC-0
   IfVarCompare = NegGain
    GetBestVar RBC-0 PosGain
    PickRandomVar RBC-0
   IfVarCond < NegGain -2
    IfNotMinAge RBC-0 { G }
     IfVarCond <= NegGain 3
      GetBestVarSnd RBC-0 PosGain
      IfVarCond = PosGain 2
       IfVarCompare <= PosGain
        GetBestVar RBC-0 NegGain
        IfRandLt 0.5
         IfVarCompare < PosGain
          GetBestVar RBC-0
            PosGain
          PickRandomVar RBC-0
         IfVarCond <= NetGain 2
          GetBestVar RBC-0
            NegGain
          GetBestVar RBC-0
            PosGain
       GetBestVar RBC-0 NetGain
    GetBestVarSnd RBC-0 NetGain
  IfVarCompare = NegGain
   IfTabu 10
    IfVarCond <= PosGain -1
     GetBestVarSnd RBC-0 NetGain
     GetBestVarSnd RBC-0 PosGain
    PickRandomVar RBC-0
   IfTabu 40
    PickRandomVar RBC-0
    GetBestVar RBC-0 NetGain
```

(h) Part H of the GP-A-5 heuristic.

Figure B.1: The GP-A-5 heuristic. Fitness value of 57.0. (Continued)

```
let I =
IfRandLt 0.1
 GetBestVarSnd RBC-0 NegGain
 GetOldestVar
  IfVarCond < NetGain 5
  GetBestVar RBC-0 NetGain
  IfTabu 30
    PickRandomVar RBC-0
    GetOldestVar
     H
     IfRandLt 0.1
      IfVarCond < NetGain 2
      GetOldestVar
        GetBestVar RBC-0 NetGain
        GetOldestVar
         PickRandomVar RBC-0
         GetBestVar RBC-0 PosGain
      IfVarCond <= NetGain -2
        PickRandomVar RBC-0
        IfVarCompare < NetGain
         GetOldestVar
          GetBestVarSnd RBC-0
             NetGain
          PickRandomVar RBC-0
         GetOldestVar
          PickRandomVar RBC-0
          GetBestVar RBC-0 NetGain
      GetOldestVar
      GetOldestVar
        GetBestVarSnd RBC-0 PosGain
        GetOldestVar
         GetBestVarSnd RBC-0 PosGain
         GetBestVar RBC-0 NetGain
      PickRandomVar RBC-0
  GetOldestVar
  PickRandomVar RBC-0
  PickRandomVar RBC-0
```

(i) Part I of the GP-A-5 heuristic

```
let J =
IfTabu 30
 IfRandLt 0.1
  GetBestVarSnd RBC-0 NetGain
  GetOldestVar
   IfVarCond < NetGain 5 { I }
    IfTabu 30
     PickRandomVar RBC-0
     GetOldestVar
      IfVarCompare <= PosGain
       IfVarCompare <= NegGain
        PickRandomVar RBC-0
        IfTabu 40
         GetBestVarSnd RBC-0
           NegGain
         IfNotMinAge RBC-0
          GetBestVar RBC-0
            PosGain
          GetBestVar RBC-0
            PosGain
       IfNotMinAge RBC-0
        GetBestVarSnd RBC-0
          PosGain
        IfVarCond <= NegGain 0
         GetBestVarSnd RBC-0
           NegGain
         IfRandLt 0.1
          GetBestVarSnd RBC-0
            NetGain
          PickRandomVar RBC-0
      F
   GetOldestVar
    PickRandomVar RBC-0
    PickRandomVar RBC-0
 IfVarCond = NetGain 4
  PickRandomVar RBC-0
  PickRandomVar RBC-0
```

(j) Part J of the GP-A-5 heuristic.

Figure B.1: The GP-A-5 heuristic. Fitness value of 57.0. (Continued)

```
let K =
IfTabu 5
 IfVarCompare = NegGain
  GetOldestVar
   GetBestVarSnd RBC-0 NetGain
   GetBestVar RBC-0 PosGain
  GetBestVarSnd RBC-0 PosGain
 IfNotMinAge RBC-0
  IfNotMinAge RBC-0
   IfVarCond = PosGain 3
    GetOldestVar
     GetBestVar RBC-0 NetGain
     GetBestVar RBC-0 PosGain
    GetOldestVar
     IfTabu 5
      IfVarCompare = NegGain
       IfTabu 20
        IfVarCond <= NetGain -1
         PickRandomVar RBC-0
         GetBestVarSnd RBC-0
           PosGain
        PickRandomVar RBC-0
       IfTabu 40
        PickRandomVar RBC-0
        GetBestVar RBC-0 NetGain
      PickRandomVar RBC-0
     J
   GetBestVarSnd RBC-0 NetGain
  IfVarCond <= NetGain 0
   GetBestVarSnd RBC-0 PosGain
   GetBestVar RBC-0 PosGain
```

(k) Part K of the GP-A-5 heuristic.

```
let L =
GetOldestVar
 GetBestVarSnd RBC-0 PosGain
 IfTabu 20
  IfVarCompare = PosGain
   PickRandomVar RBC-0
   PickRandomVar RBC-0
  IfVarCompare < NetGain
   GetOldestVar
    IfRandLt 0.5
     PickRandomVar RBC-0
     IfVarCond <= NetGain -1
      PickRandomVar RBC-0
      GetBestVarSnd RBC-0 PosGain
    IfNotMinAge RBC-0
     PickRandomVar RBC-0
     GetBestVar RBC-0 NetGain
   IfTabu 10
    IfTabu 30
     GetBestVarSnd RBC-0 PosGain
     GetBestVar RBC-0 PosGain
    IfVarCompare <= NetGain
     GetBestVar RBC-0 PosGain
     PickRandomVar RBC-0
```

(l) Part L of the GP-A-5 heuristic.

Figure B.1: The GP-A-5 heuristic. Fitness value of 57.0. (Continued)

```
let M =
IfNotMinAge RBC-0
 IfVarCond <= NegGain 4
  GetBestVar RBC-0 NetGain
  GetOldestVar
   GetBestVar RBC-0 NetGain
   IfRandLt 0.7
    PickRandomVar RBC-0
    IfRandLt 0.1
     GetBestVarSnd RBC-0 NetGain
     PickRandomVar RBC-0
 IfRandLt 0.1
  PickRandomVar RBC-0
  GetOldestVar
   GetOldestVar
    GetBestVarSnd RBC-0 PosGain
    GetBestVarSnd RBC-0 NegGain
   IfTabu 50
    GetBestVarSnd RBC-0 PosGain
    PickRandomVar RBC-0
```

(m) Part M of the GP-A-5 heuristic.

```
IfVarCompare <= NegGain
 IfTabu 20
  IfTabu 20
   GetBestVar RBC-0 NetGain
   IfNotMinAge RBC-0
    GetBestVar RBC-0 NegGain
    GetBestVar RBC-0 NegGain
  IfNotMinAge RBC-0
   IfTabu 40
    IfRandLt 0.1 { K }
     IfTabu 20
      IfVarCond = NetGain 5
       PickRandomVar RBC-0
       GetOldestVar
        GetBestVar RBC-0 NegGain
        GetBestVar RBC-0 NegGain
      IfNotMinAge RBC-0
       GetOldestVar
        GetBestVarSnd RBC-0
          NetGain
        GetBestVarSnd RBC-0
          NetGain
       IfVarCompare < NetGain
        GetOldestVar
         GetBestVarSnd RBC-0
           NetGain
         PickRandomVar RBC-0
        PickRandomVar RBC-0
    M
   L
 GetOldestVar
  GetBestVar RBC-0 NetGain
  GetBestVar RBC-0 PosGain
```

(n) The final part of the GP-A-5 heuristic.

Figure B.1: The GP-A-5 heuristic. Fitness value of 57.0. (Continued)

```
let A =
IfVarCompare < PosGain
 IfVarCond < NegGain 4
  IfVarCond = NetGain 4
   GetBestVarSnd RBC-0 NetGain
   IfRandLt 0.9
    PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  GetBestVarSnd RBC-0 NegGain
 GetOldestVar
  IfTabu 10
   IfVarCompare = NegGain
    PickRandomVar RBC-0
    IfTabu 10
     IfTabu 20
      IfVarCond < NetGain -2
       GetBestVar RBC-0 NetGain
       GetOldestVar
        GetBestVar RBC-0 NegGain
        GetBestVarSnd RBC-0
          NegGain
      GetOldestVar
       IfVarCond < NetGain 3
        GetBestVar RBC-0 NegGain
        GetBestVar RBC-0 PosGain
       GetOldestVar
        PickRandomVar RBC-0
        PickRandomVar RBC-0
     GetBestVar RBC-0 NegGain
   IfTabu 30
    GetBestVarSnd RBC-0 NegGain
    GetBestVar RBC-0 NetGain
  IfVarCompare = NetGain
   IfNotMinAge RBC-0
    GetBestVarSnd RBC-0 PosGain
    GetBestVarSnd RBC-0 PosGain
   GetBestVar RBC-0 NegGain
```

(a) Part A of the GP-A1-3 heuristic.

```
let B =
IfVarCompare < PosGain
 IfVarCond < NegGain 4
  IfVarCond = NetGain 4
   GetBestVarSnd RBC-0 NetGain
   IfRandLt 0.9
    PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  GetBestVarSnd RBC-0 NegGain
 GetOldestVar
  IfTabu 10
   IfVarCompare = NetGain
    PickRandomVar RBC-0
    IfTabu 10
     IfTabu 20
      IfVarCond < NetGain -2
       GetBestVar RBC-0 NetGain
       GetOldestVar
        GetBestVar RBC-0 NegGain
        GetBestVarSnd RBC-0
          NegGain
      IfNotMinAge RBC-0
       IfNotMinAge RBC-0
        PickRandomVar RBC-0
        IfRandLt 0.7
         PickRandomVar RBC-0
         GetBestVar RBC-0 PosGain
       GetBestVar RBC-0 PosGain
     GetBestVar RBC-0 NegGain
   IfTabu 30
    GetBestVarSnd RBC-0 NegGain
    GetBestVar RBC-0 NetGain
  IfVarCompare = NegGain
   IfNotMinAge RBC-0
    GetBestVarSnd RBC-0 PosGain
    GetBestVar RBC-0 NegGain
   GetBestVar RBC-0 NegGain
```

(b) Part B of the GP-A1-3 heuristic.

Figure B.2: The GP-A1-3 heuristic. Fitness value of 57.4.

```
let C =
IfVarCond < PosGain 3
 IfRandLt 0.7
  GetOldestVar
   IfVarCompare = NetGain
    IfVarCond = PosGain 5
     IfVarCond = PosGain -2 { B }
      GetBestVarSnd RBC-0 NegGain
     IfTabu 20
      IfVarCompare = PosGain
       PickRandomVar RBC-0
       PickRandomVar RBC-0
      IfVarCompare < NetGain
       GetOldestVar
        IfRandLt 0.5
         PickRandomVar RBC-0
         IfVarCond <= NetGain -1
          PickRandomVar RBC-0
          GetBestVarSnd RBC-0
            PosGain
        IfNotMinAge RBC-0
         PickRandomVar RBC-0
         GetBestVar RBC-0 NetGain
       IfTabu 10
        IfTabu 30
         GetBestVarSnd RBC-0
           PosGain
         GetBestVar RBC-0 PosGain
        IfVarCompare <= NetGain
         GetBestVar RBC-0 PosGain
         PickRandomVar RBC-0
    GetBestVar RBC-0 NegGain
   IfTabu 40
    PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  PickRandomVar RBC-0
 GetBestVar RBC-0 NegGain
```

(c) Part C of the GP-A1-3 heuristic.

```
let D =
IfNotMinAge RBC-0
 IfVarCond = NegGain 5
  IfVarCond = PosGain -2
   A
   GetBestVarSnd RBC-0 NegGain
  GetOldestVar
   IfTabu 5
    IfVarCompare = NegGain
     IfTabu 40
      IfVarCond <= NetGain -1
       PickRandomVar RBC-0
       GetBestVarSnd RBC-0
         NegGain
      PickRandomVar RBC-0
     IfTabu 40
      PickRandomVar RBC-0
      GetBestVar RBC-0 NetGain
    PickRandomVar RBC-0
   IfTabu 10
    IfRandLt 0.1
     GetBestVarSnd RBC-0 PosGain
     GetOldestVar
      IfVarCond < NetGain 3
       GetBestVar RBC-0 NetGain
       GetBestVar RBC-0 PosGain
      GetOldestVar
       PickRandomVar RBC-0
       PickRandomVar RBC-0
    IfVarCond = NegGain 2
     PickRandomVar RBC-0
     PickRandomVar RBC-0
 GetBestVarSnd RBC-0 NetGain
```

(d) Part D of the GP-A1-3 heuristic.

Figure B.2: The GP-A1-3 heuristic. Fitness value of 57.4. (Continued)

```
let E =
IfTabu 10
 IfNotMinAge RBC-0
  GetBestVar RBC-0 NegGain
  GetBestVarSnd RBC-0 PosGain
 IfVarCond < PosGain 4
  IfRandLt 0.7
   IfTabu 30
    IfRandLt 0.7
     GetBestVar RBC-0 NetGain
     GetBestVar RBC-0 PosGain
    IfVarCompare <= NetGain
     GetBestVarSnd RBC-0 PosGain
     GetBestVarSnd RBC-0 PosGain
   IfTabu 40
    GetBestVarSnd RBC-0 NegGain
    IfNotMinAge RBC-0
     GetBestVar RBC-0 PosGain
     GetBestVar RBC-0 PosGain
  IfNotMinAge RBC-0
   IfRandLt 0.1
    GetBestVar RBC-0 NegGain
    GetBestVarSnd RBC-0 NegGain
   IfRandLt 0.1
    IfVarCompare = NetGain
     GetBestVarSnd RBC-0 NetGain
     PickRandomVar RBC-0
    GetOldestVar
     PickRandomVar RBC-0
     PickRandomVar RBC-0
```

(e) Part E of the GP-A1-3 heuristic.

```
let F =
IfRandLt 0.1
 IfVarCond < NegGain 0
  GetOldestVar
   GetBestVar RBC-0 PosGain
   GetOldestVar
    IfNotMinAge RBC-0
     E
     IfVarCompare < NetGain
      IfVarCompare < NegGain
       GetOldestVar
        PickRandomVar RBC-0
        GetBestVarSnd RBC-0
          NetGain
       PickRandomVar RBC-0
      PickRandomVar RBC-0
    GetBestVar RBC-0 PosGain
  IfVarCompare = NetGain
   IfRandLt 0.3
    GetBestVarSnd RBC-0 NetGain
    GetBestVar RBC-0 PosGain
   IfRandLt 0.3
    GetBestVar RBC-0 PosGain
    GetBestVar RBC-0 NegGain
 GetOldestVar
  GetOldestVar
   PickRandomVar RBC-0
   GetOldestVar
    GetBestVarSnd RBC-0 PosGain
    GetBestVar RBC-0 NetGain
  PickRandomVar RBC-0
```

(f) Part F of the GP-A1-3 heuristic.

Figure B.2: The GP-A1-3 heuristic. Fitness value of 57.4. (Continued)

```
let G =
IfVarCompare = NetGain
 GetOldestVar
  IfNotMinAge RBC-0
   IfNotMinAge RBC-0
    IfRandLt 0.1
     GetBestVarSnd RBC-0 PosGain
     GetOldestVar
      IfVarCond < NetGain 2
       GetBestVar RBC-0 NegGain
       GetBestVar RBC-0 PosGain
      GetOldestVar
       PickRandomVar RBC-0
       PickRandomVar RBC-0
    IfNotMinAge RBC-0
     GetBestVarSnd RBC-0 NetGain
     PickRandomVar RBC-0
   IfRandLt 0.7
    IfRandLt 0.3
     GetBestVar RBC-0 PosGain
     GetBestVarSnd RBC-0 NegGain
    GetOldestVar
     GetBestVarSnd RBC-0 NegGain
     GetBestVarSnd RBC-0 NetGain
  GetBestVarSnd RBC-0 NetGain
 C
```

(g) Part G of the GP-A1-3 heuristic.

```
let H =
IfVarCompare <= PosGain { D }
 IfNotMinAge RBC-0
  IfNotMinAge RBC-0
   IfVarCompare = NegGain
    GetBestVar RBC-0 PosGain
    PickRandomVar RBC-0
   IfVarCond < NegGain -2
    IfNotMinAge RBC-0 { G }
     IfVarCond <= NegGain 3
      GetBestVarSnd RBC-0 PosGain
      IfVarCond = PosGain 2
       IfVarCompare <= PosGain
        GetBestVar RBC-0 NegGain
        IfRandLt 0.5
         IfVarCompare < PosGain
          GetBestVar RBC-0
            PosGain
          PickRandomVar RBC-0
         IfVarCond <= NetGain 2
          GetBestVar RBC-0
            NegGain
          GetBestVar RBC-0
            PosGain
       GetBestVar RBC-0 NetGain
    GetBestVarSnd RBC-0 NetGain
  IfVarCompare = NegGain
   IfTabu 10
    IfVarCond <= PosGain -1
     GetBestVarSnd RBC-0 NetGain
     GetBestVarSnd RBC-0 PosGain
    PickRandomVar RBC-0
   IfTabu 40
    PickRandomVar RBC-0
    GetBestVar RBC-0 NetGain
```

(h) Part H of the GP-A1-3 heuristic.

Figure B.2: The GP-A1-3 heuristic. Fitness value of 57.4. (Continued)

```
let I =
IfRandLt 0.1
 GetBestVarSnd RBC-0 NegGain
 GetOldestVar
  IfVarCond < NetGain 5
   GetBestVar RBC-0 NetGain
   IfTabu 30
    PickRandomVar RBC-0
    GetOldestVar { H }
     IfRandLt 0.1
      IfVarCond < NetGain 2
       GetOldestVar
        GetBestVar RBC-0 NetGain
        GetOldestVar
         PickRandomVar RBC-0
         GetBestVar RBC-0 PosGain
       IfVarCond <= NetGain -2
        PickRandomVar RBC-0
        IfVarCompare < NetGain
         GetOldestVar
          GetBestVarSnd RBC-0
            NetGain
          PickRandomVar RBC-0
         GetOldestVar
          PickRandomVar RBC-0
          GetBestVar RBC-0 NetGain
      GetOldestVar
       GetOldestVar
        GetBestVarSnd RBC-0 PosGain
        GetOldestVar
         GetBestVarSnd RBC-0
           PosGain
         GetBestVar RBC-0 NetGain
       PickRandomVar RBC-0
  GetOldestVar
   PickRandomVar RBC-0
   PickRandomVar RBC-0
```

(i) Part I of the GP-A1-3 heuristic.

```
let J =
IfTabu 30
 IfRandLt 0.1
  GetBestVarSnd RBC-0 NetGain
  GetOldestVar
   IfVarCond < NetGain 5 { I }
    IfTabu 30
     PickRandomVar RBC-0
     GetOldestVar
      IfVarCompare <= PosGain
       IfVarCompare <= NegGain
        PickRandomVar RBC-0
        IfTabu 40
         GetBestVarSnd RBC-0
           NegGain
         IfNotMinAge RBC-0
          GetBestVar RBC-0
            PosGain
          GetBestVar RBC-0
            PosGain
       IfNotMinAge RBC-0
        GetBestVarSnd RBC-0
          PosGain
        IfVarCond <= NegGain 0
         GetBestVarSnd RBC-0
           NegGain
         IfRandLt 0.1
          GetBestVarSnd RBC-0
            NetGain
          PickRandomVar RBC-0
      F
   GetOldestVar
    PickRandomVar RBC-0
    PickRandomVar RBC-0
 IfVarCond = NetGain 4
  PickRandomVar RBC-0
  PickRandomVar RBC-0
```

(j) Part J of the GP-A1-3 heuristic.

Figure B.2: The GP-A1-3 heuristic. Fitness value of 57.4. (Continued)

```
let K =
IfTabu 5
 IfVarCompare = NegGain
  GetOldestVar
   GetBestVarSnd RBC-0 NetGain
   GetBestVar RBC-0 PosGain
  GetBestVarSnd RBC-0 PosGain
 IfNotMinAge RBC-0
  IfNotMinAge RBC-0
   IfVarCond = PosGain 3
    GetOldestVar
     GetBestVar RBC-0 NetGain
     GetBestVar RBC-0 PosGain
    GetOldestVar
     IfTabu 5
      IfVarCompare = NegGain
       IfTabu 20
        IfVarCond <= NetGain -1
         PickRandomVar RBC-0
         GetBestVarSnd RBC-0
           PosGain
        PickRandomVar RBC-0
       IfTabu 40
        PickRandomVar RBC-0
        GetBestVar RBC-0 NetGain
      PickRandomVar RBC-0
     J
   GetBestVarSnd RBC-0 NetGain
  IfVarCond <= NetGain 0
   GetBestVarSnd RBC-0 PosGain
   GetBestVar RBC-0 PosGain
```

(k) Part K of the GP-A1-3 heuristic.

```
let L =
GetOldestVar
 GetBestVarSnd RBC-0 PosGain
 IfTabu 20
  IfVarCompare = PosGain
   PickRandomVar RBC-0
   PickRandomVar RBC-0
  IfVarCompare < NetGain
   GetOldestVar
    IfRandLt 0.5
     PickRandomVar RBC-0
     IfVarCond <= NetGain -1
      PickRandomVar RBC-0
      GetBestVarSnd RBC-0 PosGain
    IfNotMinAge RBC-0
     PickRandomVar RBC-0
     GetBestVar RBC-0 NetGain
   IfTabu 10
    IfTabu 30
     GetBestVarSnd RBC-0 PosGain
     GetBestVar RBC-0 PosGain
    IfVarCompare <= NetGain
     GetBestVar RBC-0 PosGain
     PickRandomVar RBC-0
```

(l) Part L of the GP-A1-3 heuristic.

Figure B.2: The GP-A1-3 heuristic. Fitness value of 57.4. (Continued)

```
let M =
IfNotMinAge RBC-0
 IfVarCond <= NegGain 4
  GetBestVar RBC-0 NetGain
  GetOldestVar
   GetBestVar RBC-0 NetGain
   IfRandLt 0.7
    PickRandomVar RBC-0
    IfRandLt 0.1
     GetBestVarSnd RBC-0 NetGain
     PickRandomVar RBC-0
 IfRandLt 0.1
  PickRandomVar RBC-0
  GetOldestVar
   GetOldestVar
    GetBestVarSnd RBC-0 PosGain
    GetBestVarSnd RBC-0 NegGain
   IfTabu 50
    GetBestVarSnd RBC-0 PosGain
    PickRandomVar RBC-0
```

(m) Part M of the GP-A1-3 heuristic.

```
IfVarCompare <= NegGain
 IfTabu 20
  IfTabu 20
   GetBestVar RBC-0 NetGain
   IfNotMinAge RBC-0
    GetBestVar RBC-0 NegGain
    GetBestVar RBC-0 NegGain
  IfNotMinAge RBC-0
   IfTabu 40
    IfRandLt 0.1
     K
     IfTabu 20
      IfVarCond = NetGain 5
       PickRandomVar RBC-0
       GetOldestVar
        GetBestVar RBC-0 NegGain
        GetBestVar RBC-0 NegGain
      IfNotMinAge RBC-0
       GetOldestVar
        GetBestVarSnd RBC-0
          NetGain
        GetBestVarSnd RBC-0
          NetGain
       IfVarCompare < NetGain
        GetOldestVar
         GetBestVarSnd RBC-0
           NetGain
         PickRandomVar RBC-0
        PickRandomVar RBC-0
    M
   L
```

(n) The final part of the GP-A1-3 heuristic.

Figure B.2: The GP-A1-3 heuristic. Fitness value of 57.4. (Continued)

```
let A =
IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
 GetOldestVar
  IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
   IfVarCompare >= SubPosGain
    IfIsNull { GetBestVarM DecrVars_WA NegGain }
     IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
      IfRandLt 0.9
       GetOldestVar
        IfIsNull { GetBestVarM DecrVars_WA NetGain_WA }
         IfRandLt 0.9
          GetBestVar2 CONF NetGain_WA SubNegGain_WA
          IfVarCompare >= SubPosGain
           IfRandLt 0.3 { GetBestVar2 RBC-0 NegGain PosGain }
            IfRandLt 0.9 { GetBestVarAge CONF NetGain_WA }
             GetBestVarSnd RBC_WA-0 SubNetGain_WA
           IfIsNull
            GetBestVarAgeM SubNetGain_WA
             Filter > SubPosGain 0 CONF
            GetBestVarAge CONF NegGain
        IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
         GetBestVarSnd RBC_WA-0 NetGain
       GetBestVar CONF NetGain_WA
    IfVarCond = SubNetGain 0
     GetBestVar2 CONF SubNetGain_WA NegGain
     IfVarCond = NegGain 0
      IfVarCond >= NegGain 0 { PickRandomVar RBC-1 }
       IfVarCond > SubPosGain_WA 0
        GetBestVar2 RBC-1 NetGain SubPosGain_WA
        GetBestVarSnd RBC-1 NegGain_WA
      IfRandLt 0.9 { GetBestVar2 CONF NetGain NetGain_WA }
       IfIsNull { GetBestVarM DecrVars_WA NetGain_WA }
        IfVarCompare = SubNegGain
         GetBestVar CONF NetGain_WA
         GetBestVarAge CONF NetGain_WA
  IfIsNull { GetBestVarM DecrVars SubPosGain }
   GetBestVarAge RBC_WA-1 NetGain_WA
```

(a) Part A of the GP-B-5 heuristic.

Figure B.3: The GP-B-5 heuristic. Fitness value of 65.0.

```
let B =
IfVarCond = SubNetGain 0
 GetOldestVar
  IfIsNull { GetBestVarM DecrVars NetGain_WA }
   IfRandLt 0.9 { GetBestVar2 CONF NetGain SubNegGain_WA }
    IfVarCompare > SubNetGain_WA
     IfRandLt 0.3 { GetBestVarAge CONF PosGain_WA }
      IfRandLt Adapt { GetBestVarAge CONF NetGain_WA }
       GetBestVarSnd RBC_WA-0 SubNetGain_WA
     IfIsNull
      GetBestVarAgeM SubNetGain_WA
       Filter > SubPosGain 0 RBC_WA-1
      GetBestVarAge CONF PosGain_WA
  IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
   IfVarCompare > NetGain_WA
    IfVarCompare >= NetGain_WA
     GetOldestVar { GetBestVarSnd CONF SubNegGain }
      GetOldestVar { GetBestVarSnd RBC_WA-0 NetGain }
       GetBestVarAge CONF SubNetGain
     GetBestVar CONF NetGain_WA
    IfVarCond > SubPosGain_WA 0
     IfIsNull { GetBestVarM DecrVars SubPosGain_WA }
      GetBestVarAge CONF NetGain_WA
     IfRandLt 0.3
      IfRandLt 0.3 { GetBestVarAge RBC-1 NetGain_WA }
       GetBestVar RBC_WA-1 NetGain_WA
      GetBestVar CONF NetGain_WA
 IfVarCond = NegGain 0
  IfVarCond >= SubNetGain 0 { PickRandomVar RBC-1 }
   IfVarCond > SubPosGain_WA 0
    GetBestVar2 RBC-1 NetGain SubPosGain_WA
    GetBestVarSnd RBC-1 NegGain_WA
  IfRandLt 0.9 { GetBestVar2 CONF NetGain NetGain_WA }
   IfIsNull { PickRandomM DecrVars }
    IfVarCompare = SubNegGain { GetBestVar CONF NetGain_WA }
     GetBestVarAge CONF NetGain_WA
```

(b) Part B of the GP-B-5 heuristic.

Figure B.3: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let C =
IfVarCond >= NegGain_WA 0
 IfIsNull { GetBestVarM DecrVars NetGain_WA }
  GetOldestVar
   IfIsNull { GetBestVarM DecrVars_WA NetGain }
    IfVarCompare >= SubPosGain
     GetBestVar2 CONF NegGain_WA SubNetGain
     GetBestVarAge RBC-1 NetGain_WA
   UpdatePAWS
    IfVarCond >= SubPosGain_WA 0
     IfVarCond >= NetGain_WA 0
      IfIsNull { GetBestVarM DecrVars_WA NetGain_WA }
       IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
        GetOldestVar
         IfIsNull { PickRandomM DecrVars }
          IfVarCompare >= SubPosGain
           IfIsNull { GetBestVarM DecrVars_WA NegGain }
            PickRandomVar RBC-0
           B
         IfIsNull { GetBestVarM DecrVars SubPosGain }
          GetBestVarAge RBC_WA-1 NetGain_WA
      IfVarCompare >= SubNegGain
       IfIsNull { PickRandomM DecrVars }
        GetBestVarAge RBC-1 NetGain_WA
       IfVarCond > SubPosGain_WA 0
        IfIsNull { GetBestVarM DecrVars_WA NetGain_WA }
         GetBestVarSnd RBC-0 SubNetGain_WA
        PickRandomVar CONF
     GetOldestVar { GetBestVar2 RBC_WA-1 SubPosGain SubNegGain }
      GetBestVar2 CONF NegGain_WA NegGain
 IfVarCompare >= SubNegGain
  IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
   GetBestVarAge RBC-1 NetGain_WA
  IfVarCond > SubPosGain 0
   IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
    GetBestVarSnd CONF SubNetGain_WA
   PickRandomVar CONF
```

(a) Part C of the GP-B-5 heuristic.

Figure B.4: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let D =
IfVarCond = NegGain 0
 IfVarCond > SubNetGain 0 { PickRandomVar RBC-1 }
  IfVarCond > SubPosGain_WA 0
   GetBestVar2 RBC-1 NetGain SubPosGain_WA
   GetBestVarSnd CONF SubNegGain_WA
 IfRandLt 0.9 { GetBestVar2 CONF NetGain NetGain_WA }
  IfIsNull { PickRandomM DecrVars }
   IfVarCompare = SubNegGain { GetBestVar CONF NetGain_WA }
    GetOldestVar
     IfIsNull { GetBestVarAgeM DecrVars NegGain_WA }
      IfRandLt Adapt
       IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
        IfVarCond = SubPosGain_WA 0
         GetBestVarAge CONF NetGain_WA
         GetBestVarAge RBC-1 SubNetGain
       IfIsNull { GetBestVarM DecrVars_WA NegGain_WA }
        IfVarCompare >= SubPosGain
         IfVarCond = NetGain_WA 0
          GetBestVar2 RBC-0 SubNetGain_WA NegGain
          IfVarCond > PosGain 0
           IfIsNull { GetBestVarAgeM DecrVars_WA NetGain_WA }
            WeightedVarPick RBC-1
             { ExponentFunction 3.6 SubPosGain } EndList
           IfIsNull { GetBestVarAgeM DecrVars SubNetGain_WA }
            IfRandLt Adapt { PickOldest RBC_WA-1 }
             GetOldestVar
              IfIsNull { GetBestVarAgeM DecrVars_WA NetGain_WA }
               GetBestVar RBC_WA-1 NetGain_WA
              IfVarCond = SubNetGain 0
               GetBestVarAge CONF NetGain_WA
               GetBestVarAge CONF NetGain_WA
         GetBestVarAge RBC-1 SubNegGain_WA
     GetOldestVar { GetBestVarSnd CONF NegGain }
      GetBestVarAge RBC_WA-1 NetGain_WA
```

(a) Part D of the GP-B-5 heuristic.

Figure B.5: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let E =
IfVarCompare >= NegGain
 IfVarCond <= SubPosGain_WA 0 { GetBestVar CONF NetGain }
  GetBestVarSnd CONF NegGain_WA
 IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
  IfVarCompare >= NetGain_WA
   IfIsNull { GetBestVarM DecrVars SubPosGain }
    GetBestVarAge RBC_WA-0 NegGain_WA
   IfVarCond = SubNetGain 0
    GetBestVar2 CONF SubNetGain_WA NegGain
    IfVarCond >= SubPosGain_WA 0
     IfVarCond >= NegGain_WA 0
      IfIsNull { GetBestVarM DecrVars NetGain_WA }
       GetOldestVar
        IfIsNull { GetBestVarM DecrVars_WA NetGain }
         IfVarCompare >= NegGain
          GetBestVar2 CONF NegGain_WA SubNetGain
          GetBestVarAge RBC-1 NetGain_WA
        IfVarCond >= SubPosGain_WA 0
         IfVarCond >= NegGain_WA 0
          IfIsNull { GetBestVarM DecrVars_WA NetGain_WA } A
          IfVarCompare >= SubNegGain
           IfIsNull { PickRandomM DecrVars }
            GetBestVarAge RBC-1 NetGain_WA
           IfVarCond > SubPosGain_WA 0
            IfIsNull { GetBestVarM DecrVars_WA NetGain_WA }
              GetBestVarSnd RBC-0 SubNetGain_WA
            PickRandomVar CONF
         GetOldestVar
          GetBestVar2 RBC_WA-1 SubPosGain SubNegGain
          GetBestVar2 CONF NegGain_WA SubNetGain
      D
     GetOldestVar { GetBestVarAge CONF NetGain_WA }
      IfVarCond >= SubPosGain_WA 0
       GetBestVar2 CONF SubPosGain NetGain_WA
       GetBestVarAge CONF NetGain_WA
```

(a) Part E of the GP-B-5 heuristic.

Figure B.6: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let F =
IfVarCond > PosGain 0
 IfVarCond <= PosGain 0 { GetBestVarSnd CONF NegGain_WA }
  IfVarCompare = PosGain_WA
   GetOldestVar { GetBestVar2 CONF NetGain_WA SubNegGain_WA }
    IfVarCond >= PosGain 0
     IfVarCond >= NetGain 0 { PickRandomVar RBC-1 }
      IfVarCompare = SubNegGain { GetBestVar CONF NetGain_WA }
       GetBestVarAge CONF NetGain_WA
     E
   IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
    IfVarCond > SubPosGain_WA 0
     GetOldestVar
      IfVarCond >= SubPosGain_WA 0
       IfVarCond >= NegGain_WA 0
        IfIsNull { GetBestVarM DecrVars NetGain_WA }
         GetOldestVar
          IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
           GetBestVarSnd RBC-0 SubNetGain_WA
          GetOldestVar
           GetBestVar2 CONF SubNetGain_WA NegGain
           IfIsNull { GetBestVarM DecrVars SubPosGain }
            GetBestVarAge RBC_WA-1 NetGain_WA
        IfVarCompare >= SubNegGain
         IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
          GetBestVarAge RBC-1 NetGain_WA
         IfVarCond > SubPosGain 0
          IfIsNull { GetBestVarAgeM DecrVars SubNegGain_WA }
           GetBestVarSnd CONF SubNetGain_WA
          PickRandomVar CONF
       GetOldestVar { GetBestVarAge CONF NetGain_WA }
        IfVarCond >= SubPosGain_WA 0
         GetBestVar2 CONF SubNegGain NetGain_WA
         GetBestVarAge CONF NetGain_WA
      GetBestVar2 CONF NetGain_WA NegGain
     GetBestVarSnd RBC-1 NegGain_WA
 GetBestVarAge RBC_WA-0 NetGain_WA
```

(a) Part F of the GP-B-5 heuristic.

Figure B.7: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let G =
IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
 IfIsNull { GetBestVarAgeM DecrVars } SubNegGain_WA
  IfRandLt Adapt { PickOldest RBC_WA-1 }
   IfRandLt Adapt F
    IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
     IfVarCompare >= SubPosGain
      IfVarCompare >= NetGain_WA
       GetOldestVar
        IfIsNull { PickRandomM DecrVars }
         IfRandLt Adapt
          IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
           IfVarCond = SubPosGain_WA 0
            GetBestVarAge CONF NetGain_WA
            GetBestVarAge RBC-1 SubNetGain
          IfIsNull { GetBestVarM DecrVars_WA NetGain }
           IfVarCompare >= SubPosGain
            IfVarCond = SubPosGain 0
             GetBestVar2 RBC-0 SubNetGain_WA NegGain
             IfVarCond > PosGain 0
              IfIsNull { GetBestVarAgeM DecrVars_WA NetGain_WA }
               WeightedVarPick RBC-1
                { ExponentFunction 3.6 SubPosGain } EndList
              IfIsNull { GetBestVarAgeM DecrVars SubNetGain_WA }
               IfRandLt Adapt { PickOldest RBC-1 }
                GetOldestVar
                 IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
                  GetBestVar RBC_WA-1 NetGain_WA
                 IfVarCond = SubNetGain 0
                  GetBestVarAge RBC_WA-1 NetGain_WA
                  GetBestVarAge CONF NetGain_WA
            GetBestVarAge RBC-1 SubNegGain_WA
        GetBestVar2 RBC-0 SubNetGain_WA NegGain
       GetBestVar CONF NetGain_WA
      IfVarCond = NetGain_WA 0 { GetBestVar RBC_WA-0 NegGain_WA }
       GetBestVarAge RBC-0 PosGain
```

(a) Part G of the GP-B-5 heuristic.

Figure B.8: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let H =
IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
 IfRandLt Adapt
  IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
   IfVarCond = PosGain 0
    IfVarCond >= SubPosGain_WA 0
     IfVarCond >= NegGain_WA 0
      IfVarCompare >= SubNegGain
       IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
        GetBestVarAge RBC-1 NetGain_WA
       IfVarCond > SubPosGain 0
        IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
         GetBestVarSnd CONF SubNetGain_WA
        PickRandomVar CONF
      IfVarCompare >= SubNegGain
       IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
        GetBestVarAge RBC-1 NetGain_WA
       IfVarCond > SubPosGain_WA 0
        IfIsNull { GetBestVarAgeM DecrVars PosGain }
         GetBestVarSnd RBC-0 SubNetGain_WA
        IfVarCond > SubNetGain 0 { PickRandomVar RBC-1 }
         IfVarCond > SubPosGain_WA 0
          GetBestVar2 RBC-1 NetGain SubPosGain_WA
          IfVarCompare >= SubNegGain
           IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
            PickOldest RBC_WA-1
           IfVarCond > SubPosGain_WA 0 G
            PickRandomVar CONF
     GetOldestVar { GetBestVarAge CONF NetGain_WA }
      IfVarCond >= SubPosGain_WA 0 { PickRandomVar CONF }
       GetBestVarAge CONF NetGain_WA
    GetBestVarAge RBC-1 SubNegGain_WA
  IfIsNull { GetBestVarM DecrVars_WA NetGain }
   IfVarCompare >= SubPosGain
    GetBestVar2 RBC-1 NegGain_WA PosGain
    GetBestVarAge RBC-0 PosGain_WA
```

(a) Part H of the GP-B-5 heuristic.

Figure B.9: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let I =
IfIsNull { GetBestVarM DecrVars_WA NetGain_WA }
 IfRandLt 0.9 { GetBestVar2 CONF NetGain_WA SubNegGain_WA }
  IfVarCompare >= SubPosGain
   IfRandLt 0.3
    IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
     GetBestVarAge RBC_WA-1 NetGain_WA
    IfRandLt 0.9 { GetBestVarAge CONF NetGain_WA }
     GetBestVarSnd RBC_WA-0 SubNetGain_WA
   IfIsNull
    GetBestVarAgeM NegGain
     Filter > SubPosGain 0 RBC_WA-1
    IfVarCond = NetGain_WA 0
     GetBestVar2 RBC-0 SubNetGain_WA NegGain
     IfVarCond > PosGain 0
      IfIsNull { GetBestVarAgeM DecrVars_WA NetGain_WA }
       WeightedVarPick RBC-1
        { ExponentFunction 3.0 NetGain_WA } EndList
      IfIsNull { GetBestVarAgeM DecrVars SubNetGain_WA }
       IfRandLt Adapt { PickOldest RBC_WA-1 }
        IfRandLt Adapt { GetBestVar RBC_WA-1 NegGain_WA }
         IfVarCompare <= SubNegGain
          GetBestVarAge CONF PosGain
          GetOldestVar
           IfIsNull { GetBestVarM DecrVars NegGain_WA }
            IfVarCond > SubPosGain_WA 0
             IfVarCond <= PosGain 0
              GetBestVarSnd CONF NegGain_WA
              H
             GetBestVarAge RBC_WA-0 NetGain_WA
           IfVarCond >= PosGain 0 { GetBestVar RBC_WA-1 NegGain_WA }
            GetBestVarAge RBC_WA-0 NegGain_WA
```

(a) Part I of the GP-B-5 heuristic.

Figure B.10: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let J =
IfVarCond > PosGain 0
 IfVarCond <= PosGain 0 { GetBestVarSnd CONF NegGain_WA }
  IfVarCompare = PosGain_WA
   GetOldestVar
    IfVarCond >= PosGain 0
     IfVarCond >= PosGain 0 { PickRandomVar RBC-1 }
      PickOldest CONF
     IfVarCompare >= PosGain_WA
      IfVarCond <= SubPosGain_WA 0
       GetBestVar CONF NetGain
       GetBestVarSnd CONF NegGain_WA
      IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
       IfVarCompare >= NetGain_WA
        IfIsNull { GetBestVarM DecrVars SubPosGain }
         GetBestVarAge RBC_WA-0 NegGain_WA
        IfVarCond = SubNetGain 0
         GetBestVar2 CONF SubNetGain_WA NegGain
         UpdatePAWS
          IfVarCond >= SubPosGain_WA 0  B
           GetOldestVar { GetBestVarAge CONF NetGain_WA }
            IfVarCond >= SubPosGain_WA 0
             GetBestVar2 CONF SubNegGain NetGain_WA
             GetBestVarAge CONF NetGain_WA
    GetBestVar2 CONF NetGain_WA SubNegGain_WA
   IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
    IfVarCond > NetGain 0
     GetOldestVar
      IfIsNull { PickRandomM DecrVars }
       GetBestVarAge CONF NetGain_WA
      GetBestVar2 CONF NetGain_WA NegGain
     GetBestVarSnd RBC-1 NegGain_WA
 GetBestVarAge RBC_WA-0 NetGain_WA
```

(a) Part J of the GP-B-5 heuristic.

Figure B.11: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
let K =
IfIsNull { PickRandomM DecrVars }
 IfIsNull { GetBestVarAgeM DecrVars SubPosGain }
  IfRandLt Adapt { PickOldest RBC_WA-1 }
   IfRandLt Adapt J
    IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
     IfVarCompare >= SubPosGain
      IfVarCompare >= NetGain_WA
       GetOldestVar
        IfIsNull { GetBestVarAgeM DecrVars NegGain_WA }
         IfRandLt Adapt
          IfIsNull { GetBestVarM DecrVars SubNegGain_WA }
           IfVarCond = SubPosGain_WA 0
            GetBestVarAge CONF NetGain_WA
            GetBestVarAge RBC-1 PosGain
          IfIsNull { GetBestVarM DecrVars_WA NetGain }
           IfVarCompare >= SubPosGain
            IfVarCond = NetGain_WA 0
             GetBestVar2 RBC-0 SubNetGain_WA NegGain
             IfVarCond > PosGain 0
              GetBestVarAge RBC-0 NegGain_WA
              IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
               IfRandLt Adapt { PickOldest RBC_WA-1 }
                GetOldestVar
                 IfIsNull
                  GetBestVarAgeM DecrVars_WA SubNegGain_WA
                  GetBestVar RBC_WA-1 NetGain_WA
                 IfVarCond = SubNetGain 0
                  GetBestVarAge RBC_WA-1 NetGain_WA
                  GetBestVarAge CONF NetGain_WA
            GetBestVarAge RBC-1 SubNegGain_WA
        GetOldestVar { GetBestVarSnd CONF NegGain }
         GetBestVarAge RBC_WA-1 NetGain_WA
       GetBestVar CONF NetGain_WA
      IfVarCond = NetGain_WA 0 { GetBestVar RBC_WA-0 SubNegGain }
       GetBestVarAge RBC-0 PosGain
```

(a) Part K of the GP-B-5 heuristic.

Figure B.12: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)

```
UpdatePAWS
 GetOldestVar
  IfIsNull { PickRandomM DecrVars }
   IfVarCompare >= SubPosGain
    IfIsNull { GetBestVarM DecrVars_WA NegGain_WA }
     IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
      IfRandLt 0.9
       GetOldestVar I
        IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
         IfVarCompare >= SubPosGain
          IfVarCompare >= NetGain_WA
           GetBestVarAge CONF NetGain_WA
           GetBestVar CONF NetGain_WA
          IfVarCond = NegGain_WA 0
           GetBestVar2 CONF NegGain_WA NegGain
           IfIsNull { GetBestVarAgeM DecrVars NetGain_WA }
            GetBestVar CONF PosGain
       GetBestVar CONF NetGain_WA
    IfVarCond = SubNetGain 0
     GetBestVar2 CONF  SubNetGain_WA NegGain
     IfVarCond = NegGain 0
      IfVarCond > SubNetGain 0 { PickRandomVar RBC-1 }
       IfVarCond > SubPosGain_WA 0
        GetBestVar2 RBC-1 NetGain SubPosGain_WA
        IfVarCompare >= SubNegGain
         IfIsNull { GetBestVarAgeM DecrVars_WA SubPosGain_WA }
          PickOldest RBC_WA-1
         IfVarCond > SubPosGain_WA 0 K { PickRandomVar CONF }
      IfRandLt 0.9 { GetBestVar2 CONF NetGain NetGain_WA }
       IfIsNull { PickRandomM DecrVars }
        IfVarCompare = SubNegGain
         GetBestVar CONF NetGain_WA
         GetBestVarAge CONF NetGain_WA
  IfIsNull { GetBestVarM DecrVars SubPosGain }
   GetBestVarAge RBC_WA-1 NetGain_WA
```

(a) Final part of the GP-B-5 heuristic.

Figure B.13: The GP-B-5 heuristic. Fitness value of 65.0. (Continued)