Algorithms and Data Structures for Three-Dimensional Packing

Sam D. Allen, BSc (Hons)

Thesis submitted to the University of Nottingham for the degree of

Doctor of Philosophy (PhD)

July 2011

Abstract

Cutting and packing problems are increasingly prevalent in industry. A well utilised freight vehicle will save a business money when delivering goods, as well as reducing the environmental impact, when compared to sending out two lesser-utilised freight vehicles. A cutting machine that generates less wasted material will have a similar effect. Industry reliance on automating these processes and improving productivity is increasing year-on-year.

This thesis presents a number of methods for generating high quality solutions for these cutting and packing challenges. It does so in a number of ways. A fast, efficient framework for heuristically generating solutions to large problems is presented, and a method of incrementally improving these solutions over time is implemented and shown to produce even higher packing utilisations. The results from these findings provide the best known results for 28 out of 35 problems from the literature. This framework is analysed and its effectiveness shown over a number of datasets, along with a discussion of its theoretical suitability for higher-dimensional packing problems. A way of automatically generating new heuristics for this framework that can be problem specific, and therefore highly tuned to a given dataset, is then demonstrated and shown to perform well when compared to the expert-designed packing heuristics. Finally some mathematical models which can guarantee the optimality of packings for small datasets are given, and the (in)effectiveness of these techniques discussed. The models are then strengthened and a novel model presented which can handle much larger problems under certain conditions. The thesis finishes with a discussion about the applicability of the different approaches taken, to the real-world problems that motivate them.

To my parents, Michael and Loraine.

Acknowledgements

Although one of the University of Nottingham's *Criteria for award of PhD and other qualifications at Doctoral Level* is that "[the] thesis must be the result of the candidate's own work," I feel obliged to admit that this thesis would not have been possible without the work of many people other than me. Hopefully the university won't mind too much though.

I am indebted to many of the ASAP research group, past and present — academic members, visitors, and admin team alike. Without the encouragement and support from a lot of you I would have sacked it all in to make my fortune in industry long ago. I am particularly grateful to (in no particular order) Sven Groenemeyer, Jakub Mareček, Jason Atkin, Andrew Parkes, and Núria Macià for their invaluable contributions to this thesis, and my PhD output as a whole — cheers guys, you've seen all the work in here already so you can stop reading now if you like.

I also wish to thank my supervisors Professors Edmund Burke and Graham Kendall, for providing me with the funding, flexibility and freedom to pursue my own research goals, and to even reach some of them.

Finally I am forever grateful for the love and support from my family, throughout both the PhD process, and life in general.

Contents

List of Figures

List of Tables

1

Introduction

I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

The Psychology of Science: A Reconnaissance
Abraham Maslow

1.1 Background and Motivation

The phrase 'cutting and packing' envelops a multitude of tangible problems in every day life. These problems cover such questions as "will all of this shopping fit into my car?" to "how can I best spread these audio tracks across the minimum number of compilation CDs?" and "can we finish this building project any faster?"

In an industrial setting there is an ever-growing need for efficiency of material usage and waste reduction. An increase in public awareness of environmental issues, and the pressure that is hence put on large companies to become 'greener' drives businesses to try and improve their efficiency further; fewer, better loaded, delivery vehicles on the roads not only reduces direct costs for a company, but can in some cases allow access to financial incentives from a government. It can be easily observed that in recent years the reliance on automation has increased, requiring the development of fast and efficient cutting and packing systems in order to semi-automate cutting and packing procedures and keep up with this demand. In some cases (such as Computer Numerical Control machines or robotic packing in factories) these procedures can be fully automated by a computer with the appropriate machinery. It has been suggested that automated approaches can be as efficient, or in some cases more efficient, than

a human attempting the same problem [Rob84] and usually in a substantially shorter amount of time [HRK96].

It is obvious, therefore, that improvements in automated cutting and packing procedures can lead to big savings for both companies and the environment, and are certainly a worthwhile research direction.

This has been realised by many researchers, with several different approaches being taken for solving cutting and packing problems, with only some inter-method comparison, little justification as to when to use which, and only vague insight into how well each of the approaches may fare on any given problem.

1.2 Aims and Scope

The purpose of this dissertation is to provide a contribution to cutting and packing research through improvements in several solution methodologies and some indication of the strengths and weaknesses of each. Firstly, a generalised constructive packing framework upon which to base the subsequent work is given. Secondly, some fast single-pass constructive heuristics and metaheuristics which generate good solutions for problems of up to 1000 boxes are presented, and which fit into the proposed framework. Thirdly, an analysis of the framework and an insight into the complexity of higher-dimensional packing problems in general is offered. The fourth aim is to improve the current methodology for generating packing heuristics, as well as providing methods for improving other heuristics generated through genetic programming techniques. Finally, some extensions and new models for solving smaller packing problems with guaranteed optimality are given, with the aim to give some insight into the time and quality trade-offs between these methods and the other non-exact methods presented.

1.3 Contributions of this Thesis

- Chapter 3
  - A generalised constructive three-dimensional packing framework (the 3BF framework.)
  - A set of fast and highly effective packing heuristics for the framework.
  - An integrated heuristic-metaheuristic approach to solving three-dimensional packing problems.
  - The best known results for a number of standard benchmark instances.

- Chapter 4
  - A bound on the number of gaps at any point in a packing in D dimensions.
  - A runtime performance analysis of the 3BF framework.
  - An abstract data type (ADT) for representing D-dimensional packings (Skyline.)
  - A theoretical and empirical analysis of approaches from the literature implemented for the Skyline ADT.
- Chapter 5
  - Improved methods for automatically generating heuristics for the 3BF framework, one of which that is applicable to genetic programming approaches in general.
  - The best known results for automatically generating three-dimensional packing heuristics.
- Chapter 6
  - Improvements to a state-of-the-art integer linear programming model for three-dimensional packing.
  - The first detailed experimental results from the state-of-the-art model and the highlighting of its shortcomings.
  - A new space-indexed model for three-dimensional packing with extremely tight linear programming relaxations and comprehensive experimental results.
  - Experimental/anecdotal insight into when to use exact methods and when to rely on heuristics for three-dimensional problems, based on the current state-of-the-art.

1.3.1 Publications Produced and Talks Given based on the Work in this Thesis

The following works have been produced during the creation of this thesis. Though material from all of them has been assimilated into the thesis as a whole, they can be assigned as belonging particularly to the following specific chapters.

Journal Publications

- Chapter 3
  - S. D. Allen, E. K. Burke, and G. Kendall. A hybrid placement strategy for the three-dimensional strip packing problem. *European Journal of Operational Research*, 209(3):219 – 227, 2011.
- Chapter 4
  - S. D. Allen and E. K. Burke. A data structure for higher-dimensional rectilinear packing. *INFORMS J. on Computing*, to appear.
- Chapter 5
  - S. D. Allen and E. K. Burke. Automatic generation of three-dimensional packing heuristics. Technical report, University of Nottingham, 2010, submitted for review.
- Chapter 6
  - S. D. Allen, E. K. Burke, and J. Mareček. A space-indexed formulation of packing boxes into a larger box. Technical report, University of Nottingham, 2011, submitted for review.

Conference Publications and Talks

- Chapter 3
  - S. Allen, E. K. Burke, M. Hyde, and G. Kendall. Evolving reusable 3D packing heuristics with genetic programming. In *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 931–938, New York, NY, USA, 2009. ACM.
  - S. D. Allen. A heuristic/metaheuristic approach to the three-dimensional strip-packing problem. Talk given at the Student Conference on Operational Research, SCOR 2010, March 27–29, Lancaster, UK, March 27–29 2009.
- Chapter 4
  - S. D. Allen. Data structures for higher-dimensional rectilinear packing. Talk given at the Student Conference on Operational Research, SCOR 2010, April 9–11, Nottingham, UK, April 2010.
- Chapter 6
  - S. D. Allen, E. K. Burke, G. Kendall, and S. Groenemeyer. Strip packing: What can we learn from project scheduling? Talk given at the ALIO-INFORMS Joint International Meeting 2010, June 6–9, Buenos Aires, Argentina, June 2010.

Technical Reports and Manuals

- Chapter 3
  - S. D. Allen. *CrateViewer: visualising {one, two, three}-dimensional packings.* University of Nottingham, March 2010, manual.

1.4 Structure of the Thesis

The main body of the thesis is separated into 6 chapters. Chapter 2 provides a literature review and description of the previous work undertaken in the field as well as some formalised definitions of the problems that we will be solving. Chapter 3 introduces the 3BF constructive framework and gives some heuristics and experimental results. Chapter 4 analyses the 3BF framework and provides some insight into data structures and efficiency of packing in higher dimensions. Methods of automatically generating heuristics for the 3BF framework are given in chapter 5. Some computational experience and a new integer linear programming model for exactly solving three-dimensional problems is given in chapter 6. Finally the thesis is concluded with some discussion in chapter 7.

1.4.1 Notation Used in this Thesis

The general notation used in this thesis is given in Table 1.1.

Symbol	Description	Meaning
X	Uppercase Latin character	A set or vector
x	Lowercase Latin character	Member of a set or an index
$X_{[i]}^{[j]}$	Uppercase Latin character with optional indices	A pre-defined/computed constant
$x_i^{[j]}$	Lowercase Latin character with indices	A variable; continuous or integer
$\chi_i^{[j]}$	Lowercase Greek letter with indices	A binary decision variable

Table 1.1: Notation used in this thesis.

2

Background and Related Work

> One can think effectively only when one is willing to endure suspense and to undergo the trouble of searching.
>
> *How We Think: A restatement of the relation of reflective thinking to the educative process*
> John Dewey

Summary

This chapter provides an introduction to the background of the thesis, through a review of the relevant literature in the areas of cutting and packing as a whole, and the optimisation techniques previously applied to the problems. This should provide an insight into the motivations for the techniques applied to the problems posed in later chapters. Specific results and methodology from the literature will be provided, as and when needed, at the appropriate points in later chapters.

2.1 An Overview

Research into cutting and packing has received significant research interest in recent years, with several well characterised approaches being employed to differing levels of success. Cutting and packing, in the purest academic sense, underlies a number of common problems, from the one-dimensional knapsack problem which is an integral part of many other difficult problem formulations to synonymous models of scheduling problems (resource constrained

project scheduling [BDM+99, Har00, NI02, BL03, ABKG10], parallel processor scheduling [CGJ78, GJ77a, GJ77b, CMV94] etc.)

The terms cutting and packing are often used interchangeably in the academic literature, as the problems are usually the same in a mathematical sense, but with slight variations in terminology. There are, however, many different 'flavours' of cutting and packing problems which we cover in the following sections. The recurring theme throughout all packing problems (and, in general, scheduling problems) is the assignment of 'items' to 'bins' or 'locations.'

For a comprehensive classification of different packing problems, including three- and higher-dimensional problems see the typologies of Dychkhoff and Wäscher, [Dyc90, WHS07]. Other general surveys on the academic literature related to cutting and packing problems can be found in [SP92, Dyc92, DD92, CFC94, HT01a, LMV02, CSY02].

2.2 One-dimensional Cutting and Packing

One-dimensional (or uni-dimensional) packing problems are incredibly prevalent in, and indeed can be found at the core of, many operational research (OR) problems. They are often used as a staple in teaching OR and optimisation to undergraduate students due to their inherent conceptual simplicity. One-dimensional packing problems generally fall into two main categories, the bin packing problem and the knapsack problem.

The one-dimensional problem is analogous to some real-world problems in industry, for example cutting lengths of cabling or metal beams [SSW94] whilst minimising trim wastage. Similarly, maximisation of the utilisation of one-dimensional bins or knapsacks can be found in areas such as maximising profit in advertisement scheduling [AGM02] and creating one-way cryptographic functions [MH78].

An excellent coverage of one-dimensional problems can be found in the book by Martello and Toth [MT90]. E. G. Coffman has arguably produced some of the most influential literature in the area of one-dimensional cutting and packing, and has published a number of papers and books extensively covering the topic, for example see [CLT88, CGJT80, CGJ84, CGJ87, CL91, CJGJ96]. These papers and surveys include detailed analysis of different approaches to solving the problems.

Although the problems can be elegantly defined and are conceptually simplistic, the algorithms, representations and techniques for solving them have little direct bearing on the two- or higher-dimensional packing problems tackled in this thesis, except, in some cases, when one-dimensional problems act as additional constraints or sub-problems (commonly the zero-one knapsack problem).

2.2.1 The Bin Packing Problem

The description of the bin packing problem is that, given a number of one-dimensional items with given sizes (with the size representing, for example, the required length of a section of cable or an amount of time in seconds that an audio track lasts for) and an unlimited supply of 'bins' or containers with fixed capacity (representing, using the previous examples, the length of a roll of cable or a blank audio CD), find the fewest number of bins required to assign all items to a bin such that the sum of the items' sizes does not exceed the capacity of the bin.

This can be formalised as: given N items with non-negative sizes L_i, $i \in \{1, 2, \ldots, N\}$, bin placement indicator $\chi_i^b \in \{0, 1\}$ which is 1 if piece i has been stored in bin b and 0 otherwise, and bin capacity C:

$$\text{Minimise} \qquad b_* \tag{2.1}$$

$$\text{s.t.}$$

$$\sum_{j=1}^{b_*} \chi_i^j \;=\; 1 \quad \forall i \in \{1, 2, \ldots, N\} \tag{2.2}$$

$$\sum_{i} \chi_i^j L_i \;\leq\; C \quad \forall j \in \{1, 2, \ldots, b_*\} \tag{2.3}$$

Constraint 2.2 denotes that each piece must be packed into exactly one of the bins, and constraint 2.3 ensures that the capacity of each bin is not exceeded.

The bin packing problem is well known to be strongly $\mathcal{NP}$-complete [GJ79].

2.2.2 The Knapsack Problem

The knapsack problem is similar to the bin packing problem and also has several variants. In this thesis the problem will be defined as the most common variant, which is the zero-one knapsack problem. Instead of minimising the number of bins used when packing all pieces, the objective is to maximise the summed value of a single available bin, where each piece is assigned a value as well as a size (with the value optionally being independent of the size of the piece). The formulation follows: given N items with non-negative sizes L_i and non-negative values (knapsack weights), W_i, $i \in \{1, 2, \ldots, N\}$, bin placement indicator $\chi_i \in \{0, 1\}$ which is 1 if piece i is to be placed in the bin and 0 otherwise, and bin capacity C:

$$\text{Maximise} \qquad \sum_{i=1}^{N} W_i \chi_i \tag{2.4}$$

$$\text{s.t.}$$

$$\sum_{i=1}^{N} L_i \chi_i \;\leq\; C \tag{2.5}$$

Constraint 2.5 ensures that the capacity of the bin is not exceeded.

The zero-one knapsack problem is only weakly $\mathcal{NP}$-complete, as a pseudo-polynomial time result (in this case based on the capacity, C) can be found by dynamic programming [MT90].

2.3 Two-dimensional Cutting and Packing

One of the most common forms of research in cutting and packing focuses on two-dimensional packing problems, which have many real-life practical applications — for example paper [JRZ97, HWPS98, AB98, Hah68, McD99, MS02], metal [MR06, NT00, KD02], glass [DG74, PRK04] or other sheet material cutting (see, for example, [CCT+03, CPM00, MG98]). Other, more theoretical ideas, can also be modelled as a two-dimensional packing problem such as multi-processor scheduling [CGJ78]. Dowsland and Dowsland [DD92] give an overview of some industrial applications of two-dimensional cutting and packing.

Two-dimensional problems have a much clearer mapping to higher-dimensional problems than the one-dimensional problem as the interaction of the dimensions when placing items only exists when there is more than one dimension being taken into consideration (e.g. overlapping may occur — as explained in later sections – but this cannot happen in the one-dimensional problem as the concept of overlapping constraints does not exist for it). In most cases two-dimensional problems can be treated in the same way as a higher-dimensional problem (in fact, higher-dimensional problems can be mapped to a two-dimensional problem simply by setting all but two of their dimensions to be some fixed ϵ — of course this is true of the mapping of higher-dimensional problems to any lower-dimensional problem). We leave the formalisation of these problems until section 2.4.

As with one-dimensional packing, there are many variants of the two-dimensional packing problems. A rigorous classification of these can be found in [WHS07]. For a detailed coverage of two-dimensional packing problems see [LMM02], which details exact approaches as well as heuristic and approximation results for several variants of the two-dimensional packing problems. Some lower bounds are also given.

One example of the special constraints in the two-dimensional packing literature can be found in [MM68, CJL89, EG75] where the case of only packing squares into a square or rectangular container is covered. Another constraint is known as the 'guillotine-cutting constraint' in which only straight line cuts across the full length of the bin (or sub-bin, recursively) are allowed, this is a standard constraint in industrial cutting applications in which a machine is to cut strips of material automatically. Examples of work with this constraint can be found in [TTS94, CH95, LMV99, LLM02]. Other constraints in two-dimensional packing involve limiting the rotation of pieces to be packed/cut. Sometimes, depending on the real-world problem it may suffice to permit any rotation (e.g. sheet steel cutting [VFFS89]), whereas, for example, in patterned

fabric cutting certain rotations may be restricted, or any rotation prohibited. Generally, feasible rotations are fixed to be orthogonal, that is, only rotations of 90 degrees are valid.

Several methods of solving the two-dimensional problem have been tried. These methods are explained in section 2.5.

Exact methods for solving two-dimensional problems have had limited effect due to the strong combinatorial effect on the solution space, especially when admitting rotations of pieces. Work on exact methods can be found in [Bea85, CH95, Hif01, CJCM08, KIN+09, BKCS10, SAdC10]. Only relatively small problem instances (with up to 100 pieces) can be solved optimally using exact methods, though progress is being made through improved bounding and the greater processing power of modern computers.

Due to the complexity of solving two-dimensional problems exactly, heuristic and metaheuristic approaches have generally been employed with reasonable success. Some constructive algorithms include [AO79, Cha83, CG84, CS90, CS93, BKW04, WZC09, EP09]. These tend to get fairly strong results in a very short amount of time but are not guaranteed to be optimal and generally terminate after a single, deterministic, run.

For medium – large sized instances metaheuristics tend to be a good trade-off between heuristic algorithms and exact methods — given a reasonable amount of time they tend to outperform simple constructive heuristics, and can cope with much larger problem instances than exact methods. See [HT01a, RBN09, BKW09] for examples of metaheuristic approaches to two-dimensional packing.

2.4 Three-/Higher-dimensional Cutting and Packing

There are other classes of cutting and packing problems which cannot however be solved using methods applied to the two-dimensional problem, such as block cutting (e.g. foam, wood, marble, etc. [JHD97]), load planning in freight and shipping (delivery vehicles, air cargo, etc. [Fas04, LM80]) or multi-dimensional limited resource scheduling [ABKG10]. This is where advances in three-dimensional packing can create big savings in profit and reductions in wastage to production and logistics companies. **This class of problems is the focus of the thesis**.

Even though two-dimensional strip packing and its variants have been studied since the 1960s it is only over recent years that the amount of research interest concerning three-dimensional packing problems has become prevalent, for example [MPV00, Bis06, CPT08, EP09]. There are several optimisation problems which fall under the umbrella term of three-dimensional packing. These are, again, characterised in the typology of Wäscher [WHS07].

The two main variants of the three-dimensional packing problem are the Container Loading Problem (CLP) and the Strip Packing Problem (SPP). Both of these problems are formalised in the following sections and are $\mathcal{NP}$-Hard, even to approximate [CC09].

2.4.1 The Container Loading Problem

A large D-dimensional (typically within this thesis, three-dimensional, $D = 3$) parallelepiped C is given, with dimensions $L_0^1, L_0^2, \ldots, L_0^D \in \mathbb{R}_+$ indicating its width, height and length respectively in three dimensions. A set B of N D-dimensional parallelepipeds to pack are also given with dimensions $L_i^1, L_i^2, \ldots, L_i^D \in \mathbb{R}_+$ again indicating width, height and length in three dimensions, $x_i^1, x_i^2, \ldots, x_i^D \in \mathbb{R}_{\geq 0}$ indicating the relative coordinates of their leftmost/lowest/deepest etc. corner (i.e. that which is closest to the origin) and $\chi_i \in \{0, 1\}$ indicating whether box i is packed or not. In all cases $i \in \{1, \ldots, N\}$. The objective is to position as many of the boxes within the container in order to maximise the volume utilisation. The CLP is effectively the higher-dimensional equivalent of the uni-dimensional knapsack problem.

The following constraints must hold. The interiors of all the boxes should be disjoint (i.e. the non-overlapping constraint). This means that the boxes must be non-overlapping in at least one dimension. The non-overlapping constraint is given as constraint 2.7 in the following model. The domain constraint states that the extremities of all placed boxes must lie within the bounds of the container, i.e. the boxes cannot be positioned such that they are sticking out of the container, as formalised in constraint 2.8. The orthogonality constraint states that all placed boxes be positioned such that each side must be parallel to a side of the container, i.e. if rotations of the boxes are permitted then only those of 90 degrees may be made. In the case that rotations are allowed and a rotation of box i occurs then the order of the L_i^* values are rearranged. The orthogonality constraint is implicitly held in the representation of boxes given here. The full model is as follows:

$$\text{Maximise} \qquad \sum_{i=1}^{N} \chi_i W_i \tag{2.6}$$

s.t.

$$\chi_i \prod_{k=1}^{D} \text{overlap}(k, i, j) = 0 \qquad \forall\ 1 \leq i < j \leq n \tag{2.7}$$

$$\chi_i \sum_{k=1}^{D} \text{uncontained}(k, i) = 0 \qquad \forall\ 1 \leq i \leq n \tag{2.8}$$

where

$$W_i = \prod_{j=1}^{D} L_i^j \tag{2.9}$$

$$\text{overlap}(k, i, j) = \begin{cases} 1 & \text{if } x_i^k < x_j^k + L_j^k \text{ and } x_j^k < x_i^k + L_i^k \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

$$\text{uncontained}(k, i) = \begin{cases} 1 & \text{if } x_i^k < 0 \text{ or } x_i^k + L_i^k > L_0^k \\ 0 & \text{otherwise.} \end{cases} \tag{2.11}$$

Constraints 2.7 and 2.10 ensure that non of the packed boxes overlap, i.e. for every pair of boxes, at least one dimension does not have an overlap. Constraints 2.8 and 2.11 ensure that each box is placed such that it lies within the bounds of the container. W_i (as defined by constraint 2.9) represents both the volume and the profit of the box, which are synonymous in the CLP.

The CLP has been approached in several different ways throughout the academic literature. Exact methods have been proposed. For example, the mixed integer-linear programming (MIP) model of Chen et al. [CLS95] allows the packing of boxes into multiple containers, though very little has been given in terms of the experimental results — only a maximum 6 boxes were packed using this model. Fasano [Fas99] has also given a MIP model for the container loading problem, which has been analysed and strengthened slightly by Padberg [Pad00]. This has also provided very little computational experience, but with the suggestion that 'around 20' [Pad00] boxes should be solvable using the given model. These models are all based on allowing the MIP solver to fix the continuous variables representing the coordinates of the smaller boxes, and they also introduce some binary decision variables representing precedence constraints of the boxes on a per-axis basis. Other exact methods include graph-theoretical approaches [MA93, LLM02] and tree search routines [FS97].

More extensive work has been done on heuristics for the CLP. The heuristics developed can, in general, be roughly divided into two main classes. The first of these includes the wall building method proposed in [GR80]. This is where solutions are created by building up individual layers along the depth of the container in order to simplify the problem. This approach has also been used in [BD82, HKE89, BM90]. Another approach to the first class of heuristic packings can be seen as stack building, that is, creating vertical stacks of boxes which are then positioned on the base of the container, based on the solution to a two-dimensional packing problem. This approach was first suggested in [GG65] and has also been used in [GB97]. Both of these methods, of course, create 'greedy' packings which may not be optimal, though in practice tend to get fairly good results. These two methods can be treated as one class, a 'layering' approach to packing. This also covers other heuristics that generate packings in layers such as [GMM90, BR95, BG01, Pis02, LRW03].

The second group of approaches to CLP heuristics is that of block arrangement and single pass constructive methods. This method involves placing either a single box at a time or a block of boxes (i.e. a group of boxes which have been joined together and treated as a single unit) within the container until all boxes have been packed, based on some heuristic scoring/choice function. Examples of these can be seen in [Ele02, BGM03, LRY05, CPT08, HH09, WGC$^+$10, HH11].

Metaheuristic approaches (as explained in section 2.5.2 have also been applied to the container loading problem. It has been found, as with the two-dimensional problem, that metaheuristics can tackle medium to large problems (i.e. those with more than around 50 boxes) more effectively than exact methods, and for smaller problems can often solve the problems to optimality

(but, of course, with no guarantees). Some metaheuristic methods used in three-dimensional container loading include [Dow93, GB97, BG01, BGM03, MBG04, MO05, GILM06, HZWC10, FDHI10, Tan10].

2.4.2 The Strip Packing Problem

Three-dimensional strip packing can be defined as the following: given a three-dimensional bin (parallelepiped) of fixed width and height but unconstrained length (the 'container'), and a list of smaller cuboids ('boxes') of given width, length and height, the goal is to pack the boxes into the container, such that all vertices of each box lie within the bounds of the container, minimising the container's resulting length. An obvious constraint is that the packed boxes must not overlap (i.e. their interiors are disjoint). Also the boxes must be packed feasibly — i.e. they must be packed orthogonally (parallel to the axes of the container).

The SPP follows a similar formulation to the CLP, but with some important differences. The SPP assumes a container with one dimension of infinite length, without loss of generality we assume hereafter that this is the Dth dimension (or 'length' in the two- and three-dimensional case). All boxes must be packed in the SPP. The SPP shares the same constraints (domain, non-intersection and orthogonality) as the CLP. The SPP can be formalised as follows:

$$\text{Minimise} \quad s_* \tag{2.12}$$

s.t.

$$s_* \geq x_i^D + L_i^D \quad \forall i \in \{1, 2, \ldots, n\} \tag{2.13}$$

$$L_0^D = \infty \tag{2.14}$$

$$\chi_i = 1 \quad \forall i \in \{1, 2, \ldots, n\} \tag{2.15}$$

$$\chi_i \prod_{k=1}^{D} \text{overlap}(k, i, j) = 0 \quad \forall\ 1 \leq i < j \leq n \tag{2.16}$$

$$\chi_i \sum_{k=1}^{D} \text{uncontained}(k, i) = 0 \quad \forall\ 1 \leq i \leq n \tag{2.17}$$

where

$$\text{overlap}(k, i, j) = \begin{cases} 1 & \text{if } x_i^k < x_j^k + L_j^k \text{ and } x_j^k < x_i^k + L_i^k \\ 0 & \text{otherwise.} \end{cases} \tag{2.18}$$

$$\text{uncontained}(k, i) = \begin{cases} 1 & \text{if } x_i^k < 0 \text{ or } x_i^k + L_i^k > L_0^k \\ 0 & \text{otherwise.} \end{cases} \tag{2.19}$$

s_*, as defined in constraint 2.13, is the furthest point along the Dth axis across all boxes during a packing, and is being minimised in the objective function. Constraint 2.14 allows the container to be as long as needed in order to fit all of the boxes. 2.15 ensures that all boxes are packed.

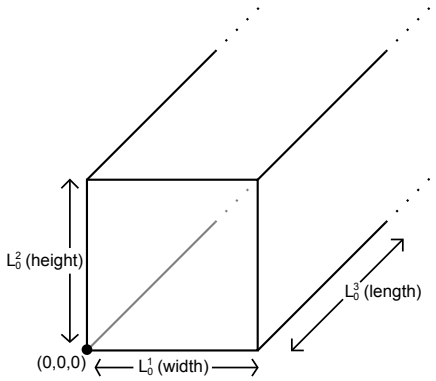

Figure 2.1: A visual representation of a three-dimensional strip packing container.

Constraints 2.16 and 2.17 are as the CLP formulation. Constraint 2.16 ensures that no boxes overlap in all dimensions. Constraint 2.17 forces all boxes to be placed within the limits of the container.

A visual example of a strip packing container can be seen in Figure 2.1. According to the cutting and packing typology of Wäscher [WHS07] this problem is classified as the 3D regular open-dimension-problem (ODP) with one open dimension.

Strip packing in three or higher dimensions has been largely neglected by the academic community. The first approximation of the three-dimensional strip packing problem was not published until 1990 by Li and Cheng [LC90] who released an improved version of the algorithm two years later [LC92]. These papers along with others (e.g. [MW04, JSO06, BHI$^+$07]) focus on algorithmic performance analysis, though there are some with actual implementation details and empirical results; more recently results of metaheuristic approaches appear in [BG99, KI04, BM07, ABK11]. There has been even less work in exact methods for strip packing with, to the author's knowledge, a single MIP model given in [WLGDS10].

2.5 Previous Approaches used in {One–Three}-dimensional Packing

Traditionally in the field of OR and, specifically, combinatorial optimisation, methods for solving problems fall into three main categories. These are heuristics and approximation algorithms (an approach used in chapter 3), iterative improvement/local search based algorithms (which we refer to as metaheuristics, also employed in chapter 3) and exact methods (applied in chapter chapter 6). This thesis encompasses all three of these methods at different points and as such they will be briefly described here for introductory purposes. A detailed coverage of these methods, and more, can be found in [BK05].

2.5.1 Heuristics and Approximation Algorithms

Heuristics, in the domains of computer science and optimisation, refer to algorithms which, regularly in practice, deliver good solutions to a problem based on a 'rule of thumb' or a set of rules derived from some experience or insight into the problem by a practitioner. Although tending to work well in practice, heuristics do not give any guarantee to the quality of solutions generated and can, in theory, deliver indefinitely bad/unknown worst-case solutions. Approximation algorithms, on the other hand, are similar in the respect that they give good solutions in practice, but differ in that they give guaranteed bounds on the worst-case solution to a problem. This property may make them much more attractive than heuristics, but often, in the domain of cutting and packing, the current state-of-the-art does not give particularly good bounds and the bounds themselves are particularly difficult to derive for anything but the most trivial of algorithms. Note that the approximation algorithms tend not to, in general, perform as well as more advanced heuristic algorithms in empirical analysis with experimental testing.

A common form of heuristics and approximation algorithms in cutting and packing is that which we refer to as the class of constructive packing algorithms. The basic premise is that, starting with an empty container/bin, iteratively add items/boxes to the packing until all of them, or as many as possible have been packed. The choice of which items to pack, and whereabouts to place them within the container can be seen as a heuristic choice, usually employing some form of 'fill the space the best' criteria, covered more extensively in chapter 3. Examples of this form of packing include [BKW04, HH09, ABK11]. The term 'bottom-up' is sometimes used to describe the fact that packings of this type are generally constructed following a linear direction, i.e. from the bottom/back of the container, towards the top end. This is not necessarily always the case, for example see [OGF00] which iteratively adds items to a central cluster of previously placed items. This paper also demonstrates a second class of constructive algorithms which place each item in an 'online' way, that is, placing each item in the order that it is passed into the algorithm. Other examples of this include the 'bottom-left' and 'bottom-left-fill' style algorithms which are explained in [Jak96] and [HT99], respectively. The bottom-left algorithm effectively takes the piece and places it in the lowest, leftmost available position in the current packing in which the item will fit. Bottom-left-fill does the same, but attempts to fill any gaps that have been created, i.e. when a smaller piece is packed later than a larger piece. This style of algorithms obviously rely on a good ordering of the pieces being passed into the algorithm, and therefore can be used in decoding a list of items into a solution — an idea that is often used in the metaheuristic packing methods described in this thesis.

2.5.2 Metaheuristics

Another way to create high quality solutions is to take a solution of any quality and iteratively improve it by making perturbations to the representation until a higher quality is obtained.

The methods for creating these perturbations (and therefore new solutions) are termed metaheuristics and are also known as local search, or iterative improvement techniques. The term is prefixed by 'meta' as the techniques can be applied to any problem where perturbations to a solution will create new solutions, without any problem-specific domain information needed. Therefore, metaheuristics are simply a way of describing a local search procedure which can be applied to any similar problem. Metaheuristics are almost always based on some form of randomness ('stochastic optimisation.')

2.5.2.1 Hill Climbing

Possibly the simplest metaheuristic is that of a hill climber. Simply put, a hill climbing algorithm takes the current solution and perturbs it in some defined way at random. Most commonly this involves swapping a number of items in a list, or flipping bits in a bitstring. The hill climber then 'moves' to the new solution (i.e. replaces the current solution with the new one) if the solution is of higher quality, judged by some evaluation function. This is then repeated until some stopping criterion is met, e.g. a time limit, a lack of improvement in solution quality over X steps etc. Sometimes this is adapted by sampling multiple one-step perturbations (known as 'neighbours' — any solutions which can be reached by one step of the defined perturbation method are collectively termed 'the neighbourhood') and moving to the highest quality one.

Hill climbing algorithms often have ways of escaping local optima. This is a condition where no neighbour can be found with a higher solution value than the current solution. One standard way of doing this in hill climbing is the concept of 'random restart,' or randomly picking another solution to start hill climbing from when no more progress can be made from the current point in the search.

2.5.2.2 Simulated Annealing

Simulated annealing is a common form of hill climbing in which worsening moves through a neighbourhood are sometimes accepted based on a given criterion. The name comes from a natural process in metallurgy in which a material is repeatedly heated and cooled in order to reduce defects in its production. The concept of simulated annealing is that there is a predefined cooling schedule, or a function which represents the probability of accepting a worsening move, over time. The function is termed the 'cooling schedule.' Generally the probability decreases over time, though depending on the application it may increase again, or continue to cycle from high to low until the algorithm terminates. The standard function for the acceptance criterion in simulated annealing is $e^{(-\Delta E/T)}$ where ΔE is the difference in solution quality between the current and potential new solution and T represents time, i.e. the number of algorithm iterations or actual 'wall clock' or CPU time taken. The algorithm was first given in [Kir84].

2.5.2.3 Tabu Search

Another common adaptation of hill climbing is that of tabu search. This is simply a form of hill climber with memory. In order to avoid becoming stuck in local optima or following short cyclic paths around the search space, either the last X previously visited states or the last X 'state perturbing actions' (moves) are stored in a 'tabu list' — a queue of fixed length, X — which prohibits them from being visited/executed again, at least within X steps.

A 'state perturbing action' is a specified perturbation to a state, for example randomly setting the value of a specified integer variable in the solution. This means that this variable cannot be changed (or at least changed back to its previous state, depending on the implementation of the tabu search) within X steps.

An iteration of a tabu search generally samples a number of neighbours in the neighbourhood (or all if the neighbourhood is sufficiently small) and moves to the neighbour with highest solution quality (which may be of lower quality than the current state) and which does not violate the tabu list.

This is a simple, yet effective, method. A standard reference for this technique is [GL93].

2.5.2.4 Evolutionary Algorithms

The term 'evolutionary algorithms' encompasses many biologically inspired algorithms for local search. The most common of these are genetic algorithms (GA's), and a common theme is that they work on a population of solutions rather than a single point search such as in hill climbing.

The idea behind genetic algorithms is to maintain a population of strong solutions and then randomly mix together pairs of solutions ("crossover") and permute them in some way ("mutation.") Standard references to these methods are [Hol92, Fog02].

Another form of evolutionary algorithm is that of a memetic algorithm. This is a two stage evolutionary algorithm which utilises a population of solutions and acts as a genetic algorithm in the first stage, with crossover and mutation operations to create new solutions. It then performs a second stage consisting of each member of the population acting as a hill climber — thus performing a local search at each point in the solution space. The stages are then repeated until a stopping criterion is met. As yet, there has been no attempt at implementing memetic algorithms in cutting and packing research. The standard reference for memetic algorithms is [Mos89].

In cutting and packing the solutions are generally represented by a permutation list, i.e. the order of which boxes shall be placed. In this case it makes very little sense to use populations of solutions and mix together individual solutions. As each box may appear only once in a packing, special care must be taken in order to avoid duplicate boxes being packed or individual boxes getting 'lost' in the algorithm. Mixing together two strong solutions in no way guarantees that

the newly generated solution will be strong, and often any structural information or 'good' properties of either solution gets lost in the crossover. In the case of cutting and packing, the method may as well just pick a random point in a solution and randomly shuffle the permutation after this point. For these reasons evolutionary algorithms have had very little positive impact in cutting and packing, where no suitable representation has yet been proposed in which crossover would be an appropriate operation.

2.5.3 Automated Heuristic Generation

A fairly recent research direction is that of allowing a computer to generate novel heuristics to effectively tackle new problems, or new instances of standard problems such as, in this case, cutting and packing. These methods involve generating constructive heuristics or sequential patterns of constructive heuristics to be applied to packing problems, based on algorithm training phases on representative problems. This can effectively reduce the work of trial-and-error on the part of the practitioner/researcher, allowing a computer to try and piece together components of a heuristic or combinations of heuristics in order to build a heuristic placement procedure that can efficiently pack an instance of a packing problem. This should, in theory, be able to get fair results without having to enumerate all of the possible heuristic placement strategies either by hand or by computer, which could be computationally infeasible given a large enough set of heuristic components/permutation space.

Some success has been found in this area in other fields. For example generating production/machine scheduling dispatching rules [GUA06, JJB07, TH08], Boolean satisfiability [Fuk08, BEDP08] and function optimisation [TMC$^+$04, Olt05]. Work in the field of cutting and packing, for example [BHKW11, TMFÁR05], has had a fairly limited effect. As such, the results from the works (on packing problems) are acceptable, though not yet human-competitive in a significant way.

The authors of these papers often refer to these methods of problem solving as 'hyper-heuristics', the definition of which, used in the wider literature, often differs to encompass broader fields of search methodologies, including automated parameter tuning and dynamic neighbourhood searches. Hyper-heuristics were originally described in [BHK$^+$03] as "using (meta-)heuristics to choose (meta-)heuristics to solve the problem in hand" but a more general definition can be given as "an algorithm that works on a search space of heuristics, as opposed to a search space of solutions".

Automated heuristic generation methods, or hyper-heuristics, can be seen as a cross between heuristics (section 2.5.1) and metaheuristics (section 2.5.2).

2.5.4 Exact Methods

Work in exact methods for three-dimensional problems has shown that small problem sizes can be solved to optimality or near-optimality in a reasonable time [Fas99, MPV+07]. On the other hand, heuristic and metaheuristic approaches have been shown to give good results on much larger problem instances in a comparably short amount of time, as evidenced in the references from sections 2.5.1 and 2.5.2.

Exact methods used in solving cutting and packing problems currently fall into three main areas: mathematical programming models, tree searches and dynamic programming, though other approaches exist such as graph-theoretical models [MAA92, LLM02].

2.5.4.1 Mathematical Programming Models

Although many mathematical programming solver classes exist, the standard class of problems formulated in the cutting and packing literature is that of mixed integer linear programming problems (MIP or, synonymously, MILP). These problems are all formulated with linear objective functions (generally, the sum of packed box volumes) and linear constraints (equalities and inequalities). The formulation of mathematical programming models involves describing the problem precisely in terms of combinations of continuous variables, integer variables, binary decision variables and constants. Examples of MIP formulations of the three-dimensional packing problem are given in chapter 6.

Linear problems (that is, without integrality constraints on the unknown variables) can be solved in polynomial time using interior point methods and, in most practical cases, using the worst-time exponential simplex method [Dan98, PS98].

In the case where integer variables are required, the problem becomes $\mathcal{NP}$-hard. This is the standard case of the problems in this thesis. Relaxing the integrality constraints and solving the problem as a continuous linear problem gives an upper bound on the optimal solution quality (when maximising an objective function). The mathematical programming solver then needs to find a solution with integral variables that matches closest to this bound. Of course, the solution space quickly becomes very large due to the combinatorial nature of the problem and many other tricks built up over decades of research must be employed to find the provably optimal integer solution. A good introduction to integer linear programming can be found in [NW88].

Generally, the use of a well refined MIP solver is preferred over creating one from scratch for any given problem. The most common commercial solvers used in research are ILOG CPLEX and Gurobi. Popular non-commercial solvers include SCIP [Ach09] and CBC [LH10].

2.5.4.2 Tree Searches

Tree searches are a common enumeration technique found in computer science and OR. In the case of cutting and packing the standard method is to model each branch of the tree as adding a single box, or a group of boxes to a packing in order to construct a full solution. Of course, bounding by using a predictive heuristic (a.k.a. branch-and-bound) can then be applied to the search in order to 'cut off' infeasible or lower quality solutions during the search. Tree searches have limited success, and can only be applied to small problems due to the high branching factor of modelling a problem in this way. They do, however allow for constraints such as the guillotine cutting constraint to be modelled implicitly. See [Bea85, CH95, FS97, MPV$^+$07] for some examples of tree searches applied to two-/three-dimensional problems. The majority of tree searches in cutting and packing, as opposed to other disciplines, do not guarantee optimality and are often used only to provide strong heuristic/approximate results [PE06].

2.5.4.3 Dynamic Programming

Another enumeration technique that is often found is that of dynamic programming. Dynamic programming takes many forms, but usually involves breaking a master problem down, solving the smaller sub-problems and recombining them in a way which will produce an exact or approximate solution to the original problem, depending on whether recombining the sub-problems can guarantee optimality or not. This has been found to be highly effective in solving the one-dimensional knapsack problem in pseudo-polynomial time and has been applied to other problems in higher dimensions, though it does not often provide many improvements over tree searches and is generally just another form of enumeration.

Dynamic programming can be used to try and solve the problem directly, or as a method to solve a sub-problem in the cutting or packing master-problem [MPT99, CMWX08].

2.6 Summary

An overview of some of the background to cutting and packing problems, in one dimension and higher has been presented. The theoretical and real-world applications of these problems should provide the motivation for the rest of the thesis. We have also covered the main approaches to tackling these problems given previously in the literature and highlighted the strengths and weaknesses of each. These techniques will also be touched upon in later chapters, but in greater context.

3

The 3BF Heuristic and Framework

> Simplicity is the ultimate sophistication.
>
> Leonardo da Vinci

Summary

This chapter presents a hybrid placement strategy (consisting of heuristic construction and metaheuristic improvement phases) for the three-dimensional strip packing problem along with an underlying generalised framework for three-dimensional packing. In addition to the proposed strategy, a number of test results on available literature benchmark problems are presented and analysed. The results of empirical testing of the algorithm show that it outperforms other methods from the literature, consistently in terms of speed and solution quality — producing 28 best known results from 35 test cases.

3.1 Introduction

The method of generating a three-dimensional packing in this chapter is the 'bottom up' approach, as described in section 2.5.1 and used in [BKW04, CPT08, ABHK09, ABK11]. This involves iteratively taking the deepest (i.e. lowest along the y axis - note that in this thesis we will use the terms height, width, length and depth. To avoid any confusion we have labelled these for 3D packings in Figure 3.1, showing how the x, y and z coordinates relate to the width,

length/depth and height of the box or container respectively) available gaps in a packing, evaluating all boxes that will fit into these gaps and using some heuristic scoring method to evaluate them in order to select which boxes will be placed in which gaps. This obviously allows for the exploitation of information regarding the current partial solution, or sub-packing. For example, the locations of the previously placed boxes and the amount of remaining space available within the container etc. can be used, allowing for a dynamic choice of which box to place at a given time — an ability that is not as simple to implement using other methods mentioned in section 4.1. If a gap is unusable (i.e. all available unplaced boxes are too large to fit) then the gap can safely be assumed to be waste and a virtual filler piece can be placed within the exact dimensions of the gap, raising it to the furthest point of its nearest neighbour in order to create a larger surface on which to place subsequent boxes. This general packing methodology can be thought of as a 'best-fit approach' — boxes are placed in locations which are somehow judged to be 'best' by some criteria specified by the algorithm. This approach is formalised in the pseudocode of Algorithm 1. Note that the exact implementations of the methods used in the pseudocode are explained in depth in Section 3.2.

In order to produce a packing, the method finds the available position(s) for a box to be placed in the container, scoring all unpacked boxes in each of the positions based on a given evaluation function, and placing the box with the highest placement score in the 'best' position. That is, a box with a higher placement score is assumed to be a better candidate according to the current placement scoring function. If a gap cannot be filled (or partially filled) by any of the available boxes then it is deemed to be unusable and is subsequently discarded. Due to the problem being offline, and therefore the list of boxes to be packed being static, it can be assumed that if no box can fit a gap on the current iteration of the packing process then no box will be able to fit it at a later stage. If this is the case for any of the gaps then each of the unusable gaps are adjusted to the same y position of their least protruding neighbour (i.e. the box touching the edge of the gap which has the smallest value of its furthest face along the y axis), thus creating

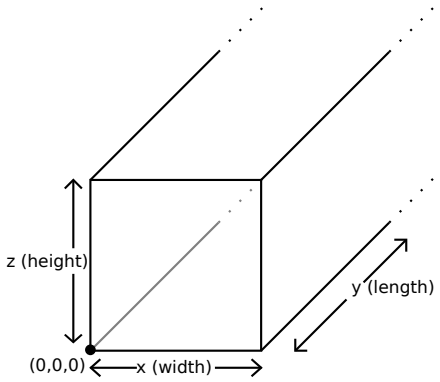

Figure 3.1: A strip packing container, with x, y, and z axes labelled, and accompanying width, length, and height indicators

Algorithm 1 Constructive Packing Algorithm/3BF Framework

Input: a set of boxes to place B, a container C
Output: a set of boxes with associated orientation and placement information, P

$P \leftarrow \emptyset$
while $|B| > 0$ **do**
  $G \leftarrow getLowestGaps(C, P)$
  $bestScore \leftarrow 0$
  $bestOrientation \leftarrow 0$
  for all i such that $1 \leq i \leq |G|$ **do**
    for all j such that $1 \leq j \leq |B|$ **do**
      for all valid orientations o such that $1 \leq o \leq D$ **do**
        $t \leftarrow evaluationScore(g_i,\ b_j^o)$
        if $t > bestScore$ **then**
          $bestScore \leftarrow t$
          $bestBox \leftarrow j,\ bestGap \leftarrow i,\ bestOrientation \leftarrow o$
        end if
      end for
    end for
  end for
  if $bestScore \neq 0$ **then**
    position $b_{bestBox}^{bestOrientation}$ in $g_{bestGap}$
    remove $b_{bestBox}$ from B
    add placement of $b_{bestBox}^{bestOrientation}$ to P
  else
    for all i such that $1 \leq i \leq |G|$ **do**
      mark g_i as invalid for future iterations
    end for
  end if
end while
return P

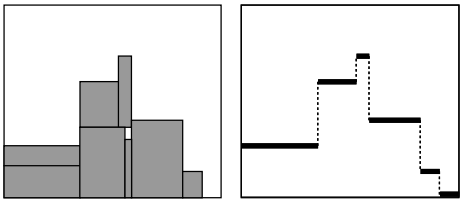

Figure 3.2: (left) a two-dimensional packing, (right) the gaps available in this packing, marked as bold lines.

a larger platform for subsequent boxes to be placed upon. We can imagine this as filling the unusable gaps with an invisible box which fills each unusable gap perfectly up to the protrusion of the neighbour.

3.2 The 3BF Placement Heuristic

This section introduces a new heuristic algorithm, which we refer to as the Three-Dimensional Best Fit (3BF) placement heuristic for the three-dimensional strip packing problem. We also present an implementation for the algorithm in order to gauge the results against a number of available benchmark tests.

The algorithm on which 3BF draws some of its inspiration is [BKW04], which we refer to as 2BF. Though inspiration is drawn from this paper and the general methodology (i.e. a 'best fit' scheme) is the same, there are many significant differences which are discussed in later sections.

In many heuristic algorithms designed for the two-dimensional problem, the solution is found by iteratively taking the head of the list of input rectangles (usually sorted by a common property such as width or volume). Instead of searching the list sequentially, in 3BF the list is searched dynamically in order to find the most suitable rectangles to fill the available gaps. We define gaps as suitable locations at the furthest points (as illustrated two dimensions in Figure 3.2) in a packing in which a box may be placed, unobstructed by other boxes. This means that gaps are effectively two-dimensional 'sites' for a box to be positioned in. This is referred to as a 'skyline' representation in the paper by Burke et al. [BKW04], i.e. the two-dimensional skyline in a three-dimensional packing can be imagined as a heightmap with only rectilinear features (the height/shade is defined by the furthest face of the furthest box in the packing at any point), where every rectangle that can be drawn enclosing a single shade represents a gap that a box can be placed into. The exact implementation details are described in greater detail in chapter 4. In the same paper by Burke et al. the methodology of greedily placing boxes based on their size is referred to as 'best fit.' This method is also employed by our proposed 3BF methodology, which attempts to find a box whose base/footprint (we use the term 'footprint' of a box to refer to the deepest face which will be placed within a gap, i.e. the back of the box when rotated to a given orientation) fills the deepest available gap entirely or, failing that, to

fill as much of the gap as possible — effectively making this a greedy algorithm. When multiple orientations of a box or different boxes fill the same amount of the gap (or multiple gaps are available) then tie breaking is done according to additional rules (explained in section 3.2.1), and remaining ties are broken arbitrarily.

As with the 2BF algorithm there are some issues with time complexity as, at each stage, first a gap needs to be found by searching the container profile and then a box has to be chosen from the input list. There are some attempts to relieve these time complexity bottlenecks as described in the later sections of this chapter.

The representation of gaps and boxes used here is similar to the "extreme-point" method as introduced in Crainic et al. [CPT08]. Initially, in an empty container, a single point will be available for boxes to be placed in — (0,0,0). When a box is added to this location the point is removed and a number of new points are added to the list, allowing subsequent boxes to be placed on top of and around the newly placed box. For every box that is placed, nine new points are pushed back into the gap list, sorted by increasing y location, followed then by increasing z and x locations. The location of the new points are as in the paper of Crainic et al., which also proves the complexity of the point list update procedure as $O(n)$ where n is the number of boxes already placed in the container.

This representation, using a sorted point list, means that in order to find the best gaps in the box the list must only be searched from the beginning until a point with a higher y location than the first point in the list is reached in the case of the Neighbour Score and Maximum Contact placement strategies, or only the first point in the case of the Deepest-bottom-leftmost and Smallest Extrusion strategies (see section 3.2.1). Once potential locations are found, suitable boxes are placed at each location in order to gauge their suitability and checked for collisions with previously placed boxes (as explained in section 3.3.2)

In order to speed up the input list dynamic search each input box is first oriented so that width $\geq$ height $\geq$ length. The list is then sorted into order of decreasing width (if widths are equal the second priority is height and the third is length). This means that only around $\frac{n}{2}$ inspections are needed at each iteration of the algorithm on average, as given a gap of dimensions i by j (where $i \geq j$) we need only traverse the input list until all boxes of dimensions equal to i and j is found (or the first closest answers), as any box found after this will be worse at fitting the gap than these boxes have already been.

As well as speeding up the search this also means that larger boxes are generally placed earlier on in the packing, which often leads to a higher quality solution as there are fewer boxes protruding over the top of the three-dimensional skyline of the container.

After a gap has been selected, the list of input boxes is searched in order to find the best fit for the gap. The result will always be one of three options:

1. One or more boxes are found to have a rotation whose footprint has exactly the right

dimensions to totally fill the gap, possibly after rotation. In this case, the box with the highest evaluation score (as explained in section 3.2.1) is chosen.

2. One or more boxes are found to fit the gap, possibly after rotation, but without filling the gap entirely. If this is the case then the box which fills most of the gap (or in the case of multiple boxes filling the same amount of space, the box with the highest evaluation score) is chosen.

3. No boxes are found to fit the gap.

In the case of option 1, there is no choice as to where the box should be placed within the gap, so it only undergoes the rotations necessary to make it fit and it is placed. In the case of option 2 then the box that fills the most of the gap is used after any necessary rotations, with tie breaking and placement according to the placement strategy (see 3.2.1) currently in use. Finally, for option 3 the gap can successfully be ignored on future iterations of the algorithm (i.e. removed), as the input list of boxes is static.

A pseudocode representation of the class of constructive algorithms that 3BF is part of is provided in Algorithm 1. The set D represents the set of box placements, where a placement consists of a box (originally from B), a specified orientation and a placement point as provided by the `getLowestGaps` method. The `getLowestGaps` method returns a list of placement points (each representing a position along the x-, y-, and z-axes) that a box may be placed in — i.e. so that its lower back left vertex is positioned at the given point, taking into account the previously placed boxes. It is assumed that `getLowestGaps` marks gaps that are unusable (i.e. lie within the location of an already placed box in D) as invalid and ignores invalid gaps. Finally, `evaluationScore` which defines the behaviour of 3BF is described in 3.2.1. We assume that all orientations, o, of any given box are valid unless specified in the dataset.

3.2.1 Placement Strategies

As with 2BF, there are a number of placement strategies, although they are substantially different due to the third dimension having to be taken into consideration. There are also different priorities (i.e. the minimisation of the length of the filled container being the highest priority in 3BF, as opposed to the minimisation of the height of the packing in 2BF). The scores generated by the placement strategies are defined such that a higher score is deemed a better location for that orientated box/gap pair. The placement strategies are illustrated in (the two-dimensional) Figure 3.3 and are as follows:

1. **Deepest-bottom-leftmost**: The box is placed at the deepest-bottom-leftmost position (i.e. with the lowest y position, ties broken by lowest z position and finally lowest x position) in the container, i.e. at the point closest to (0,0,0). The evaluation function

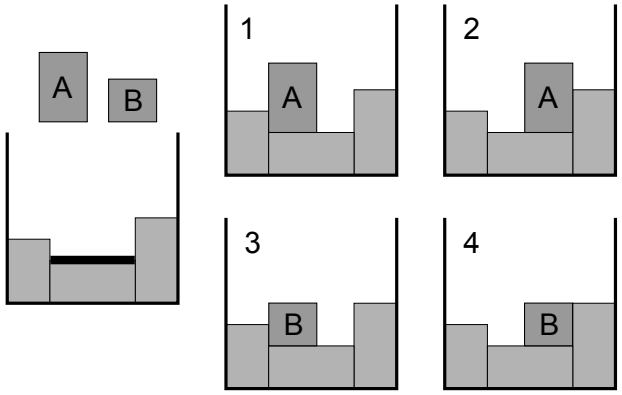

Figure 3.3: The placement strategies in action. (Left) a partial packing and two non-rotatable candidate boxes (with identical widths) for packing next and the lowest gap indicated by a thicker line, (right) packing after one iteration following the (1) Deepest-bottom-leftmost, (2) Maximum Contact, (3) Smallest Extrusion and (4) Neighbour Score rules.

returns the area of the footprint of the box, with ties being broken by furthest extrusion in the y-axis, i.e. a bigger extrusion is deemed a better solution in this placement strategy. Large extrusions placed towards the end of a packing are dealt with by the tower processing technique described in section 3.2.2.

2. **Maximum Contact**: The box with the largest volume is placed so that the sum of its surface areas in contact with other boxes and edges of the container is maximised. The faces of the box are weighted differently by multiplying the surface area (in contact with other boxes/the container) of the back by 4, the left side by 2, the underside by 2 and all other surfaces by 1. Obviously, changing the weightings can lead to different packings with different priorities, e.g. weighting the underside by more would lead to much more stable packings in terms of real-world problems with gravitational constraints.

3. **Smallest Extrusion**: As with the deepest-back-leftmost strategy, though the evaluation function returns 1 divided by the extrusion in the y-axis; thus the box with the smallest far point in the y direction (which fills the gap the most) is chosen and placed in the deepest-bottom-leftmost position.

4. **Neighbour Score**: The box with the largest footprint is placed so that it is touching as many boxes whose furthest face on the y-axis are equal to, or less than, that of the box being placed, i.e. neighbouring boxes that could form vertical walls for subsequent boxes to be placed in front of. This scoring method effectively takes all boxes surrounding the box being tested, and sums the perimeter which is in contact with any of those boxes. Boxes protruding further than the tested box/orientation pair are not included in the summation, and boxes with an identical protrusion to the tested box/orientation pair are given a weighting of 2, i.e. summed twice, as they directly create a larger gap for subsequent boxes to be placed in.

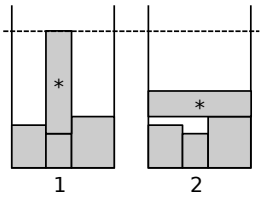

Figure 3.4: A two-dimensional 'tower' (marked with an asterisk) being rotated during the tower processing stage, in order to increase solution quality in a packing.

In the case of Figure 3.3 it can be seen in part (1), the Deepest-bottom-leftmost strategy packs box A into the position as it is the deepest, lowest position available and given that box A has an equal width but higher volume than box B it is packed. Part (3) follows the same logic but as box B has a lower volume it is packed instead. In part (2) of the diagram, box A is placed in the rightmost location of the gap as it shares more contact with the rightmost box previously packed within the container. Box A is chosen over B as it is earlier in the packing list. In part (4) of the diagram box B is placed in this location as it shares the same extrusion as the rightmost box and therefore creates a larger 'platform' for subsequent boxes to be placed on.

3.2.2 Tower Processing

Due to the nature of the algorithm, solutions may potentially be found of relatively poor quality as 'towers' can be placed. These occur when boxes are placed late on in the packing process with their rotations such that the greatest (or second greatest) dimension is assigned to the y-axis, causing them to stick out above the 'skyline' of the container. To combat this problem a further stage is introduced into the algorithm, where the tallest tower is taken and rotated so that the dimension currently assigned to the y-axis is assigned to either of the other two axes and a shorter dimension is assigned to the y-axis. It is then placed back somewhere within the solution in its new orientation. This is repeated until no improvement in solution quality can be found. A two-dimensional representation of this can be seen in Figure 3.4 and the pseudocode is presented in Algorithm 2. The function `furthestPoint` returns the location (i.e. along the y-axis) of the front-most face, that is, the extent of the extrusion of the box in the skyline. The `getAllGaps` method works as the `getLowestGaps` method except that it returns all gaps that have not been marked as invalid rather than just those sharing the lowest y location.

3.3 Metaheuristic Enhancements to 3BF

Though the solution results presented previously represent good quality in short computational times, the findings of Burke et al. [BKW09] suggest that with a hybrid heuristic/metaheuristic

Algorithm 2 Tower Processing Algorithm

Input: a set of boxes with orientation and position information, P, the container they have been placed in, C

Output: the set of placed boxes with orientation and position information, P, with 'tower' boxes processed where possible

1: **loop**
2: let $b_i^o \in P$ be a box with orientation o and position information, with maximum $furthestPoint(b_i^o)$
3: $j \leftarrow furthestPoint(b_i^o)$
4: $k \leftarrow j$
5: $b_i^{best} \leftarrow b_i^o$
6: remove b_i^o from P
7: **for all** orientations $p \in \{1 \dots 6\}$ of b_i, b_i^p, such that $length(b_i^p) < length(b_i^o)$ **do**
8: *place* b_i^p in lowest available gap from $getAllGaps(C, P)$
9: **if** $furthestPoint(b_i^p) < j$ **then**
10: $j \leftarrow furthestPoint(b_i^p)$
11: $b_i^{best} \leftarrow b_i^p$
12: **end if**
13: remove b_i^p from P
14: **end for**
15: position b_i^{best} in lowest available gap from $getAllGaps(C, P)$
16: add placement information of b_i^{best} to P
17: **if** $furthestPoint(b_i^{best}) = k$ **then**
18: exit
19: **end if**
20: **end loop**
21: **return** P

packing strategy the quality of solutions may be further improved. This leads to two distinct placement phases, which are shown in Figure 3.6.

The aim of implementing a second, metaheuristic stage to the initial 3BF algorithm is to increase solution quality whilst maintaining short execution times. This effectively means creating a hybrid algorithm which will place the majority of the input list of boxes using the fast 3BF algorithm, and then placing the remaining boxes using a metaheuristic method for higher quality solutions.

The 3BF heuristic, coupled with a metaheuristic search method along with a decoding procedure (which we call the deepest-bottom-left-fill method, DBLF) gives the hybrid packing strategy. In order to produce orderings that may generate higher quality solutions, the DBLF method is paired with a tabu search (see section 3.3.3 and chapter 2).

3.3.1 Deepest-Bottom-Left-Fill Strategy

Algorithm 3 DBLF Algorithm

Input: a set of ordered, unplaced boxes, B and a container, C
Output: a set of boxes with associated placement information, P

```
1:  P ← ∅
2:  for all i such that 1 ≤ i ≤ |B| do
3:     G ← getAllGaps(C, P)
4:     placed ← false
5:     for all j such that 1 ≤ j ≤ |G| do
6:        if placed = false and fits(b_i, g_j) then
7:           position b_i in g_j
8:           remove b from B
9:           add placement information of b to P
10:          placed ← true
11:       end if
12:    end for
13: end for
14: return P
```

In the research area of two-dimensional cutting and packing problems the most commonly used method for packing regular and irregular shapes involves the bottom-left class of heuristics. These methods involve simply placing the input list of rectangles into the bottom-leftmost location on the packing sheet. Applications of this class appear in the literature as early as 1980 [BJR80, Cha83].

The main issue with using the bottom-left approach is that holes can appear in the packing sheet where subsequent rectangles are unable to be placed, as shown in Figure 3.5.

The solution to this problem is suggested as the bottom-left-fill method, which maintains a list of potentially usable gaps (ordered by increasing distance from (0,0,0) with lowest depth being

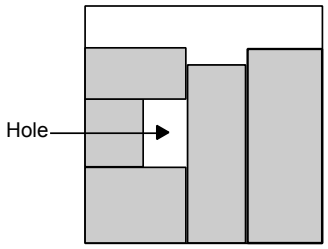

Figure 3.5: 2D packing sheet with a hole.

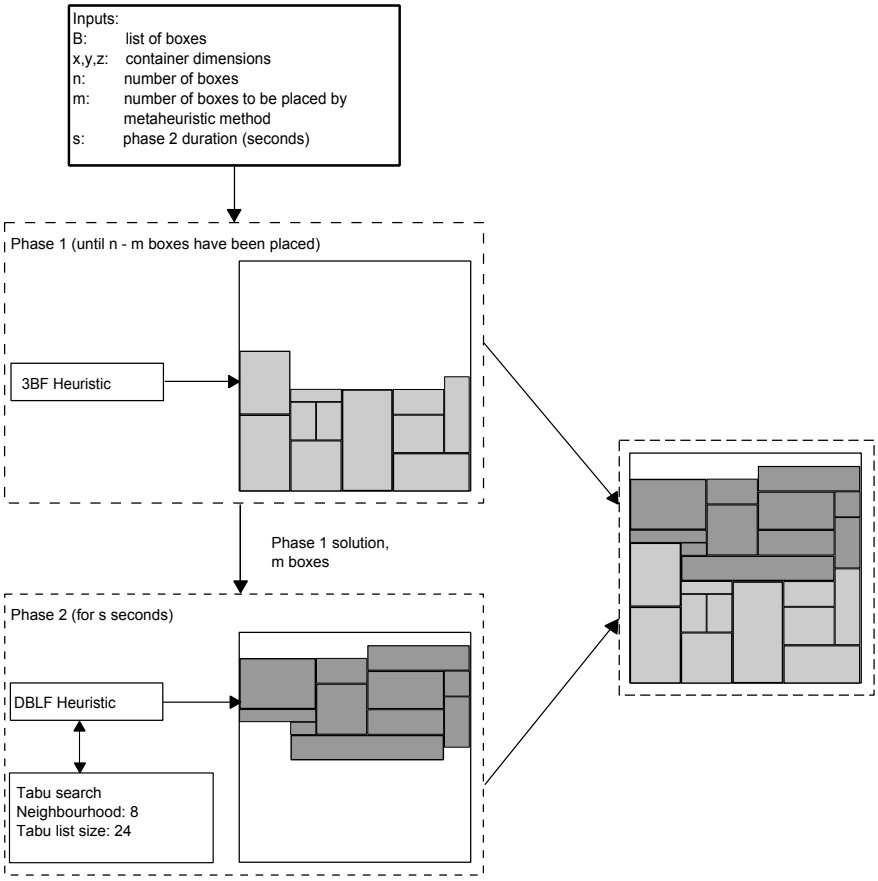

Figure 3.6: A summary of the proposed process.

highest priority and furthest left being lowest priority) for boxes to be packed within and also a list of the previously placed boxes in order to check for collisions with subsequently placed boxes. This method has been shown to improve solution quality over pure bottom-left methods [HT99]. DBLF is an obvious extension of BLF as it simply places boxes in the deepest available position in three dimensions (i.e. closest to (0,0,0) — with the depth/y-axis being of highest priority to minimise, the bottom/z-axis next and finally the left/x-axis) whilst filling available gaps where possible. This approach has also been used previously in literature [KI04], where a detailed description can be found.

The DBLF method is an online algorithm (i.e. it processes each box in the input list in order, as and when the algorithm reaches it) and therefore the order of the input list is important as it determines the output solution. It has been shown in the literature that intelligently pre-ordered input sequences generally generate higher quality solutions [CGJ84] and, specifically, ordering by decreasing dimensions tends to yield the best results [HT01b].

Pseudocode for the DBLF algorithm is provided in Algorithm 3. The `getAllGaps` method returns all usable gaps in ascending order of their y positions (ties broken by ascending z then x positions). The `removeUnusableGaps` method removes all gap locations which lie within any placed boxes, i.e. marks them as invalid, meaning that no other boxes can be placed there without causing overlaps.

3.3.2 The DBLF Method and Axis-Aligned Bounding-Box Trees

If implemented naively, the collision detection in DBLF can take quadratic time, with N boxes requiring a total of $O(N^2)$ comparisons. To attempt to overcome this bottleneck a simple hierarchical space partitioning method used in many three-dimensional computer graphics and gaming systems [Wal02] was implemented — the Axis Aligned Bounding Box tree (AABB tree). This structure is explained more thoroughly in chapter 4 but an explanation is given here for clarity.

AABB trees are used for fast collision detection in some games and other three-dimensional modelling systems as they model each object in the 'world' with a 'bounding box' — effectively a cuboid that encompasses the entire object. Axis-aligned bounding boxes are a special case of bounding boxes which are always parallel to all three axes in three dimensions. Obviously, as the objects being manipulated in this case are already cuboid in shape and can only perform rotations of 90 degrees this is an ideal solution.

Firstly, the AABB tree partitions the container into smaller sections, by recursively splitting each section (initially the whole container) in half along its longest axis each time and placing each half into the child nodes of a binary tree structure, up to a maximum depth of 5 in this implementation. For every candidate box that is added to the AABB tree structure, the tree is searched recursively until the leaf space containing the location of the candidate box is

found. Only boxes from this leaf are then individually tested for collisions with the candidate box, drastically cutting down the amount of time in the practical case for collision detection compared to the naive approach. It is worth noting, however, that the theoretical worst-case complexities of the two methods are the same, as shown in chapter 4.

3.3.3 Tabu Search

The metaheuristic employed in manipulating the input list of m boxes to be assessed using DBLF is tabu search (see [GL93]). Initially the input list for the search was a list of boxes rotated and sorted. Although it is suggested in [BKW09, HT01b] that rectangles (or in this case boxes) are sorted by decreasing height, it was found through experimentation that boxes sorted by decreasing width as their first priority and decreasing height as the lowest priority created substantially better solutions for this problem. The tabu search method generates 8 neighbours (i.e. direct moves from the current position) of the local state at random and assesses the fitness of each state by decoding it using the DBLF method as described previously. The method used to generate a neighbouring solution is straightforward. Two boxes from the given list are chosen at random and swapped by position. The first chosen box is then randomly rotated to one of its possible orientations, each with equal probability (1/6 if all orientations are allowed) and this list is then returned to the DBLF function for evaluation. The neighbour with the highest score (meaning lowest overall solution length) is accepted as the next move and is added to the tabu list, meaning that it can not be revisited again (though due to the finite nature of memory, and the linearly increasing time factor involved with checking if states are already in the tabu list, the length of the tabu list is limited to a certain number of states, S, and when S is reached the first state in the list is simply 'forgotten' and may be revisited).

The parameters for the tabu search were decided upon after initial testing, consisting of setting the neighbourhood sample sizes between 2–100 (in steps of 2) and tabu list length of 0–100 (in steps of 2). The tabu search tended to work well on the datasets provided with a tabu list size of 24 and neighbourhood sample size of 8, though these parameters tended to be fairly robust and values in a similar region produced results of much the same quality.

3.3.4 Adaptive value of m

During initial testing it was also found that a fixed value of m (the number of boxes to be packed using the metaheuristic method) between approximately 10 and 35 yielded the best results, though the implementation competed more strongly when the value of m was allowed to adjust dynamically throughout each run. In this implementation m was started at 10 and was increased by a value of 2 if a higher solution quality was not found within 10 seconds of the run, i.e. after 10 seconds of running time if a higher quality solution had been found then m would remain the same for the next 10 seconds, otherwise it would increase by 2.

		3BF		Karabulut and Inceoglu [KI04]
Dataset	N	**Time (s)**	**Utilisation**	**Utilisation**
C1P1	15	< 0.1	**100.0**	**100.0**
C1P2	29	0.1	**100.0**	**100.0**
C1P3	50	0.4	**100.0**	**100.0**
C1P4	106	1.6	**98.5**	98.0
C1P5	155	4.8	88.0	**100.0**
C2P1	16	< 0.1	**100.0**	**100.0**
C2P2	25	0.1	**100.0**	92.6
C2P3	52	0.4	97.1	**99.0**
C2P4	100	1.8	87.0	**96.2**
C2P5	151	4.0	**98.0**	96.2
C3P1	21	< 0.1	**100.0**	**100.0**
C3P2	31	0.1	92.6	**94.3**
C3P3	51	0.4	87.0	**92.6**
C3P4	101	1.8	89.3	**91.3**
C3P5	151	3.9	82.4	**90.9**
Average			94.7	**96.7**

Table 3.1: Results on the KI datasets.

3.4 Comparative Evaluation of the Packing Strategy

We now offer some experimental results in order to compare the 3BF packing procedure and others that have been proposed in the literature. Results can be seen in Tables 3.1, 3.2, 3.3, and 3.4. The experimental setup is described in the following section.

3.4.1 Experimentation and Results of 3BF

The implementation of the algorithm is in C++ using the Standard Template Library (STL) where appropriate and compiled using the Microsoft Visual C++ 8.0 compiler. A viewer application used for visualisation and inspection of solution quality was also written in C++, using the OpenGL library [All10a]. In order to try and get the best result possible, all four of the placement strategies were tried and the best result returned as the final solution.

Several different sources of test data were used to evaluate the performance of 3BF, and are marked in the results tables. Firstly the KI test datasets used in [KI04] (Table 3.1) were attempted in order for a direct comparison with a previously established packing strategy producing the previous state-of-the-art results (using genetic algorithms along with the DBLF method, GA+DBLF. Though this method is probabilistic, no indication of the number of runs or whether the results are best results or average results across the runs is given by the authors).

		3BF		Bortfeldt and Mack [BM07]	Bortfeldt and Gehring [BG99]
Dataset	N	**Time (s)**	**Utilisation**	**Utilisation**	**Utilisation**
BR1	139	0.3	88.7	87.3	**92.3**
BR2	140	0.4	89.0	88.6	**93.5**
BR3	135	0.5	87.8	89.4	**92.3**
BR4	132	0.6	87.8	90.1	**90.8**
BR5	128	0.7	87.7	89.3	**89.9**
BR6	134	0.9	87.6	**89.7**	89.2
BR7	129	1.1	87.4	**89.2**	87.1
BR8	138	1.3	86.8	**87.9**	84.0
BR9	128	1.6	86.5	**87.3**	80.9
BR10	129	2.0	86.3	**87.6**	79.1
Average			87.6	**88.6**	87.9

Table 3.2: Results on the BR datasets.

		3BF		Bortfeldt and Mack [BM07]
Dataset	N	**Time (s)**	**Utilisation**	**Utilisation**
BR1-XL	1000	8.6	**92.2**	86.9
BR2-XL	1000	10.2	**92.2**	88.3
BR3-XL	1000	12.5	**91.8**	89.8
BR4-XL	1000	14.9	**92.0**	90.2
BR5-XL	1000	16.8	**92.4**	89.9
BR6-XL	1000	19.4	**92.5**	91.5
BR7-XL	1000	22.9	**92.4**	91.0
BR8-XL	1000	26.2	**92.6**	90.8
BR9-XL	1000	30.1	**92.1**	90.9
BR10-XL	1000	35.3	**92.5**	90.4
Average			**92.3**	90.0

Table 3.3: Results on the BR-XL datasets.

		3BF		Burke et al. [BKW04]	
Dataset	N	**Time (s)**	**Utilisation**	**Time (s)**	**Utilisation**
N1	10	< 0.1	**100.0**	< 0.1	88.9
N2	20	< 0.1	**94.3**	< 0.1	**94.3**
N3	30	< 0.1	**98.0**	< 0.1	96.2
N4	40	< 0.1	**96.4**	< 0.1	**96.4**
N5	50	< 0.1	**97.1**	< 0.1	95.2
N6	60	0.1	**99.0**	< 0.1	97.1
N7	70	0.1	**97.1**	< 0.1	93.5
N8	80	0.2	**97.6**	< 0.1	95.2
N9	100	0.4	**98.7**	< 0.1	**98.7**
N10	200	0.6	**99.3**	< 0.1	98.7
N11	300	1.0	**99.3**	< 0.1	98.7
N12	500	1.2	**99.0**	< 0.1	98.0
N13	3152	8.5	**99.6**	1.4	**99.6**
Average			**98.0**		96.2

Table 3.4: Results on the N datasets.

Note that there is a constraint that no box may be rotated for this dataset.

Secondly, the BR datasets (thpack1–thpack10) originally suggested in [BR95] and available to download from the OR-Library (`http://people.brunel.ac.uk/~mastjjb/jeb/info.html`) were adapted for use in the strip packing problem by ignoring the container length dimension but maintaining the rotation restrictions as suggested in the original paper, in order to compare against [BM07] which is also the current state-of-the-art for some of these datasets (BR6–BR10). Tables 3.2 and 3.3 show these results. The method proposed in [BM07] is deterministic and therefore no repeat runs were necessary in the original paper with regards to solution qualities. The current state-of-the-art results for sets BR1–BR5 are, however, held by a parellelised tabu search method described in [BG99]. This paper also uses several variants of probabilistic and deterministic methods to generate solutions but only a single run of each method is used to determine the 'winning' method. It is worth noting that the problem formulation used in the paper includes additional stability constraints where boxes must be supported by the boxes beneath them, i.e. lower in the z-axis. As with the aforementioned papers, only the first 10 instances from each dataset were used in the comparison.

Finally, the two-dimensional datasets proposed in [BKW04] were used for comparison with the different placement strategies to those proposed here (comparing a three-dimensional packing strategy to a two-dimensional strategy is, in this implementation, simply a case of setting all height values in the two-dimensional datasets to a fixed, arbitrarily small number). The packing methods given in [BKW04] are deterministic. The results are shown in Table 3.4.

The number of boxes, N, is shown in each results table for comparison purposes, with the value for the BR1–BR10 datasets being the mean number of boxes per instance. All tests were carried out on an Intel PC with a core speed of 1.86 Ghz using single-threaded code.

Volume utilisation (as a percentage) in the tables can be calculated by taking the optimal length for the packing divided by the maximum extreme y position over all the boxes, where the optimal length is known for the C1P1–C3P5 and N1–N13 datasets, and is calculated in a similar fashion to that of Bortfeldt and Mack [BM07] using a volume lower bound as shown below for the other datasets (where B is the set of boxes to be packed and cw and ch are the container width and height respectively).

$$VolumeLowerBound = \left\lceil \left(\sum_{b \in B} volume(b) \right) / (cw \cdot ch) \right\rceil \tag{3.1}$$

3.4.2 Notes on comparison with 2BF

The results on the N1-N13 datasets are not compared against the best known results, instead they are compared against the original 2BF implementation in order to get a clearer idea of how competitive the newer placement policies are against the original policies presented in [BKW04]. The times taken to run the 3BF experiments can be seen to be notably longer than the 2BF counterparts, though this is understandable due to the different methods of representing the problem between the 2BF and 3BF implementation. A two-dimensional array-based method was initially trialled for use in the 3BF implementation but was shown to be too slow for containers with large dimensions (due to the memory allocation and the constant memory traversal needed) and also did not provide the flexibility of being able to handle boxes and containers with floating point dimensions.

3.4.3 Experimentation and Results of 3BF with Metaheuristic Enhancements

Results in this section relate to Tables 3.5, 3.6, 3.7, and 3.8.

Each run of the algorithm was allowed 160 seconds in total (i.e. including both the deterministic and the metaheuristic phases) to attempt each problem instance, the same running time as used in [BM07]. For comparison on the N1–N13 datasets a time limit of 60 seconds was imposed to compare more fairly with the experimental set up used in the original paper. The initial value of m was set to 10 for all datasets and was allowed to adapt dynamically. The same rotation restrictions described in section 3.4 were imposed and the same test machine used. Note the comparison on the N1–N13 datasets is now with the 2BF heuristic with metaheuristic enhancements (in this case simulated annealing) included as described in [BKW09]. The results included from [BKW09] are the best results over 10 runs. Results reported by 3BF+TS are averaged over 100 runs, with standard deviations being at, or incredibly close to, zero. This is likely due to near optimal solutions being found (local optima) with little to distinguish between

		3BF+TS	Karabulut and Inceoglu [KI04]
Dataset	N	**Utilisation**	**Utilisation**
C1P1	15	**100.0**	**100.0**
C1P2	29	**100.0**	**100.0**
C1P3	50	**100.0**	**100.0**
C1P4	106	**99.5**	98.0
C1P5	155	**100.0**	**100.0**
C2P1	16	**100.0**	**100.0**
C2P2	25	**100.0**	92.6
C2P3	52	**100.0**	99.0
C2P4	100	**98.0**	96.2
C2P5	151	**99.0**	96.2
C3P1	21	**100.0**	**100.0**
C3P2	31	**100.0**	94.3
C3P3	51	**98.0**	92.6
C3P4	101	**95.2**	91.3
C3P5	151	**95.5**	90.9
Average		**99.0**	96.7

Table 3.5: Results on the KI datasets using metaheuristic enhancements.

		3BF+TS	Bortfeldt and Mack [BM07]	Bortfeldt and Gehring [BG99]
Dataset	N	**Utilisation**	**Utilisation**	**Utilisation**
BR1	139	90.0	87.3	**92.3**
BR2	140	89.6	88.6	**93.5**
BR3	135	89.0	89.4	**92.3**
BR4	132	88.8	90.1	**90.8**
BR5	128	88.5	89.3	**89.9**
BR6	134	88.6	**89.7**	89.2
BR7	129	88.7	**89.2**	87.1
BR8	138	**88.3**	87.9	84.0
BR9	128	**87.9**	87.3	80.9
BR10	129	**87.9**	87.6	79.1
Average		**88.7**	88.6	87.9

Table 3.6: Results on the BR datasets using metaheuristic enhancements.

		3BF+TS	Bortfeldt and Mack [BM07]
Dataset	N	**Utilisation**	**Utilisation**
BR1-XL	1000	**92.4**	86.9
BR2-XL	1000	**92.4**	88.3
BR3-XL	1000	**91.9**	89.8
BR4-XL	1000	**92.1**	90.2
BR5-XL	1000	**92.5**	89.9
BR6-XL	1000	**92.6**	91.5
BR7-XL	1000	**92.6**	91.0
BR8-XL	1000	**92.8**	90.8
BR9-XL	1000	**92.3**	90.9
BR10-XL	1000	**92.7**	90.4
Average		**92.4**	90.0

Table 3.7: Results on the BR-XL datasets using metaheuristic enhancements.

		3BF+TS	Burke et al. [BKW09]
Dataset	N	**Utilisation**	**Utilisation**
N1	10	**100.0**	**100.0**
N2	20	**100.0**	**100.0**
N3	30	**100.0**	98.0
N4	40	**99.8**	97.6
N5	50	**99.1**	97.1
N6	60	**99.3**	98.0
N7	70	**98.0**	96.2
N8	80	**97.6**	**97.6**
N9	100	**98.7**	**98.7**
N10	200	**99.3**	98.7
N11	300	**99.3**	98.0
N12	500	**99.0**	98.0
N13	3152	**99.6**	**99.6**
Average		**98.9**	98.3

Table 3.8: Results on the N datasets using metaheuristic enhancements.

improving solutions. Another evaluation method, used at least in the interim, for generating the final solution, such as the 'area below the skyline' score given in [BKW09] could possibly be used to help the search in this respect.

3.4.4 Evaluation of Results

As can be seen in Tables 3.1, 3.2, 3.3 and 3.4, the results obtained by pure 3BF are very competitive with the best known results from the literature. In 18 out of 35 cases 3BF achieves the highest volume utilisation currently known (shown in a bold typeface) further to all 13 from the 2004 Burke et al. paper, and the time taken to do this is considerably less than that of the results from the literature, with the exception of N1-N13 (see section 3.4.2).

As with the results of the purely deterministic 3BF shown previously, 3BF with metaheuristic enhancements performs extremely competitively with the datasets from the literature, using a similar running time in each case. As can be seen in Tables 3.5, 3.6, 3.7 and 3.8 the best known results were achieved (shown in a bold typeface) in 28 of 35 test instances, and a further 13 best results compared to the paper of Burke et al. [BKW09].

3.5 Conclusions

This chapter presents an efficient three-dimensional hybrid placement strategy, which tackles the three-dimensional strip packing problem in an effective manner.

It has been shown that 3BF is a suitably powerful and efficient method of packing boxes within a container for the three-dimensional strip packing problem, and also that with some metaheuristic enhancements (coupled with a DBLF packing strategy) to this solution, quality can be improved even further in a very reasonable amount of execution time. In this work, the 3BF heuristic alone has provided the best known solutions for 18 of 35 datasets, and when coupled with the metaheuristic enhancements it has provided the best known solutions for 28 of the 35 datasets. Both approaches have led to better solutions than the original 2BF papers.

The 3BF packing algorithm has been shown to be very effective over a wide range of problem instances from the literature, and could be directly applied to many other two-/three-dimensional packing problems. The proposed methodology could easily be extended to include other constraints such as balancing and weight distribution due to the flexibility of the framework. Further improvements to solution quality could also be incorporated with novel evaluation functions which would integrate with the framework with minimal changes to the structure of the methodology. The time taken to reach suitably high solution quality with this strategy is very reasonable on realistically sized instances and could be implemented in many industrial settings with relative ease.

4

A Data Structure for Higher-Dimensional Rectilinear Packing

> Representation is the essence of programming.
>
> *The Mythical Man-Month*
> Fred Brooks

Summary

This chapter presents an Abstract Data Type (ADT) for producing higher-dimensional rectilinear packings using constructive methods, the Skyline ADT. Furthermore, a novel method and several approaches from the literature are provided in the form of concrete implementations of the presented ADT. Formal definitions of two higher-dimensional packing problems are given, the concept of gaps is explained and the polynomial growth worst-case behaviour of gaps is shown. The complexity of both the 3BF algorithmic framework and implementations of the ADT are analysed and comparative run-time speeds are compared over a range of datasets from the literature. The data structure described herein may be used in the majority of constructive heuristic and metaheuristic techniques used to tackle three- and higher-dimensional problems.

4.1 Introduction

In this chapter we focus on the bottom-up constructive algorithmic style in packing, which has been used in recent years in many papers, e.g. [MPV00, CPT08, BKW09, RBN09, ABK11]. We

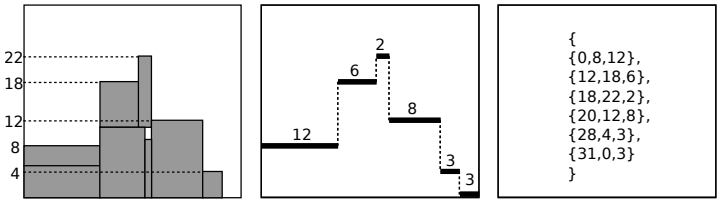

Figure 4.1: (Left) a two-dimensional packing, (centre) the gaps which subsequent boxes may be placed in, represented by thick black lines (i.e. the skyline), (right) the vector representation of the one-dimensional skyline.

first formalise some of the notions and bound the running time of a standard three-dimensional constructive algorithm, the 3BF framework presented in chapter 3. We then present an abstract data type (ADT) for the creation of a solution (or partial solution) and finally give results and bounds on the running times of various concrete implementations of this, both from the literature and a novel approach given here.

4.2 Skylines and Gaps

The term 'skyline' as used to describe the outline of the topmost packed rectangles (i.e. projection in $D-1$ dimensions of a D-dimensional packing) was first coined by Burke et al. [BKW04]. An example of a skyline (along with its numerical representation) can be seen in Figure 4.1. This representation method means that only the feasible positions for subsequent rectangles to be placed in are recorded. Also, rectangles that are placed deeper in the packing, and positions that are unusable, are effectively disregarded as this information is not needed. In the Burke et al. paper the skyline was implemented as a one-dimensional structure representing a two-dimensional packing. A 'one-dimensional skyline' as presented in the paper can be formalised as a vector of n elements (lines), represented as tuples $(\{x_1^1, x_1^2, l_1^1\}, \{x_2^1, x_2^2, l_2^1\}, \ldots, \{x_n^1, x_n^2, l_n^1\})$ where l_n^1 represents the width of the n-th part of the skyline and x_n^d represents its coordinate in dimension d. In the case of the one-dimensional skyline, x^1 represents the position along the x-axis and x^2 is the position along the y-axis. Obviously, $\sum_{i=1}^{n} l_i^1$ equals the width of the container/strip, L_0^1. For all $n > 1$ the relative x^1 position of part n, x_n^1, can be calculated as $x_n^1 = \sum_{i=1}^{n-1} l_i^1$ and $x_1^1 = 0$.

As we are treating a two-dimensional packing as a one-dimensional projection we lose the y-position value associated with each gap, but we store this information as the $D+1$th position value, x_i^{D+1} — in this case x_i^2. This also means that we can use the same data structure (presented in this chapter) in D dimensions to represent a D dimensional packing if we associate a binary value of $x_i^{D+1} = 1$ to a location containing a box and $x_i^{D+1} = 0$ if the location is empty, i.e. by modelling the problem as a $D+1$ dimensional problem with a maximum X_i^{D+1} value of 1. This adds a level of flexibility (as any position in the container may be used rather than

constructing a packing from 'bottom-up') to the packing process but will, as we will see later, increase the computational complexity regarding the number of dimensions and gaps.

In order to find the lowest gaps, the list must be traversed and references to the gaps with the lowest x_i^2 value stored for use in the packing algorithms. The number of gaps at any given point in the packing is denoted as g. Splitting a gap is merely a case of changing its l_i^1 value to its new width, w', and inserting a new gap after it in the list with the same x_i^2 value, a width of $l_i^1 - w'$ and a position value of $x_i^1 + w'$. Merging two adjacent gaps with the same x_i^2 value at positions i and $i+1$ is just a case of adjusting the l_i^1 value by adding the second gap's width, l_{i+1}^1, and finally removing the second tuple (at position $i+1$) from the vector.

For higher-dimensional packing (i.e. D-dimensional with a $(D-1)$-dimensional skyline representation) a similar formalisation can be employed, though the operations on this representation are more involved procedures (see Section 4.3). Formally, the $(D-1)$-dimensional skyline can be represented as a vector of $(D-1)$-dimensional rectilinear parallelepipeds (i.e. gaps) with different associated x^D values, i.e.
$(\{x_1^1, x_1^2, \ldots, x_1^D, A_1\}, \{x_2^1, x_2^2, \ldots, x_2^D, A_2\}, \ldots, \{x_g^1, x_g^2, \ldots, x_g^D, A_g\})$. In this case A_i is a vector of oriented facets for gap i, as the simple line representation used in the one-dimensional skyline does not suffice in higher dimensions. This representation means that the polytopes representing gaps may have holes and may self-intersect (see [dBCvKO08]).

4.2.1 Rectangular Simplification of Gaps

In a simplified two-dimensional skyline representation, gaps can be realised as rectangles. In this case, the set of gaps consists of a set of rectangles whose summed areas may be greater, but are not less, than that of the container's area, i.e. they may overlap as in Figure 4.2. Obviously, this example could be represented using one rectangular gap (shaded) and one L-shaped gap (unshaded) if a suitable representation for gaps is used, but in the case where only parallelepipeds are being placed within the container, and therefore only rectangular footprints are added to the skyline, it is acceptable to simplify using purely rectangular gaps in the skyline implementations, and some implementations lend themselves more naturally to this representation. In this chapter we refer to rectangular gaps of maximal size as 'simple gaps.' More generally we can define a simple gap as one which can be defined fully by two extreme coordinates $(x_a^1, x_a^2, \ldots, x_a^{D-1})$ and $(x_b^1, x_b^2, \ldots, x_b^{D-1})$ where $x_a^j < x_b^j \quad \forall 1 \leq j \leq (D-1)$. For all simple gaps $i, j, i \neq j$ at least one extreme coordinate of i does not lie within j. Note that 'non-simple' gaps (which in this chapter we refer to as just 'gaps') are also maximal by definition, and as such are not fully contained within any other non-simple gap.

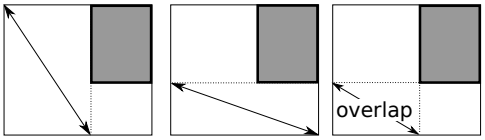

Figure 4.2: Two overlapping simple gaps in a two-dimensional skyline (representing a three-dimensional packing).

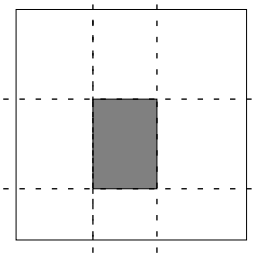

Figure 4.3: The hyperplanes (shown as dotted lines) created within a container when a two-dimensional box (shaded) is added. The 9 defined regions (including the region where the box itself lies) can be observed.

4.2.2 The Relationship Between the Number of Gaps and the Number of Placed Boxes

We now look at the relationship between the number of boxes placed at any stage within the packing, n, and the maximum number of rectangular positions (simple gaps) in the skyline in which subsequent boxes may be placed, g. The results presented in this chapter can also act as valid upper bounds when considering the case where other rectilinear gaps (i.e. non-simple) are allowed in the skyline as all rectilinear polygons can be represented as a set of rectangles (in fact the rectangle covering problem which involves representing a polygon with the minimum possible number of rectangles is itself an $\mathcal{NP}$-complete problem [GJ79]). It is obvious, however, that if non-simple gaps are represented then the number of non-simple gaps can be bounded by $O(n)$ as the set of all polygons at any given height/level can be represented as a single polygon (with holes if more than one box extends to the same height and its 'footprint' is disjoint to the other boxes of that height), and the number of all unique heights in a skyline can be no more than $n + 1$. This is useful in the interval-tree method which we introduce in Section 4.3.1.

We prove bounds on the number of simple gaps by modelling the problem as a specialised case of the 'space partitioning using planes' problem [YY87], where all planes must be aligned orthogonally to all axes and, therefore, it is not possible for all planes to intersect all other planes.

Lemma 1. *For any D-dimensional parallelepiped box, b, being placed within a D-dimensional parallelepiped container the maximum number of dividing hyperplanes (i.e. hyperplanes defined by box b which divide or partition the space) added to the space is $2D$.*

Proof. A visual example of the dividing hyperplanes created by placing a two-dimensional box within a container can be seen in Figure 4.3. We define a D-dimensional parallelepiped box, b, as a vector, d_b (of size D, where the i-th element is indexed by d_b^i) containing sets of the locations of the two extreme points in each dimension. I.e. $\{x_b^i, x_b^i + l_b^i\}$ where $1 \leq i \leq D$). We place b within a D-dimensional parallelepiped container, represented similarly as d_c but containing extreme points in the respective dimension from *all* previously placed boxes plus the container itself. The box being placed is a parallelepiped, and as such may be represented as two extreme points in each dimension. Therefore, $|d_b^i| = 2 \ \forall 1 \leq i \leq D$. We can define a hyperplane by taking each point per dimension and extending it indefinitely in all other dimensions. Each extreme point in each dimension may have a unique position not in d_c^i. The number of new partitioning hyperplanes is thus equal to $\sum_{i=1}^{D} |d_b^i \backslash d_c^i|$. In the worst case $(\sum_{i=1}^{D} |d_b^i \backslash d_c^i|) = (\sum_{i=1}^{D} 2) = 2D$. Therefore, the number of new hyperplanes dividing the D-dimensional space after a D-dimensional parallelepiped has been placed can be no greater than $2D$.

□

Lemma 2. *If n D-dimensional parallelepiped boxes have been placed into a D-dimensional parallelepiped container there are at most $(2n+1)^D$ regions (defined here as an area within space bounded in each direction by a positioned hyperplane or container boundary) in the D-dimensional container.*

Proof. Due to Lemma 1 we know that there can be at most $2D$ hyperplanes per placed box, with the alignment of the hyperplanes being distributed evenly (i.e. 2 perpendicular to, and therefore partitioning, each dimension) and orthogonally throughout space. The number of regions defined in any dimension is the number of hyperplanes dividing it plus 1 (i.e. an empty space within a container is one region, a single dividing plane through this space leads to two regions etc.) assuming all hyperplanes are distinct. Due to the orthogonality constraint no hyperplane aligned to a specific dimension may intersect any other hyperplane of the same alignment (therefore creating new, non parallelepiped regions) and thus the total number of regions in the space is the product of the maximum number of regions for each dimension, i.e. no more than $(2n+1)^D$ if all hyperplanes are disjoint. □

Algorithm 4 Gap Generation Procedure.

Input: a skyline S broken into $r \times r$ two-dimensional rectangular regions with associated heights and an empty $r \times r$ matrix M

Output: an $r \times r$ matrix M, with non-empty elements each containing a simple gap defined by an upper-left and lower-right region location pair (each as a coordinate)

```
1: for all j such that 1 ≤ j ≤ m do
2:   for all i such that 1 ≤ i ≤ m do
3:     Coordinate s ← {i, j}
4:     h ← heightOf(s)
5:     for all k such that m ≥ k > i do
6:       if heightOfAllRegionsBetween(s, {k, j}) = h then
7:         l ← j
8:         while heightOfAllRegionsBetween(s, {k, l + 1}) = h do
9:           l ← l + 1
10:        end while
11:        g ← {s, {k, l}}
12:        if g does not exist as a sub-gap in M then
13:          Store g in M
14:        end if
15:      end if
16:    end for
17:  end for
18: end for
19: return M
```

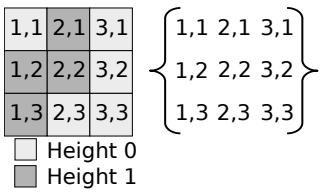

Figure 4.4: A two-dimensional skyline, divided into 9 regions, and the corresponding matrix (transposed for clarity, initially empty, element positions indicated in diagram).

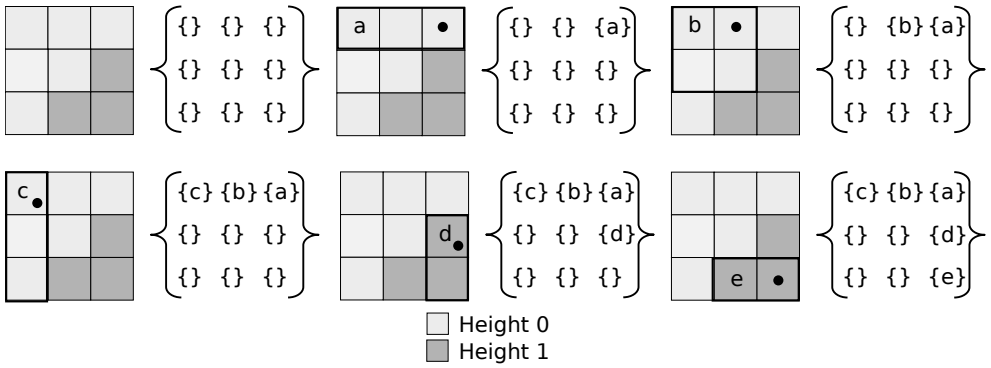

Figure 4.5: Generating all simple gaps in a skyline (with a corresponding matrix, transposed for clarity). Black dots represent top-rightmost regions of the gaps.

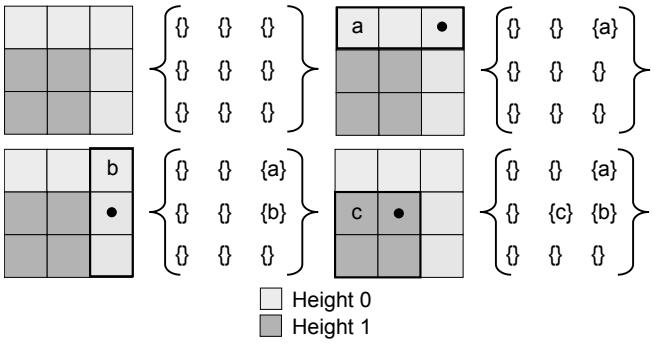

Figure 4.6: Generating all simple gaps in a skyline. Black dots represent the top-rightmost regions of the gaps which have not been used previously in the generation procedure.

Theorem 1. *If n D-dimensional parallelepiped boxes are packed into a D-dimensional containing parallelepiped there will be, in the worst case, $O(n^D)$ simple gaps for subsequent boxes to be placed in.*

Proof. If we know the locations of all regions, as defined in Lemma 2, then we can generate all possible simple gaps.

It is obvious that any simple gap can be defined by either a single region or a pair of regions (representing two extreme points of a simple gap), and therefore the number of simple gaps is bounded from above by the number of single and combined pairs of regions, $(2n+1)^D + \frac{(2n+1)^D(2n)^D}{2} = O(n^{2D})$.

We will now show that this bound can be tightened to $O(n^D)$. We do so by presenting a constructive approach to generating all of the gaps in a packing and show this can not be more than the number of regions bounded previously. We will demonstrate this with a two-dimensional skyline, but the idea can be extended to any number of dimensions.

The gap generation procedure is given in Algorithm 4. We now give an example of the algorithm based on Figure 4.5. Figure 4.4 shows the layout of a given skyline, and the (initially empty) corresponding matrix, which has been transposed to make the mapping more intuitive in this example. M is initially empty, and the skyline is broken into 9 regions with heights either 0 or 1. Lines 1 – 3 set s to be $\{1,1\}$. Line 4 sets the value of h to be the height of region $\{1,1\}$, i.e. 0. Lines 5 and 6–16 'stretch' the gap to the edge of the container. Lines 8 to 10 then attempt to extend the gap downwards, though are unable to due to the height difference of region (3,2). Gap a (g in the pseudocode), represented by the pair {(1,1),(3,1)} is then stored in the matrix at position (3,1), i.e. the position of the top-rightmost region of a. The process is then repeated, ignoring gaps which are already contained within the gaps in M, such as {(1,2),(2,2)}. In the case given, the gap can always be stored in the matrix at the position of the top-rightmost region of the gap. This is not always the case, and if this happens then the highest, rightmost available position is to be used, as is the case in Figure 4.6.

It is obvious that the matrix can store all possible simple gaps using the method provided, and that all simple gaps are represented in this manner. It then holds that the $r \times r$ matrix M (where $r \times r = r^2 = (2n+1)^2$, or more generally for D dimensions $O(n^D)$) can represent all possible simple gaps in the two-dimensional skyline, and by extension the number of simple gaps cannot exceed the number of regions in the skyline, and hence is bounded by $O(n^D)$.

□

4.2.3 A Bottom-Up Constructive Approach to Packing

We now take the 3BF framework presented in chapter 3 and [ABK11]. In order to give a detailed analysis of its complexity we must break the 3BF framework (given again in this chapter as

Algorithm 5, which is functionally equivalent to its counterpart Algorithm 1 which we have already seen in chapter 3) down into finer-grained routines which we can then characterise the complexity of later. We present an analysis of the general 3BF packing framework in terms of the input list size, N, and the number of dimensions, D.

Algorithm 5 Constructive Packing Algorithm.

Input: a set B of N boxes to place, a container C
Output: a set of boxes with associated placement information, P

```
1:  P ← ∅
2:  while |B| > 0 do
3:    G ← lowestGaps(C)
4:    bestScore ← 0
5:    for all i such that 1 ≤ i ≤ |G| do
6:      for all j such that 1 ≤ j ≤ |B| do
7:        if isContained(b_j, g_i) then
8:          t ← evaluationScore(g_i, b_j)
9:          if t > bestScore then
10:           bestScore ← t
11:           bestBox ← j, bestGap ← i
12:         end if
13:       end if
14:     end for
15:   end for
16:   if bestScore ≠ 0 then
17:     position b^bestBox in g^bestGap
18:     splitGap(g^bestGap, b^bestBox)
19:     changeHeight(g^bestGap, height(b^bestBox))
20:     remove b^bestBox from B and add to P
21:   else
22:     for all i such that 1 ≤ i ≤ |G| do
23:       n ← neighbouringGaps(g_i)
24:       changeHeight(g_i, min(height(n)))
25:     end for
26:   end if
27: end while
28: return P
```

4.2.3.1 Complexity Analysis

As can be seen from the now familiar 3BF framework given in Algorithm 5, the main algorithm loop at line 2 is dependent on the time taken to place all boxes. This, in the worst case, will mean that for every box placed (lines 18 to 20) all previous gaps will need to be filled and raised to create a suitably large gap to place each subsequent box, in (lines 22 to 25). The main loop, starting at line 2 will therefore take $O(ng)$ iterations to complete. We will therefore call function `lowestGaps` $O(ng)$ times. As `isContained` is nested within two for-loops of sizes

$|G| = g$ and $|B| = n$ in the worst case it is obvious that it will be called ng times per iteration, or $O(n^2g^2\texttt{isContained})$ iterations in total. This same argument holds for `evaluationScore`, giving $O(n^2g^2\texttt{evaluationScore})$. If we were to allow all rotations of each box then we would have to increase the runtime of `evaluationScore` and `isContained` by a factor of $D!$ as this is the maximum number of unique orthogonal alignments of a parallepiped in D dimensions. The maximum number of rotations being $D!$ is intuitive, as when deciding a rotation for a box we fix one of its D lengths to dimension one, then one of the remaining $D-1$ lengths to dimension two and so on until all dimensions have been fixed, so we have $D \times (D-1) \times \ldots \times 1$ possibilities, or $D!$.

Having $D!$ allowable rotations would obviously dominate many of the terms in the analysis, and would quickly cause any packing method to become intractable in higher dimensions, so for the purpose of this analysis we assume that a fixed, constant number of rotations are allowed for any given D. It is clear that the remaining pseudocode inside the two for-loops takes constant time per iteration. As lines 18 to 20 are contained within the main loop they will be called once per box placed (i.e. a box cannot be placed more than once in a packing). Therefore `splitGap` and `changeHeight` will be called n times in total from this section of pseudocode, regardless of how many times the main loop is iterated, giving a bound of $O(n(\texttt{splitGap} + \texttt{changeHeight}))$. Line 23 will be run in $O(ng^2\texttt{neighbouringGaps})$ and due to the fact that N may contain a list of all gaps (minus the one being inspected), and the function to find the minimum height in this list will take time linear with respect to $|N| = g$, the remaining line will take $O(ng^2(\texttt{changeHeight} + min(height(N)))) = O(ng^2(\texttt{changeHeight} + g))$.

This leads us to an overall complexity for the algorithm, depending heavily on the implementation details of the functions and also the upper bound total number of gaps, g, of:

$$
\begin{aligned}
& O(ng(\texttt{lowestGaps}) + n^2g^2(\texttt{isContained} + \texttt{evaluationScore}) + n(\texttt{splitGap} + \\
& \texttt{changeHeight}) + ng^2(\texttt{neighbouringGaps}) + ng^2(\texttt{changeHeight} + g)) \\
= \; & O(ng^3 + n^2g^2(\texttt{isContained} + \texttt{evaluationScore}) + ng^2(\texttt{neighbouringGaps} + \\
& \texttt{changeHeight}) + ng(\texttt{lowestGaps}) + n(\texttt{splitGap})).
\end{aligned}
$$

4.3 An Abstract Data Type for Packing

Given in Figure 4.7 is a UML description of the ADT used for representing a three-dimensional packing, i.e. a two-dimensional skyline. As we have seen in Section 2.5.1, the functions of the ADT are all that are needed to generate a three-dimensional packing in conjunction with Algorithm 5.

Skyline
+getLowestGaps() : {RectilinearPolygon} +getNeighbouringGaps(rp : *RectilinearPolygon*) : {RectilinearPolygon} +splitGap(rp : *RectilinearPolygon*, gap : *RectilinearPolygon*) +changeHeight(rp : *RectilinearPolygon*, newHeight : *Integer*) +isContained(rp : *RectilinearPolygon*, gap : *RectilinearPolygon*) : Boolean

Figure 4.7: The Skyline ADT for higher-dimensional packing.

The `lowestGaps` function returns a list of gaps (i.e. rectilinear polygons in the three-dimensional packing/two-dimensional skyline approach) that are located at the lowest position in the skyline (i.e. the deepest position in the container).

Initially this will be a polygon representing the base of the container. The `isContained` function returns a Boolean value: true if the specified box fits within the specified gap and false otherwise.

`evaluationScore` is specific to the associated packing algorithm and is a 'fitness' score which dictates which box will be placed in which gap at each iteration of the algorithm. The box/gap pair with the highest fitness score are always chosen to be placed. Depending on the evaluation function tie breaking may need to be specified.

`splitGap` is used to split a gap that has been returned previously into a number of new gaps. The number of new gaps ranges from zero if the box fits the gap perfectly, to any positive number if the box does not fill the gap entirely, e.g. its footprint is smaller than the gap.

`changeHeight` is used to change the height/depth of part of the skyline to a new value, for example when a box has been placed or an unusable gap is found. The final function,

`neighbouringGaps`, returns a list of the gaps in the skyline that are at higher positions than the gap being queried, but touch its perimeter. This is used to determine how high to raise an unusable gap so that it creates a larger platform along with a neighbouring gap that may be usable in a future iteration of the algorithm.

4.3.1 Implementations of the ADT

We will now look at five methods of implementing the ADT in total, four taken from the literature and a new implementation described in this chapter. The implementations will be analysed in terms of computational complexity of the operations and timing tests on representative benchmark instances from the literature.

We assume that `evaluationScore`, i.e. the scoring method for the boxes, takes constant time, for instance simply returning $L_i^1 \cdot L_i^2$ for box i. Without loss of generality we hereby describe the L^d dimension as the 'height' of the skyline at a specific point. The complexities of the implementations are summarised in Table 4.1.

4.3.2 Array Representation

The first and simplest implementation of the Skyline ADT is by using an array of $D-1$ dimensions to represent the packing, i.e. a two-dimensional array to represent a three-dimensional packing. The array obviously has dimensions of $L_0^1 \cdot L_0^2 \cdot \ldots \cdot L_0^{D-1}$, and L_0^D is the unconstrained dimension to minimise for the strip packing problem, or the maximum value allowed in the container loading problem. The value of each array element is the corresponding L_0^D value of the current packing. In order to determine a list of simple gaps at the lowest possible position the array must be traversed fully. To get neighbouring gaps the array needs to be checked around the edges of the queried simple gap location. Raising (and lowering) areas of the skyline to a new position based on a placed box, i, is simply a case of setting the array values between $(x_i^1, x_i^2, \ldots, x_i^{D-1})$ and $(x_i^1 + L_i^1, x_i^2 + L_i^2, \ldots, x_i^{D-1} + L_i^{D-1})$ to the new $x_i^D + L_i^D$ value. Splitting gaps is a redundant operation in this implementation as it dynamically searches for new gaps at each iteration and implicitly holds this information in the representation. The `isContained` function is simply a rectilinear polyhedron in rectilinear polyhedron check.

For the complexity analysis of the array representation we introduce a new variable, s, representing the mean size of the dimensions of the container, i.e. $mean(L_0^1, L_0^2, \ldots, L_0^{D-1})$. As we will see, the size of the container dominates the complexity of some of the operations in this representation.

- `lowestGaps`. To generate all lowest gaps the array must be fully traversed to find the lowest point in the packing. Note that this information can be cached, but will not affect the overall asymptotic complexity of the operation. Scanning the array once will take $O(s^D)$ time. Once the height of the lowest point of the packing is known the array must be traversed again, creating gaps at every position (i.e. 'starting point') where the lowest value is found, and growing these outwards until an element of different (i.e. greater) height is encountered. The gap is then added to the list to be returned to the algorithm. This leads to a complexity of the order $O(s^{2D})$ as each element may be inspected per 'starting point' found.

- `neighbouringGaps`. The procedure to find a list of heights of neighbouring gaps when a box is placed is simple. We just inspect all array elements around the perimeter of the box in question. This leads to a worst case complexity of $s^{D-1}2D = O(s^{D-1}D)$.

- `splitGap`. This operation is redundant as a change in height of a section of the skyline/array will implicitly change the location/size of any gaps affected (i.e. as the gaps are calculated dynamically in this representation so this operation does not need to be called). This leads to a complexity of $O(1)$.

- `changeHeight`. In the worst case a change of height of the skyline will mean updating all elements of the array to the new height. $O(s^D)$.

- `isContained`. As the size of each gap is calculated as it is discovered this will take time bounded by $2D$, or $O(D)$, as all that is needed is a rectilinear polyhedron in rectilinear polyhedron check to ensure that a box fits within the gap.

4.3.3 Collision Detection Representation

The second implementation uses brute-force comparison for collision detection. The method stores an ordered list (in a balanced binary tree) of D-dimensional coordinate locations based on previously positioned boxes. The list is ordered from lowest to highest x_i^d value. Each time a box, i, is placed, the coordinate location in which it is placed is removed, and D new coordinates are added — e.g. in the three-dimensional case at $(x_i^1, x_i^2 + L_i^2, x_i^3)$, $(x_i^1 + L_i^1, x_i^2, x_i^3)$ and $(x_i^1, x_i^2, x_i^3 + L_i^3)$. This obviously makes finding the lowest gaps an easy operation and splitting the gaps redundant, though other operations require comparing each box with all other previously placed boxes. This method also only allows a maximum of D gaps for every box placed, and thus when using this implementation of the data structure $g = O(nD)$ rather than $O(n^D)$ which means that portions of the solution space are inaccessible as obviously not all gaps can be represented.

- `lowestGaps`. In the worst case the number of gaps returned will be a constant factor of the number of previously placed boxes due to the fact that a factor of D number of potential gap coordinates may be added to the skyline at each iteration of the algorithm, but only one per axis — therefore only one per box can be 'lowest' in the packing. Recall that the list of coordinates is sorted by depth. Therefore, the complexity is at most no more than $O(log(nD) + k)$, where k is the number of gaps returned.

- `neighbouringGaps`. The number of neighbouring gaps in this representation will actually be the number of neighbouring boxes previously placed, and therefore a constant factor time per dimension collision check will be needed for each previously placed box, i.e. a complexity of $O(nD)$.

- `splitGap`. This operation involves checking the list of potential gap coordinates and removing any that lie within the bounds of the box in question (taking time proportional to D). As the number of gap coordinates is bounded by a factor of nD this will take $O(nD^2)$ time.

- `changeHeight`. In this representation we mean adding D new coordinates to the gap list (of size bounded by $O(nD)$). This can be done by a binary insertion, and so takes $O(D \cdot log(nD))$ time.

- `isContained`. This involves checking all previously placed boxes for collision and therefore takes $O(nD)$ time.

4.3.4 Plane Representation

The third implementation is similar to the brute-force collision detection method but represents gaps as D lists of axis-aligned $(D-1)$-dimensional 'planes' (actually rectangles) at each coordinate (as described in the brute-force method). Using these planes means that checking whether boxes fit into the gaps is a simple operation, but updating the lists of planes at each iteration of the algorithm is time consuming. For a detailed explanation of the implementation refer to [ABHK09]. This method also bounds the number of gaps available by a factor of D of the number of placed boxes, i.e. $g = O(nD)$ and as such portions of the solution space cannot be accessed as not all possible gaps are represented. In fact, this method is even more restricting than the collision detection method as large boxes cannot initially be placed adjacent to smaller boxes (a constraint of having 'planes' of fixed sizes) and a series of decreasing box sizes can be observed early on in the solutions generated. For this reason it is less effective than the simple collision detection method at creating efficient packings.

Although in the original paper describing this method in detail ([ABHK09]) it is not specified in which order the list of gaps is stored, we hereby assume that they are stored, sorted by depth, in a balanced binary tree so as to improve the complexity of some of the following methods.

- `lowestGaps`. As with the collision detection representation the number of gaps in the skyline will be a factor of nD, and the lowest at any point a constant factor of nD. In the worst case the complexity will be $O(log(nD) + k)$.
- `neighbouringGaps`. Again, this is similar to the collision detection method and so will have a complexity of $O(nD)$.
- `splitGap`. The operation of splitting gaps involves changing the sizes of all planes that overlap with the area to be split and potentially adding new gaps to the representation (i.e. if the split off area is smaller than the planes it is splitting, new gaps/planes need to be added to the representation to store the 'leftover plane'). In the worst case this could involve updating planes stored in every gap location already in the representation. A gap may itself hold planes created by every box previously placed and therefore to update all planes (with each update costing time $O(D)$) in all gaps would take $O(n^2D^3)$ time.
- `changeHeight`. To change the height of a gap the plane information must be updated in surrounding gaps. As with `splitGap`, in the worst case this may involve updating every other gap in the representation, and there may be one plane per gap for every box previously placed. Therefore, this operation will also take $O(n^2D^3)$ time.
- `isContained`. The test for containment within a gap will take time bounded by $O(D)$ for each plane being stored within the gap. As we have seen previously there may be one plane per dimension per gap for every box previously placed. Therefore, this will take $O(nD^2)$ time.

4.3.5 Axis-Aligned Bounding Box Tree Representation

Axis-Aligned Bounding-Box trees, or AABB trees, are used in [ABK11] to represent the packing structure. These are standard tree structures used in many computer games and other three-dimensional graphics applications ([Wal02]). The idea is that space is recursively partitioned into smaller regions so that collision detections for boxes need only be applied to the subsets of previously placed boxes, located in small portions of the space. Other than this spatial partitioning, the operations are much the same as with the brute-force comparison method. In this implementation the height of the AABB tree was set to five as this number produced the best results during initial testing.

This representation is very similar to the naive collision detection method in terms of its time complexity (assuming a binary tree of fixed depth). We will see later that although this method has the same theoretical time complexity as the collision detection method, in empirical testing it generally works faster than the naive method.

- `lowestGaps`. Due to the similarity with the collision detection method $O(log(nD) + k)$.
- `neighbouringGaps`. In the worst case all partitioned spaces will need to be checked (i.e. in the case that a box manages to lie across all spaces) and therefore the time taken will be $O(nD)$.
- `splitGap`. As with the collision detection method, $O(nD^2)$
- `changeHeight`. As with the collision detection method, $O(D \cdot log(nD))$.
- `isContained`. As with `neighbouringGaps`, $O(nD)$.

4.3.6 Interval Tree Representation

The final implementation proposed in this chapter is the use of interval trees. This approach has not been presented in the three-dimensional cutting and packing literature previously. Interval trees are data structures used widely in computational geometry applications ([dBCvKO08]). The implementation presented here stores the skyline as a collection of rectilinear polygons (with an associated height position) in an ordered list, sorted from lowest position in the skyline to highest. Each polygon is made up of lines stored in the interval tree and doubly linked to the polygon. This means that fast comparisons of surrounding 'neighbours' in the skyline and fast changes of height of portions of the skyline can be executed by querying the interval tree or changing the height value and moving the polygon in the list respectively. Splitting gaps and merging adjacent gaps in the skyline after a box has been placed can be done using fast Boolean mask operations on the rectilinear polygons (in two dimensions see [WW88] and for a general, higher-dimensional approach see [BMP99]) and reinserting them into the list and interval tree.

A good reference for information and proofs about complexity of the operations on the interval tree itself can be found in [dBCvKO08], which we refer the interested reader to.

It is worth remembering that as the interval tree representation is not limited to simple gaps it can in fact just store a polygon (with holes as necessary) per unique height in the skyline, lowering the number of gaps at any time to n.

- `lowestGaps`. As we store gap information in two separate data structures (as described in Section 4.3.1) we only need to lookup lowest gap information in the depth sorted list of gaps, therefore taking $O(log(n) + k)$.
- `neighbouringGaps`. To get information about surrounding gap polygons we only need to query the interval tree, taking $O(Df \cdot log(f) + k)$ where k is the number of results returned and f is the number of facets defining the gap. We know that f is bounded from above by a factor of n (i.e. 4 in the two-dimensional case as only rectangles can be added to the skyline, or $2D$ generally) and as such $O(f) \leq O(Dn)$.
- `splitGap`. To split a gap we first need to remove the gap from both the interval tree and the ordered list, taking $O(Df \cdot log(f))$ and $O(log(n))$ respectively. We then perform Boolean masking operations taking $f \cdot log(f)$ in two dimensions or $O(f^2D^3 2^D)$ in higher dimensions (as explained in [WW88] and [BMP99] respectively) . Insertion of the new gaps now takes place back into both the interval tree and the ordered list, again taking $O(Df \cdot log(f))$ and $O(log(n))$ respectively. Therefore, this operation takes $O(Df \cdot log(f))$ in two dimensions, or $O(f^2D^3 2^D)$ generally.
- `changeHeight`. Changing height of a gap is simply a case of removing the gap from the ordered list and reinserting it at the correct position, finding its neighbours and merging with any of them at the same height. This obviously takes $O(Df \cdot log(f) + k)$ or $O(f^2D^3 2^D)$ time as explained previously (removal/insertion in a sorted list, a call to `neighbouringGaps` and the Boolean masking operations explained previously).
- `isContained`. The containment operation will check (in time $O(D)$) each of the $2D$ facets of the box being placed against every facet (of which there are f) of the polygon. As such this operation takes $O(Df)$ in the worst case.

Representation	`isContained`	`neighbouringGaps`	`changeHeight`	`lowestGaps`	`splitGap`
Array	$O(D)$	$O(s^{D-1}D)$	$O(s^D)$	$O(s^{2D})$	$O(1)$
Collision	$O(nD)$	$O(nD)$	$O(D \cdot log(nD))$	$O(log(nD)+k)$	$O(nD^2)$
Plane	$O(nD^2)$	$O(nD)$	$O(n^2D^3)$	$O(log(nD)+k)$	$O(n^2D^3)$
AABB Tree	$O(nD)$	$O(nD)$	$O(D \cdot log(nD))$	$O(log(nD)+k)$	$O(nD^2)$
Interval Tree	$O(Df)$	$O(Df \cdot log(f)+k)$	*[1]	$O(log(n)+k)$	*[1]

Table 4.1: Summary of the complexities of the implementations.

4.3.7 Summary of Complexities

When all operation complexities are applied to the packing algorithm complexity the following overall complexities can be observed:

- *Array.* $O(n^{3D+1} + n^{2D+1}s^D + n^{D+1}s^{2D})$.
- *Collision.* $O(n^5D^3)$.
- *Plane.* $O(n^5D^5)$.
- *AABB Tree.* $O(n^5D^3)$.
- *Interval Tree.* $O(n^4(Df))$ if $D \leq 2$, $O(n^3f^2D^32^D)$ otherwise.

It is worth recalling that the collision detection, plane and AABB tree methods do not allow for all possible simple gaps to be represented ($|G| = O(nD)$ as opposed to $|G| = O(n^D)$) and therefore scale better into higher dimensions as the number of gaps dominates most operations in the packing problems studied here. On the other hand, the array based method and the interval tree representation do allow for all gaps being represented (and therefore can achieve higher volume utilisations in packing compared to the other methods) and for lower dimensions are at least competitive with the 'approximating' implementations.

4.4 Empirical Results

In order to test the 'actual' run time of the implementations in an empirical setting, several experiments were run on some representative problem sets from the literature (BR1–BR10 and BR1XL–BR10XL, see [BR95] and [BM07] respectively) and some artificial sets created to test various aspects governing problem difficulty (i.e. number of boxes to be placed and physical magnitude of the container/boxes). The evaluation/scoring method was simply the footprint of the largest box that would fit in the lowest gap (i.e. the area that would be updated in the skyline) with ties being broken randomly. All tests were carried out 100 times on an Intel PC with a core speed of 1.86 GHz using single-threaded C++ code for the MSVC compiler. The average time taken across all runs is reported.

[1] $O(Df \cdot log(f)+k)$ or $O(f^2D^32^D)$, see section 4.3.6

4.4.1 Datasets from the Literature

The results in Figure 4.8 show the time taken to run the first 10 instances of each of the Bischoff-Ratcliff (BR) datasets presented in [BR95] for each implementation of the ADT. The graph shows a fairly constant increase in time taken, from dataset BR1–BR10. This is due to the increasingly heterogeneous property of the boxes in the datasets. There are approximately 90–150 boxes to pack in each instance of the BR datasets. The number of distinct box types in each of BR1–BR10 is 3, 5, 8, 10, 12, 15, 20, 30, 40 and 50 respectively. As such the homogeneity of the datasets decreases (or the heterogeneity increases). The implementation of the packing algorithm only checks each box type (not each box individually) to see if it will fit in the lowest gaps at each iteration and for its evaluation score, naturally leading to more comparisons as the homogeneity of the datasets decreases. The increase in heterogeneity also leads to a more 'fractured' skyline, as there are fewer boxes of identical dimensions which 'fit together' perfectly (allowing the merging of adjacent gaps). The only implementation that does not seem to suffer from this property is the collision detection method. This is unsurprisingly fairly robust as it has to (in the worst case) check for collisions with *every* previously placed box regardless of its position or size and also each box placed using this implementation always creates exactly $D = 3$ new gap positions, whereas the other methods may create substantially more.

Similarly, the results in Figure 4.9 are from the first 10 instances of each of the extended datasets (BRXL), described in [BM07]. The number of box types increases identically to the BR datasets and the increase in time taken for all but the collision detection method can be observed in a similar fashion.

The trend in time taken holds across both datasets (from BR3 and BR3XL onwards), with the interval tree method consistently fastest, followed by the axis-aligned bounding box tree method, array based method, and finally the planes method. The brute force collision detection method stays fairly constant despite the change in heterogeneity.

4.4.2 Scaling the Problem Instance Size

A simple dataset was created to investigate the effect of problem instance size on runtime for each implementation of the ADT. A container with base of dimensions 233×220 units was used, and initially 12 cube boxes of dimensions $10 \times 10 \times 10$ units were packed into it. The number of boxes was then scaled by 1–250 (i.e. a range of 12–3000 boxes being packed) and the experiment repeated. The results can be seen in Figure 4.10. There are periodic 'bumps' in the graphs, shown clearly in all methods except the interval tree and indicated by the vertical dashed lines. These bumps can be explained fairly easily. The boxes being packed are homogeneous, and therefore the skyline will always be at most at two different heights, h and $h + 10$ (as the dimensions of the boxes are all 10 units). All of the implementations of the ADT adjust the gaps in the skyline as a box is placed. The four implementations displaying visible bumps all

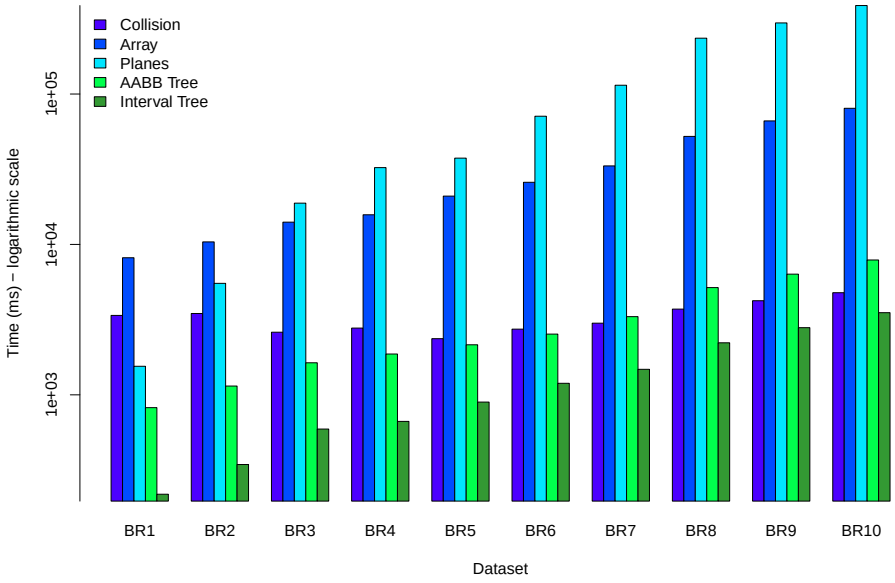

Figure 4.8: Graph showing time taken on BR datasets (logarithmic time scale).

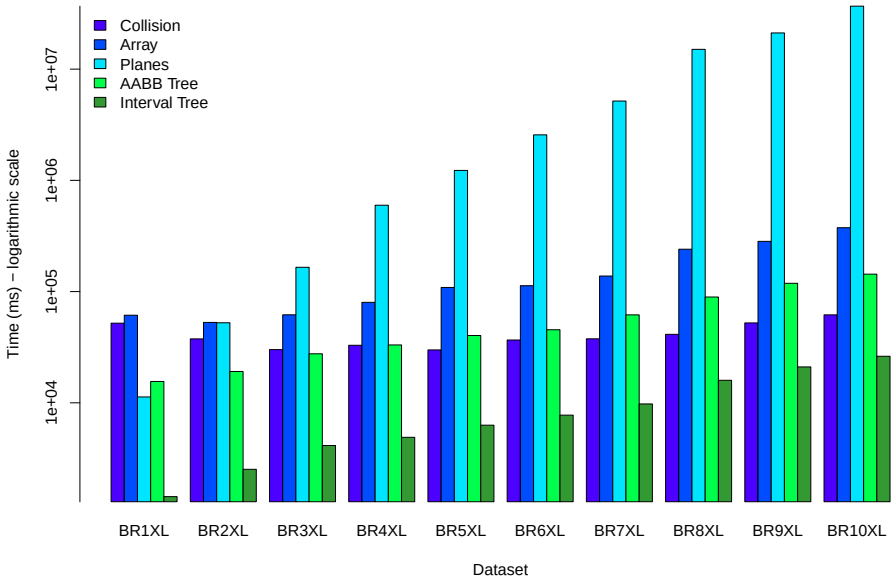

Figure 4.9: Graph showing time taken on BRXL datasets (logarithmic time scale).

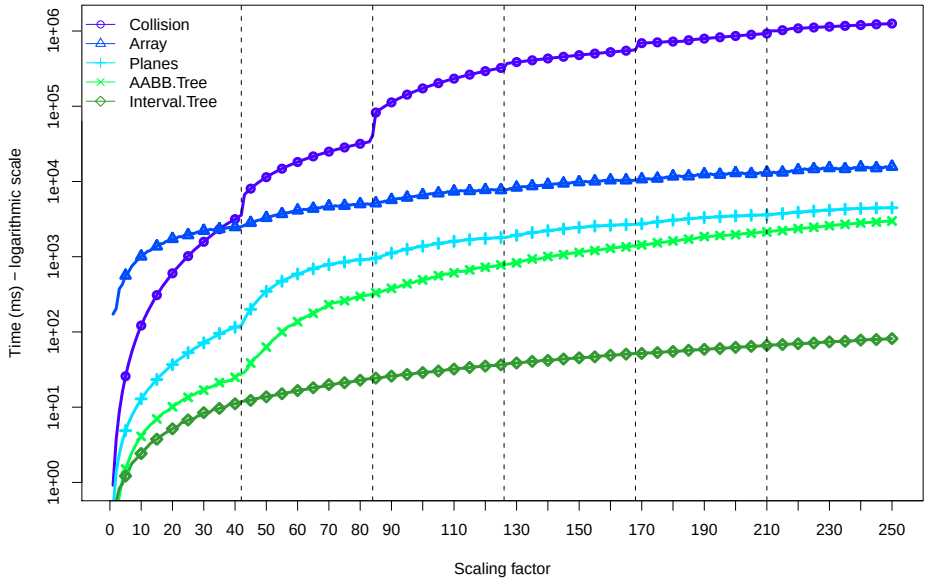

Figure 4.10: Graph showing the effect of scaling the problem instance size (logarithmic time scale).

add new gaps at higher positions within the skyline, whilst the interval tree implementation simply updates the gap at the higher position. As the number of box-placeable locations at a certain height, h (which we assume to be the lowest point in the skyline), approaches zero the time taken to evaluate fitness/validity of the boxes in all of the remaining lowest gaps will decrease. As the number decreases and boxes are being added, the number of gaps at height $h + 10$ will keep increasing. If the list of gaps at height h has been exhausted, then on the next iteration of the algorithm the skyline implementation will report the large list of gaps at height $h + 10$ as being at the lowest position and so the evaluation and placement process will take a much longer time again whilst the gaps at height $h + 10$ are being filled. The vertical dashed lines on the graph are at multiples of 42 as the number of boxes is always a multiple of 12, and $23 \times 22 = 506$ boxes will fill a 'layer' in the packing, therefore $\lfloor \frac{506}{12} \rfloor = 42$. The graph also illustrates what we have seen in Figures 4.8 and 4.9, reinforcing the idea that the planes method can only outperform the brute force collision detection method when packing exceptionally homogeneous datasets.

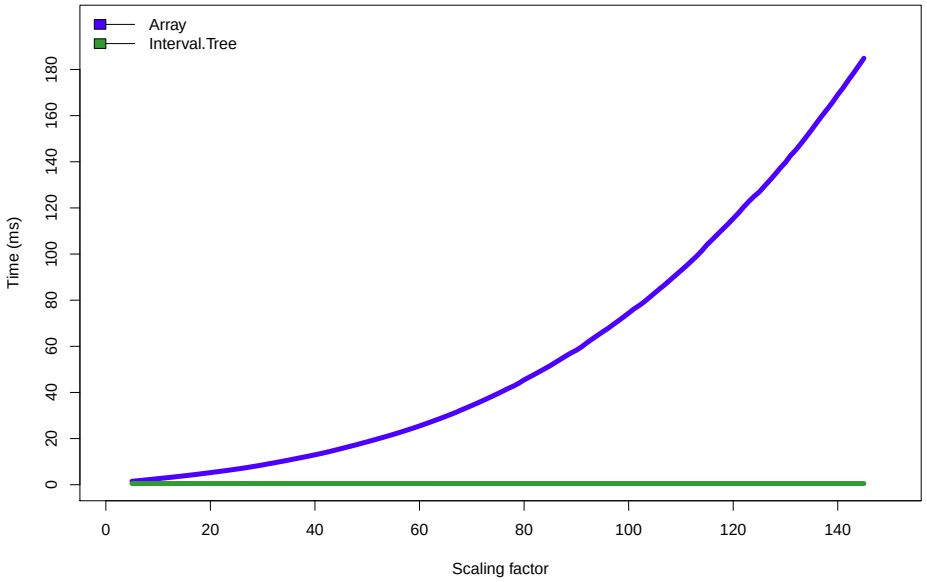

Figure 4.11: Graph showing the effect of scaling the box/container sizes.

4.4.3 Scaling the Container/Box Sizes

In order to ascertain the effect on run time of increasing the physical sizes of the container and boxes to be placed, an artificial dataset was created. A container of dimensions $9 \times 9 \times 9$ and 27 cube boxes of dimensions $3 \times 3 \times 3$ were packed by all of the ADT implementations. The dimensions were scaled from 1–150 and the runs repeated. The results are shown in Figure 4.11. Note that the slope (or lack thereof) of the interval tree line is indicative of the other experiments which are not shown for clarity. It can be seen that the array implementation is affected by an increase of the scale of the problem dimensions, as suggested by the theoretical result of Section 4.3.2, whereas the other methods are unaffected by this factor.

4.4.4 Improving the State-of-the-Art

We now show that using the interval-tree representation can significantly speed up non-trivial heuristics from the literature.

4.4.4.1 Container Loading

The current state-of-the-art heuristic for the container loading problem is FDA (fit degree algorithm) which is given in [HH11]. This consists of a single-pass constructive heuristic which

has 4 stages (summarised below), and an iterated improvement phase utilising local search. We use the constructive phase as the basis of the comparison.

1. Select a corner of a gap (or 'action space' in the parlance of the original paper) based on the following conditions, using a lexicographic ordering:
   (a) Distance from the nearest identically oriented corner of the container, smaller distance being better.
   (b) Volume of gap, larger is better.
   (c) Deepest-bottom-leftmost coordinate, smaller is better, i.e. closest to the origin of the container.
   (d) Furthest-highest-rightmost coordinate, smaller is better, i.e. closest to the origin of the container.
2. Select a box/group of boxes (called a block) to fit the gap based on the following conditions, using a lexicographic ordering:
   (a) Fit degree of the block, larger is better. This is a measure of how much a block is in contact with previously placed boxes.
   (b) Volume of an item in the block, larger is better.
   (c) Length of an item in the block, larger is better.
   (d) Length of the shortest side of an item in the block, larger is better.
   (e) Deepest-bottom-leftmost coordinate of the block, smaller is better.
   (f) Furthest-highest-rightmost coordinate of the block, smaller is better.
   (g) Orientation number, smaller is better. This is a number from 1–6 representing the orientation of a block.
3. Go to step 1 if there is no box placeable in the gap and select the next best gap.
4. Repeat 1–3 until all boxes have been packed or there are no more usable gaps.

The original timing test results are given as 'FDA' in Table 4.2 and the results using the interval-tree representation are given as 'FDA*.' The times given as 'FDA' in Table 4.2 have been scaled due to the differences in test computers and the result given for the entire dataset is given as opposed to a per-instance average. Results are given from the BR1–10 datasets used previously as well as the BR11–15 datasets (with 60, 70, 80, 90 and 100 unique box types per instance respectively). All 100 instances from each dataset are used in each experiment, and the average run times over 100 runs is given. The results are very strong and a significant speed up can be observed. The test computer (1.86 GHz Intel PC) used previously was used for the 'FDA*' results.

	Time (seconds)	
	FDA	FDA*
BR1	1.3	0.4
BR2	1.3	0.5
BR3	2.5	0.8
BR4	2.5	0.9
BR5	2.5	1.1
BR6	5.0	1.3
BR7	5.0	1.6
BR8	7.5	2.2
BR9	13.8	2.6
BR10	31.3	3.3
BR11	31.3	3.7
BR12	102.7	4.3
BR13	109.0	4.9
BR14	125.3	5.4
BR15	194.2	5.7

Table 4.2: Time taken to pack the BR datasets using the FDA algorithm with the interval-tree representation.

4.4.4.2 Strip Packing

The 3BF heuristic presented in chapter 3 and [ABK11] is used in our testing for the strip packing problem, and is implemented in the original paper using an AABB-tree representation. The heuristic is currently the best performing single-pass constructive strip packing heuristic for the three-dimensional strip packing problem.

Boxes are allowed to be rotated in this experiment (according to the restrictions put on them by the datasets). A post-processing stage referred to in [ABK11] as tower-processing is used, but takes less than 1% of the recorded runtime.

The original timing test results are given as '3BF' in Table 4.3 and the results using the interval-tree representation are given as '3BF*.' The average number of boxes per instance is given in the table as N. The first 10 instances from each dataset are used in each experiment, and the average run times over 100 runs is given. It is worth noting that the the AABB-tree representation in the original paper is a direct implementation of the data structure, and the implementation given here using interval-trees is called through an extra layer of abstraction (i.e. the Skyline ADT interface) and therefore requires some additional computational overhead. The results are still, however, very strong and a significant speed up can be observed. The same test computer (1.86 GHz Intel PC) was used in both sets of experiments.

		Time (seconds)	
	N	3BF	3BF*
BR1	139	0.3	0.2
BR2	140	0.4	0.2
BR3	135	0.5	0.4
BR4	132	0.6	0.4
BR5	128	0.7	0.5
BR6	134	0.9	0.6
BR7	129	1.1	0.7
BR8	138	1.3	1.1
BR9	128	1.6	1.2
BR10	129	2.0	1.5
BR1-XL	1000	8.6	1.4
BR2-XL	1000	10.2	1.6
BR3-XL	1000	12.5	2.0
BR4-XL	1000	14.9	2.4
BR5-XL	1000	16.8	3.0
BR6-XL	1000	19.4	3.3
BR7-XL	1000	22.9	3.9
BR8-XL	1000	26.2	5.7
BR9-XL	1000	30.1	6.8
BR10-XL	1000	35.3	8.4

Table 4.3: Time taken to pack the BR and BRXL datasets using the 3BF algorithm with the interval-tree representation.

4.5 Conclusions

Algorithms are only as efficient as the data structures that support them. In this chapter we have demonstrated a series of data structures for representing packings (and partial packings) created in a constructive manner and provided a uniform interface in the form of an abstract data type which allows these packings to be constructed effectively. The computational complexity has been given for a standard D-dimensional packing algorithm (the 3BF framework/heuristic) along with the complexity of several common data structures from the literature and one novel, efficient, implementation based on a small set of standard computational geometry methods. The performance of this new approach has been analysed theoretically and computationally and has been shown to be very efficient generally, especially in lower dimensions. The approach given in this chapter can be used for implementing both skyline type packing strategies (i.e. packing along one dimension in a linear fashion) and full packing strategies, where box placement is unrestricted within the container — though as we have seen, this increases the time complexity of the procedure. We have shown several fast methods for packing in higher dimensions, both with approximating the number of possible gaps and two methods (array and interval tree) for packing with all possible gaps. We have also demonstrated that existing

state-of-the-art heuristic packing strategies can be significantly sped up by using the improved problem representation data structures given in this chapter.

5

Automatically Generating Evaluation Functions for the 3BF Heuristic Framework

> On two occasions I have been asked,—"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" [...] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> *Passages from the Life of a Philosopher*
> Charles Babbage

Summary

In this chapter we explore techniques for automatically generating evaluation functions that can be used within the three-dimensional packing framework presented previously — the 3BF framework. An evaluation of genetic programming compared to other local search-based methods for generating evaluation functions is given (with statistical analysis) and the shortcomings highlighted. We then introduce some implementations of generating procedures that overcome these shortcomings. We show that automatically generating evaluation functions is a promising research direction and give computational results over some standard literature benchmarks, indicating that this method of generating evaluation functions for packing can be competitive with the state-of-the-art constructive heuristics.

5.1 Constructive Heuristics and the Strip Packing Problem

The work in this chapter focuses on automatically generating single-pass constructive heuristics and shows that automatically generating constructive heuristics can, in fact, produce practical packing methods.

The 3BF framework that we use for three-dimensional packing is presented in chapter 3. The framework supports a class of offline heuristic algorithms, i.e. at any stage in the packing the properties of all boxes, packed or unpacked, are known. The problem being tackled in this chapter is the strip packing problem (SPP), which according to the cutting and packing typology of Wäscher [WHS07] is classified as the 3D regular open-dimension-problem (ODP) with one open dimension.

The evaluation function, for generating a per box/gap pair placement score, used in Algorithm 1 is one that we generate using the methods explained in this chapter.

5.1.1 Representing Placement Scoring Functions

In the 3BF framework a scoring function is a function that takes a feasible box/gap pair and assigns a numerical value to it — an estimation of the fitness of this pair. The box/gap pair at each iteration of the algorithm with the perceived highest fitness is then chosen as the next step in the packing. As such, a scoring function coupled with the 3BF framework can be thought of as a three-dimensional packing heuristic.

Scoring functions which score box/gap pairs differently will generally generate different packings. If the ordering of all box/gap pairs in terms of fitness is the same between functions then, of course, the packings will be identical. This is formalised below.

Given two expressions $e_1, e_2 \in L$ where L is a language for representing arbitrary arithmetic expressions, two assignments of values to the variables in the expressions, represented as vectors a, a' such that $n = |a| = |a'|$, $a, a' \in \mathbb{R}^n$ and an evaluating function $f : (e, \mathbb{R}^n) \mapsto \mathbb{R}$ we say that the expressions have an equivalent order mapping iff:

$$f(e_1, a) \geq f(e_1, a') \Leftrightarrow f(e_2, a) \geq f(e_2, a') \quad \forall a, a' \in \mathbb{R}^n. \tag{5.1}$$

This can, of course, also be expressed as a Boolean pairwise preference relation: given two expressions $e_1, e_2 \in L$ as before, and the evaluating function $f' : (e, \mathbb{R}^n, \mathbb{R}^n) \mapsto \{0, 1\}$, and $a, a' \in \mathbb{R}^n$ as before, we say that the expressions have an equivalent order mapping iff:

$$f'(e_1, a, a') \Leftrightarrow f'(e_2, a, a') \quad \forall a, a' \in \mathbb{R}^n \tag{5.2}$$

where a value of 0 indicates that a' is preferred to a, i.e. a has a strictly lower rank position (and will therefore be chosen later in the case of the 3BF packing framework) than a', and 1 implies that a is ranked at least as highly as a'.

Note that the equivalent order mapping applies in the general case, for all possible assignments of values. Given, instead, a subset of all possible values (i.e. a single packing instance or dataset) it is possible that two functions may produce the same packings, despite not having an equivalent order mapping in the general sense.

In order to generate sensible evaluation/scoring functions for the 3BF framework, some specific information about the problem should be exploited, and a suitable representation employed that can structure the information and its interactions in a logical way. To this end, expression trees, or parse trees, are a reasonable representation to model such an arithmetic expression. This arithmetic expression defines the behaviour of the constructive heuristic as it is used to determine which box/gap pair should be chosen at each stage by assigning it a score (and the box/gap pair with the highest score is used in the packing). This method of automatically generating functions has been popularised by the increased interest in genetic programming (GP) [Koz92]. GP involves using a GA with a population of, in this case, parse trees representing scoring functions, and 'evolving' them through crossover and mutation, much the same as with a GA. The fitness of the parse tree is calculated by plugging it into an algorithm or framework (in this case the 3BF framework) and evaluating it against a given data set, or collection of data sets.

Parse trees obviously allow precedence of operations to be implicit in the structure of the tree. Leaves/external nodes represent variables and constants (or terminals in GP parlance). The internal nodes may represent unary operations (such as negation or log_2), binary operations (such as addition or subtraction), ternary (maximum value of three child nodes) or in general n-ary and are termed 'functions'. Examples of two general parse trees can be seen in Figure 5.1.

5.1.2 Experimental Set-up

We now take a state-of-the-art GP approach from the literature ([BHKW11]) and show that it can be improved to produce reasonable results. We start by generating a population of parse trees. The terminals and functions used in the parse trees are as shown in Table 5.1. The top-level algorithm manipulating the population of parse trees is a GA, effectively turning this parse tree generation method into a GP model. The parameters for the GA are given

in Table 5.2 and are taken, along with the terminals and functions, from the original work in [BHKW11]. The authors make no claims as to the optimality or robustness of these parameters, though the authors of the original work performed extensive computational testing in order to choose these parameters. Initial experimentation with other parameters of the GA for the strip packing problem presented here also resulted in no significant improvements or negative changes and therefore the parameters seem to be fairly robust on this occasion. The tree initialisation method is 'ramped half-and-half', having half of the initial population of randomly generated parse trees fully balanced and half allowed to be unbalanced. The total height of the parse tree is limited to 17. Throughout this chapter we use results based on experimentation on the first 10 instances of the BR5 dataset [BR95] which is sometimes referred to as the 'thpack5' dataset. The dataset is treated as a strip packing problem, ignoring the container length and ensuring all boxes are packed — not to be confused with the container loading problem which is similar in regards to the methods used for solving it but has slightly different constraints and objectives, see, for example, [Pis02]. As such, results from the container loading problem are not directly comparable to the ones given in this chapter. This dataset was chosen for the testing phase as it has a balance between heterogeneous and homogeneous packing problems, with 98–142 boxes per instance and 12 unique box types per instance. Each instance assumes a maximum possible volume utilisation of 100%, i.e. assumes that the boxes can be positioned in some way such that there are no gaps within the container, and therefore each run is scored out of 1000 (10 instances, maximum of 100 for each). It is worth noting that the optimal solution for these instances is as yet unknown, but that they were not generated as to be perfectly packable (without gaps) and therefore a score of 100% is unlikely to be feasible. The volume utilisation calculation is calculated by the volume lower bound divided by the used length of the container. The volume lower bound is as given previously in chapter 3. Each experiment is run 100 times and the best found result from each run used in the analysis.

Figure 5.1: General arithmetic expressions as parse trees, (left) $x \cdot y + z$, (right) $\frac{a-cos(b)}{\sqrt{|c|}}$.

5.1.3 Initial results

Using the initial set-up of the experiment the results gave a mean (± standard deviation) of 775.36 ± 2.54. This result will be used throughout the chapter as a reference point. For the purposes of comparison an identical number (i.e. 50000) of parse trees were generated at random

Node	Arity	Description
$+$	Binary	Addition of child nodes
$-$	Binary	Subtraction of right child node from left child node
$*$	Binary	Multiplication of child nodes
$/$	Binary	Division of left child node by right child node (or division by 0.001 if right child evaluates to zero)
v	Nullary	Volume of box, $\prod_{k=1}^{3} L_i^k$
w	Nullary	Weight (profit) of box
x	Nullary	x coordinate of box, x_i^1
y	Nullary	y coordinate of box, x_i^2
z	Nullary	z coordinate of box, x_i^3
$f_{XYWaste}$	Nullary	Sum of differences between the box's and gap's width and length, $(L_j'^1 - L_i^1) + (L_j'^3 - L_i^3)$
$f_{XZWaste}$	Nullary	As above, but width and height, $L^{(\prime)1}$ and $L^{(\prime)2}$
$f_{YZWaste}$	Nullary	As above, but length and height, $L^{(\prime)3}$ and $L^{(\prime)2}$

Table 5.1: Function and Terminal nodes used in experimental set-up (for box i and gap j).

Population size	1000
Generations	50
P(Crossover)	0.85
P(Mutation)	0.1
P(Reproduction)	0.05
Selection method	Tournament, size 7
Mutation method	'Grow' method [Koz92] to regenerate a random sub-tree

Table 5.2: GA parameters from [BHKW11].

for each of 100 runs and the best result from each run used. The random generation runs gave an average best result of 777.18 ± 1.44 which, using a two-tailed Welch t-test, gives a p-value of 4.02×10^{-9} indicating that the GA search method is significantly outperformed by simply generating and evaluating the same number of random parse trees. This, of course, should not be the case if a sensible search methodology is used.

5.2 Improving the Problem Representation

We now discuss several ways of improving the initial experimental set-up to gain increases in solution qualities over time.

5.2.1 Packing and Partial Packing Data Structure

One of the biggest disadvantages of the current problem representation (that is, the plane-representation described in section 4.3.4) with regards to its handling of packings and sub-packings is that it disallows larger boxes to be placed next to smaller boxes. We refer to this restriction as the plane restriction. Figure 5.2 shows two simple unique parse trees for the equivalent two-dimensional problem. Figure 5.4 shows the packing procedure followed when taking into account the plane restriction and Figure 5.3 shows the resulting packings, first with the plane restriction and second/third when this restriction is lifted.

The examples given show that the initial packing representation may lead to identical solutions from separate parse trees, as well as potentially creating inefficient solutions. In order to lift the plane restriction, the packing representation was replaced with the interval-tree representation given in chapter 4. Ultimately, the representation is much the same as the plane representation but with no restriction against larger boxes being placed around smaller boxes (and a significant decrease in the runtime cost, though this issue is not examined within the scope of this chapter as it has already been covered previously). After running the experiments again (100 runs, best result from each run used) the mean is increased from 775.36 ± 2.54 to 815.33 ± 5.58. The t-test now gives a p-value $< 2.2 \times 10^{-16}$, showing that, with probability greater than 0.9999, lifting the plane restriction increases the solution quality within the experimental parameters.

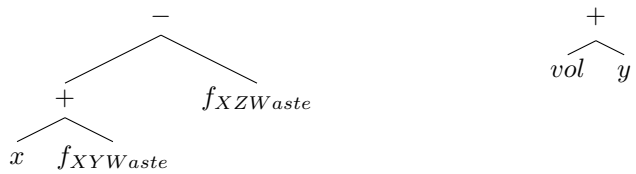

Figure 5.2: Two distinct parse trees.

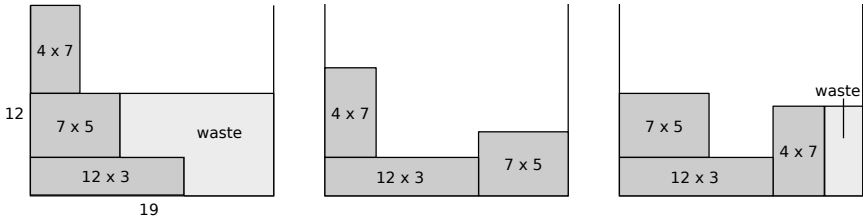

Figure 5.3: Result of using (left) either tree, with the plane restriction, (centre) first tree with no plane restriction, (right) second tree with no plane restriction.

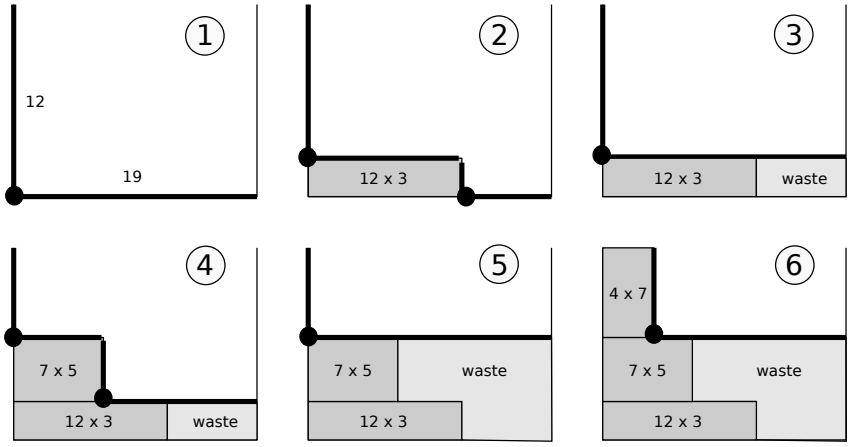

Figure 5.4: Packing steps for both parse trees in Figure 5.2. Packing three boxes in a 19x12 container, with the plane restriction.

5.2.2 Terminals

The next change to the experiments involves replacing the originally proposed nodes in the parse trees to more carefully considered nodes, allowing more problem-specific information to be exploited in the generated heuristic functions. The full list of nodes used in this experimental set-up can be seen in Table 5.3. The new nodes are explained in the table and the two new 'function' nodes which take into account the current solution state are f_{RF} and f_{CS}, meaning Rotation Flexibility and Contact Score respectively. During initial testing other nodes were considered, e.g. log_2, $sine()$, the ration between box/container lengths etc. but were found not to be beneficial to the results, and were not found in the final best solutions. The nodes shown in Table 5.3 are sufficient to reproduce the heuristics in chapter 3 given a well structured parse tree, or can at least approximate the behaviours with fairly simple parse trees.

Rotation flexibility is a measure of how well a box can be rotated. A higher score indicates that the box can be rotated in to a higher number of unique orientations. Obviously this is a score from between one and six. If a box, for example, has two sides the same length then only three valid rotations are possible, i.e. a score of three. A cube has one valid rotation and therefore a score of one. This score also takes into account rotation restrictions defined in the datasets, e.g. in the BR datasets used here some boxes have between 2–4 orientations disallowed.

Contact score is a measurement of how well a placed box would fit in with its neighbouring boxes. The sum of all the areas of the surface of the box that would be in contact with either an already placed box or the container is used for this score. In practise this can mean that similarly proportioned boxes tend to be grouped together as they have the highest amount of surface area in common with each other. This can be seen as the primary component of the $MaximumContact$ placement heuristic in chapter 3.

The terminal changes again improved the results: the quality of the best solutions over 100 runs was 777.80 ± 2.36, up from the initial point of 775.36 ± 2.54. With a p-value from the t-test of 3.1×10^{-11} this indicates a difference, with probability > 0.9999.

5.3 Improving the Search Methodology

5.3.1 Hill Climbing

GA's are traditionally a powerful optimisation technique when a problem can be represented in a logical way in which 'crossing' two good (in terms of a specific evaluation function) parent solutions leads to a good child solution. This is often the case when the individual symbols from the genome string represent, e.g., separate dimensions in the problem space, and are independent. On the other hand, crossing two individuals when the genome represents, e.g. a permutation (where the structure/order of the symbols are crucial to the quality of the

Node	Arity	Description
$+$	Binary	Addition of child nodes
$-$	Binary	Subtraction of right child node from left child node
$*$	Binary	Multiplication of child nodes
$double$	Unary	Doubles the value of child node
$tenTimes$	Unary	Multiplies the value of the child node by ten
$halve$	Unary	Halves the value of the child node
$invert$	Unary	$\frac{1}{c+1}$ where c is the value of the child node
1	Nullary	The value 1
v	Nullary	Volume of box, $\prod_{k=1}^{3} L_i^k$
x	Nullary	x coordinate of box, x_i^1
y	Nullary	y coordinate of box, x_i^2
z	Nullary	z coordinate of box, x_i^3
$xyArea$	Nullary	The area of the lower face of the box, width multiplied by length
$xzArea$	Nullary	The area of the back face of the box, width multiplied by height
$yzArea$	Nullary	The area of the side face of the box, length multiplied by height
$xSize$	Nullary	Width of box
$ySize$	Nullary	Length of box
$zSize$	Nullary	Height of box
$yMax$	Nullary	y coordinate plus length of box
f_{CS}	Nullary	Contact score
f_{RF}	Nullary	The rotation flexibility of the box

Table 5.3: Function and Terminal nodes used in experimental set-up (for box i).

solution), is unlikely to capture the successful properties of both parents and is more likely to lose the strength of at least one of the parents in the process. This is similar to the parse tree representation of a mathematical function, such as in GP, in that crossing two functions will not necessarily combine the properties of either of the functions from either parent tree in any sensible way. To this end it was thought to try manipulating the parse trees in a different manner.

Instead of using a GA to work on populations of trees, a simple hill climbing approach was taken in the next set of experiments. The hill climber was set to randomly sample 1000 possible moves in the neighbourhood at each iteration and select the move with the highest evaluation score. The number of iterations was set to 50 so as to match with the number of generations of the GA approach. It is obvious, therefore, that the same number of fitness evaluations takes place in each approach and the two approaches can be considered together in a fair comparison. These parameters were chosen so that a per-generation comparison can be made with the GA approach, and the convergence properties of each can be identified.

5.3.2 Neighbourhood Moves

As well as the changes stated in Section 5.3.1 the mutation operator was changed as per the following: choosing the 'grow' operation as before with probability 0.25 or a random switch

operation otherwise (i.e. with probability 0.75). The switch operation takes a random node in the tree and replaces it with a different node of the same arity. This allows the search to be more incremental, making smaller jumps around the search space (giving, through initial testing, solutions with around 5-10% improvement on using the grow operator alone).

The quality of the best solutions over 100 runs was 777.47 ± 2.10, up from the initial point of 775.36 ± 2.54 with a p-value of 1.215×10^{-9}. Again, this indicates a difference with probability > 0.9999.

5.4 Putting it all Together — Results

The final set of experiments utilised all of the improvements shown previously to test whether they would give better results than any of the individual improvements. That is, the improved operators were included along with the improved problem representation and the hill climbing search methodology. The results are shown in Figure 5.5 and Table 5.4. It should be noted that the authors do not claim that the improvements, combinations of improvements or parameteres used are optimal — merely that there exists a large scope for improvement over the current methodology. In order to ascertain such stronger assertions a full combinatorial experimental design would be required as well as extensive parameter-tuning, which rarely in practice increases the algorithmic results in a significant way.

As can be seen from the box plot in Figure 5.5, using all of the improvements does indeed give a better solution than any of the improvements individually. The mean is 869.60 ± 3.43 and naturally this leads to a p-value, when compared to any individual change or the initial set-up, of $< 2.2 \times 10^{-16}$, again showing that with a very high probability it can be assumed that these improvements combined do make a statistical difference.

The solution quality over time/per generation can be seen in Figures 5.6 and 5.7. The graphs represent the solution quality of the fittest member of the population at each generation, averaged over the 100 runs.

	Min.	Max.	Mean	Median	Standard Deviation
Base code	765.27	780.72	775.36	775.53	2.54
Random trees	768.65	779.58	777.18	777.44	1.44
Interval Tree representation	803.36	829.97	815.33	815.26	5.58
Improved Terminals	773.43	791.32	777.80	779.02	2.36
Hill Climbing	771.80	783.61	777.47	777.60	2.10
All Improvements	862.37	879.39	869.60	869.15	3.43

Table 5.4: Final solution quality of each method after 100 runs.

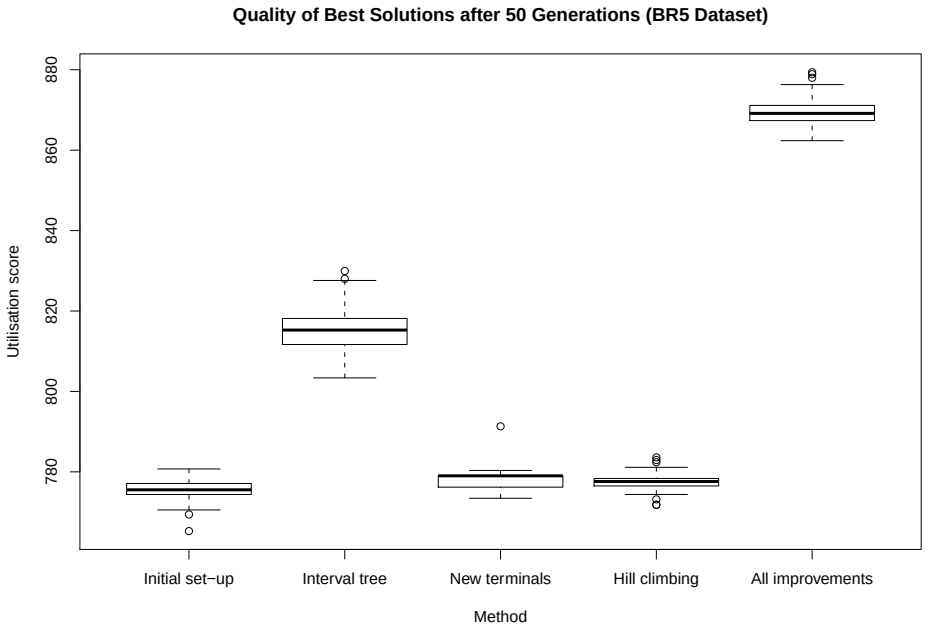

Figure 5.5: Box plot showing the results of all experiments on the BR5 dataset.

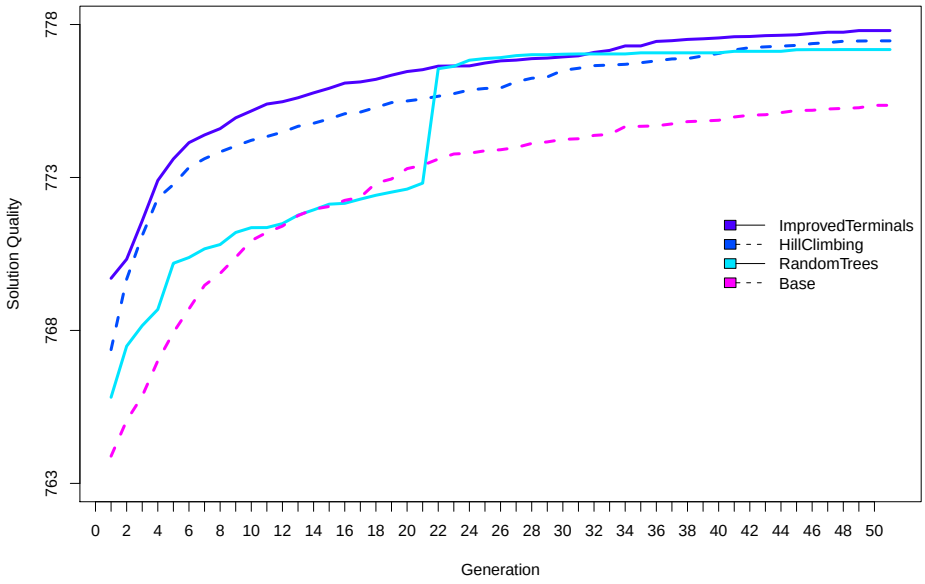

Figure 5.6: Mean solution quality over time/per generation of the base code, random, hill climbing and improved terminals methods.

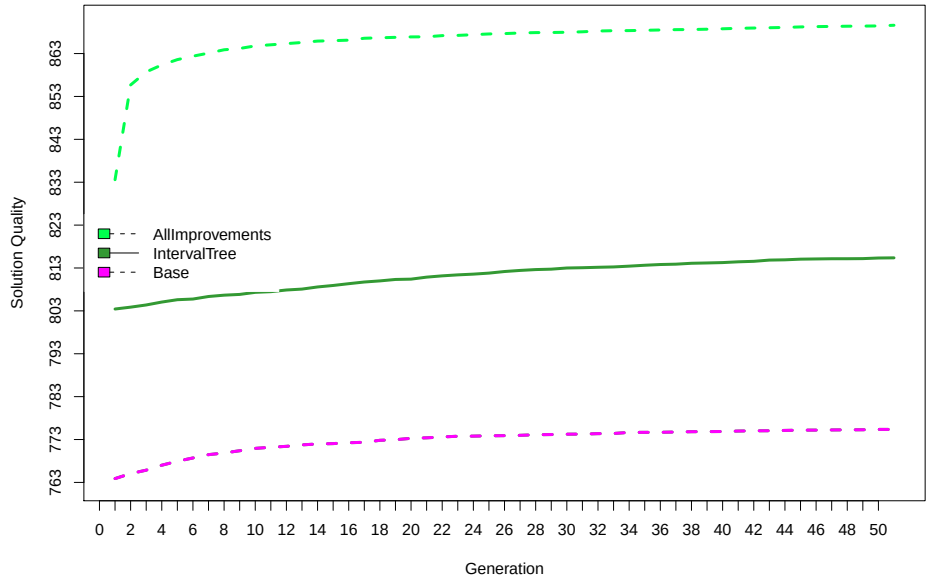

Figure 5.7: Mean solution quality over time/per generation of the base code, interval tree representation and 'all improvements' methods.

5.4.1 Comments on Generated Functions

All 100 final trees were visually inspected and analysed. Visualisations of the trees can be found at `http://www.cs.nott.ac.uk/~sda/research/alltrees.pdf` and a sample can be found in Figure 5.8.

Although the height of the generated trees was (for all intents and purposes) unbounded, the best trees were all relatively short. The range of heights for the best trees generated was $3-12$, with a mean of 5.86 ± 1.67. Information about the number of trees that each node appeared in (tree count) and the total number of times that each node appeared across all of the best trees can be found in Table 5.5. It can be seen that the terminal nodes f_{CS} and $xzArea$ are the most frequently occurring individual nodes in the trees — appearing in 99 and 97 of the final trees respectively. In fact, the two nodes appear together in 96 of the 100 final trees, dominating any other pairwise combination of terminals.

As is often the case with automatically generated heuristics, analysis of the functionality is particularly difficult due to the level of abstraction between the generated tree/function and the behaviour of the heuristic (and subsequent solution produced). Usually one of the reasons for this is the size and complexity of the generated tree, but obviously in the case of generating relatively small trees, such as this example, this should not be so much of an issue. Inferring behaviour of the heuristic from the trees alone is non-trivial, however. For example, the two

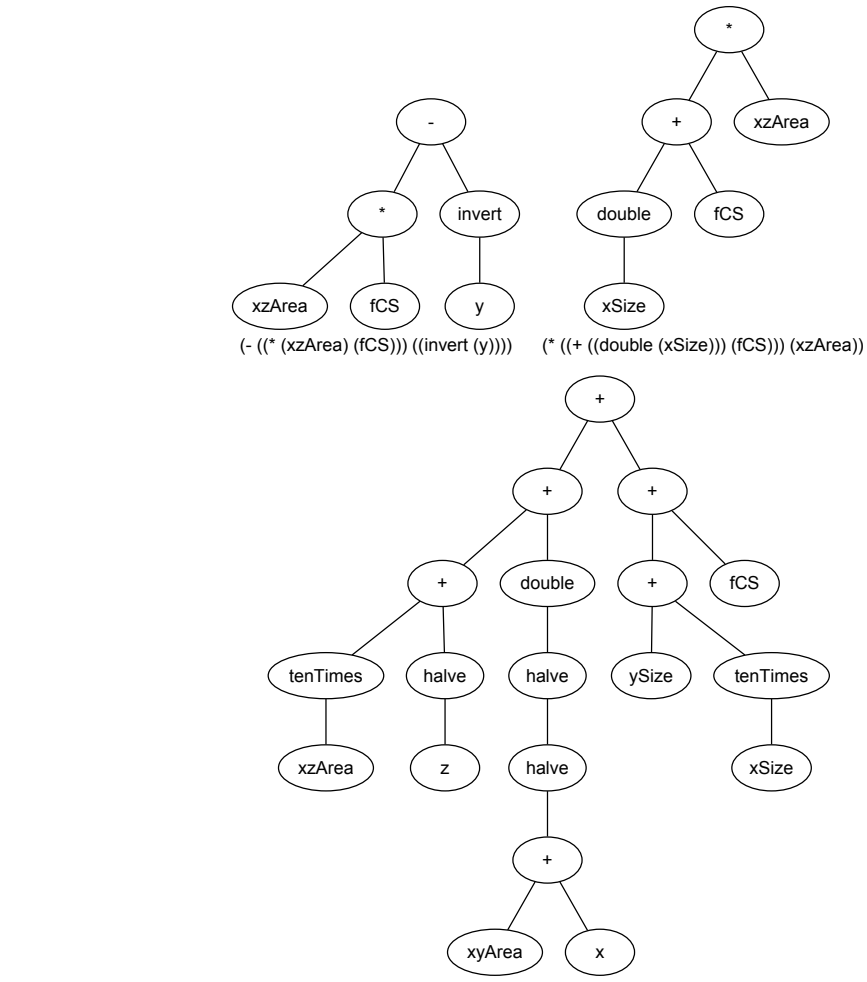

Figure 5.8: Some of the automatically generated best trees.

trees in Figure 5.9 are functionally different (i.e. the result returned by one is always a constant factor of the other) though the packings produced by them will always be the same. On the other hand, the trees in Figure 5.10 are, by any standard tree-distance measure, very similar to each other, whereas the packings they produce may be wildly different.

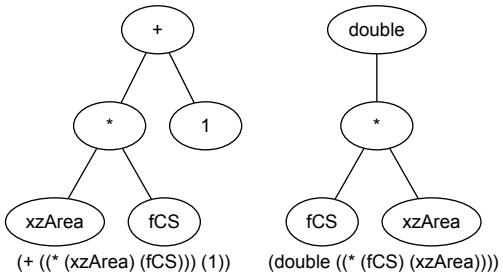

Figure 5.9: Two trees which are visually dissimilar but equivalent within the proposed heuristic framework. Automatically generated by search methods.

5.4.2 Comparison with the State-of-the-Art

We now take the 100 generated functions and apply them to a wider range of datasets from the literature and compare them to the current state-of-the-art in constructive heuristics.

The 3BF heuristic presented in chapter 3 is used in our testing. The heuristic is currently the best performing single-pass constructive strip-packing heuristic for the three-dimensional strip-packing problem.

The results in Table 5.6 show the solution qualities of the generated heuristics on the first 10 instances of each of the Bischoff-Ratcliff (BR) datasets presented in [BR95]. There are approximately 90-150 boxes to pack in each instance of the BR datasets. The number of distinct box types in each of BR1-BR10 is 3, 5, 8, 10, 12, 15, 20, 30, 40 and 50 respectively. As such the

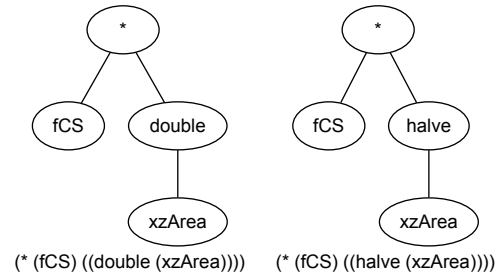

Figure 5.10: Two trees which are visually similar but very dissimilar within the proposed heuristic framework. Manually created.

Node	Arity	Tree count	Frequency
$+$	Binary	90	2.22
$-$	Binary	47	0.76
$*$	Binary	80	1.06
$double$	Unary	54	0.70
$tenTimes$	Unary	55	0.73
$halve$	Unary	45	0.65
$invert$	Unary	29	0.38
1	Nullary	21	0.26
v	Nullary	12	0.12
x	Nullary	18	0.21
y	Nullary	15	0.15
z	Nullary	20	0.21
$xyArea$	Nullary	12	0.13
$xzArea$	Nullary	97	1.06
$yzArea$	Nullary	17	0.17
$xSize$	Nullary	50	0.54
$ySize$	Nullary	22	0.23
$zSize$	Nullary	34	0.41
$yMax$	Nullary	25	0.28
f_{CS}	Nullary	99	1.11
f_{RF}	Nullary	14	0.15

Table 5.5: Function and terminal nodes' frequency in 100 best evolved trees.

homogeneity of the datasets decreases (or the heterogeneity increases). The implementation of the packing algorithm only checks each box type (not each box individually) to see if it will fit in the lowest gaps at each iteration and for its evaluation score, naturally leading to more comparisons as the homogeneity of the datasets decrease. Similarly, the results with the postfixed '-XL' are from the first 10 instances of each of the extended datasets (BRXL), described in [BM07], where each of the original BR instances are scaled to contain 1000 boxes to pack. The number of box types increases identically to the BR datasets.

It can be seen from Table 5.6 that the results are very competitive. The ratio between the best performing automatically generated heuristic and the 3BF heuristic is consistently between 0.97 and 1.01 for each dataset and even the ratio between the median automatically generated heuristic and 3BF remains between 0.94 and 0.98. This shows that the heuristics generated in this chapter are at least competitive with the expert created, state-of-the-art — and in some cases can even outperform it by a small margin. Of course, if the generating procedures were to be run on the individual instances or datasets then an improvement could be observed.

	Min	Max	Mean	Median	Std. Dev.	3BF	Median/ 3BF	Max/ 3BF
BR1	79.4	85.7	83.8	84.3	1.9	88.3	0.95	0.97
BR2	76.4	85.6	81.2	83.8	4.0	88.5	0.95	0.97
BR3	73.8	85.2	79.5	82.8	4.8	87.0	0.95	0.98
BR4	74.1	86.7	79.7	82.6	4.8	87.1	0.95	1.00
BR5	73.3	86.8	79.3	82.9	5.1	86.0	0.96	1.01
BR6	73.5	83.7	78.8	81.8	4.5	86.3	0.95	0.97
BR7	72.0	84.7	78.6	82.9	5.7	86.6	0.96	0.98
BR8	72.4	84.3	78.6	82.5	5.3	86.2	0.96	0.98
BR9	71.7	85.0	78.4	82.3	5.7	85.1	0.97	1.00
BR10	73.1	85.5	79.1	82.4	5.2	85.4	0.96	1.00
BR1-XL	83.9	90.5	87.2	87.9	2.5	93.1	0.94	0.97
BR2-XL	83.3	90.8	86.7	88.7	2.9	92.0	0.96	0.99
BR3-XL	81.8	91.5	85.9	87.9	3.5	91.4	0.96	1.00
BR4-XL	82.1	92.0	86.1	88.0	3.5	91.2	0.96	1.01
BR5-XL	81.9	91.7	86.5	88.7	4.0	91.9	0.97	1.00
BR6-XL	81.6	92.0	86.4	89.2	4.1	91.9	0.97	1.00
BR7-XL	81.9	92.4	86.6	89.4	4.1	91.7	0.97	1.01
BR8-XL	81.7	92.7	86.7	89.5	4.3	91.4	0.98	1.01
BR9-XL	81.4	92.4	86.9	89.9	4.7	91.7	0.98	1.01
BR10-XL	80.8	92.7	86.6	90.1	5.1	91.6	0.98	1.01

Table 5.6: Results of the generated functions against the 3BF heuristic across the BR1 – BR10 and BR1-XL – BR10-XL datasets.

5.5 Conclusions

This chapter has shown that, though the field of automatically generating heuristics for packing is a relatively young one, it can be fairly successful given the correct 'building blocks' and search methodology, supported by a suitable problem representation. The heuristic scoring functions presented here have been shown to fit in with the 3BF framework well, providing the power and flexibility of using different scoring functions and specialised problem representations within the same framework, which can be altered and tailored to fit specific problems. Future work in this area on implementing better search methods (e.g. simulated annealing, tabu search etc.), better neighbourhood moves and more intelligent parse tree nodes for specific packing problems could produce even better solutions. The techniques for improving standard approaches given here can most likely be applied to similar areas of automatically constructing heuristics for optimisation in wider fields. Ultimately, given enough domain-specific knowledge and enough time to automatically generate procedures it is foreseeable that automatically generating heuristics for cutting and packing can be human competitive.

6

Integer Programming Models for Solving Three-Dimensional Packing Problems

Essentially, all models are wrong, but some are useful.

Empirical Model-Building and Response Surfaces
George E. P. Box

Summary

There are a number of heuristic methods and solvers for multi-dimensional packing, as we have seen in previous chapters, but the progress in exact solvers, in general, and integer programming solvers, in particular, has been limited. Padberg [Math. Methods Oper. Res., 52(1):1–21, 2000] estimated an integer linear programming formulation of Fasano (the current state-of-the-art) could cope with "about twenty boxes." Our computational experiments confirm this hypothesis, and show that the seemingly trivial decision of whether twelve unit-cubes can be packed into a box with a near-unit-base and height eleven, cannot be made within an hour by modern integer programming solvers using this formulation.

We present a new, "space-indexed," linear programming relaxation, which often provides lower bounds within 1 percent of optimality, and allows for integer optima to be computed on instances with many thousands of boxes within an hour. Results of extensive computational tests of both formulations are reported.

6.1 Introduction

Although there are a number of heuristic solvers of varying quality, the progress in exact solvers for the Container Loading Problem has been limited, so far. Branch-and-price-and-cut solvers

exist for realistic models of transportation considering both weight and volume of vehicle load [GILM06, Lüb10]. However, one needs to solve the van loading problem (a variant of the CLP, explained in section 6.3) to optimality as the pricing sub-problem for these models.

Fasano [Fas99] proposed a mixed integer linear programming (MIP) formulation for the CLP, but did not provide comprehensive computational results. The model was very similar to that of Chen [CLS95] who used it for a multiple container loading problem, whereas Fasano applied it to a single container loading problem. Chen was only able to pack a maximum of 6 boxes using the formulation. Padberg [Pad00] suggested the MIP formulation of Fasano could cope with "about twenty boxes," but without providing any computational results. We confirm these limits using ILOG CPLEX, Gurobi, and SCIP in Section 6.5. This observation motivates the rest of the chapter, where we present a new space-indexed formulation providing strong linear programming relaxations. Integer optima for instances of over 100000 boxes can be computed within an hour. Throughout this chapter, we use the notation as described in Table 6.1.

Symbol	Meaning
N	The number of boxes
h	A fixed axis, in the set $\{x, y, z\}$
a	An axis of a box, in the set $\{1, 2, 3\}$
x_i^h	The coordinate of the centre of box i along the h-axis
L_i^a	The length of axis a of box i
l_i^a	The length of axis a of box i halved
L_0^h	The length of axis h of the container
W_i	The volume of box i, in the CLP

Table 6.1: Notation used.

6.2 Related Work

Fasano [Fas99] introduced an integer linear programming formulation using the relative placement indicator:

$$\lambda_{ij}^h = \begin{cases} 1 & \text{if box } i \text{ precedes box } j \text{ along axis } h \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

$$\delta_{ai}^h = \begin{cases} 1 & \text{if box } i \text{ is rotated so that axis } a \text{ is parallel to fixed axis } h \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

The full model, as given by Fasano/Padberg is as follows:

$$\text{Maximise} \sum_{i=1}^{N} \sum_{h} W_i \delta_{1i}^h \tag{6.3}$$

$$s.t.$$

$$\sum_h \delta_{2i}^h = \sum_h \delta_{1i}^h = \sum_a \delta_{ai}^h \tag{6.4}$$

$$L_{j(i)}^1 \lambda_{j(i)i}^h + \sum_a l_i^a \delta_{ai}^h \leq x_i^h \leq \sum_a (L_0^h - l_i^a)\delta_{ai}^h - L_{j(i)}^1 \lambda_{ij(i)}^h \tag{6.5}$$

$$L_0^h \lambda_{ji}^h + \sum_a l_i^a \delta_{ai}^h - \sum_a (L_0^h - l_j^a)\delta_{aj}^h \leq x_i^h - x_j^h$$

$$\leq \sum_a (L_0^h - l_i^a)\delta_{ai}^h - \sum_a l_j^a \delta_{aj}^h - L_0^h \lambda_{ij}^h \tag{6.6}$$

$$\sum_h (\lambda_{ij}^h + \lambda_{ji}^h) \leq \sum_h \delta_{1i}^h, \sum_h (\lambda_{ij}^h + \lambda_{ji}^h) \leq \sum_h \delta_{1j}^h \tag{6.7}$$

$$\sum_h \delta_{1i}^h + \sum_h \delta_{1j}^h \leq 1 + \sum_h (\lambda_{ij}^h + \lambda_{ji}^h) \tag{6.8}$$

$$\sum_{i=1}^{n} \sum_h \left(\prod_a L_i^a \right) \delta_{1i}^h \leq \prod_h L_0^h \tag{6.9}$$

$$\delta_{ai}^h, \lambda_{ij}^h \in \{0, 1\} \tag{6.10}$$

$$L_i^1 \leq L_i^2 \leq L_i^3 \tag{6.11}$$

$$l_{ai} = \frac{1}{2} L_{ai} \tag{6.12}$$

$$j(i) \text{ such that } L_{j(i)}^1 = max\{L_j^1\} \text{ for } 1 \leq i \neq j \leq N. \tag{6.13}$$

Constraint 6.4 enforces the orthogonality constraint — if a box is placed then each axis is aligned to exactly one axis of the container and vice versa. Constraint 6.5 is the domain constraint, i.e. all boxes must lie fully within the container, or not be placed at all. Constraints 6.6, 6.7 and 6.8 represent the non-intersection constraints, that is that no packed boxes may overlap each other in all dimensions. Constraint 6.9 is the knapsack constraint, i.e. there cannot be more volume filled than is available in the container.

Padberg [Pad00] has unified some of the variables and studied features of the model. In particular, he identified the subsets of constraints with the integer property. Despite the interesting theoretical properties, modern integer programming solvers fail to solve instances larger than 10–20 boxes using this formulation, as evidenced in Table 6.2. This is far from satisfactory.

	Time (s)		
	Gurobi 4.0	CPLEX 12.2.0	SCIP 2.0.1 + CLP
Pigeon-01	< 1	< 1	< 1
Pigeon-02	< 1	< 1	< 1
Pigeon-03	< 1	< 1	< 1
Pigeon-04	< 1	< 1	< 1
Pigeon-05	< 1	< 1	3.3
Pigeon-06	< 1	< 1	37.9
Pigeon-07	1.5	3.6	779.3
Pigeon-08	7.4	25.6	–
Pigeon-09	88.6	398.4	–
Pigeon-10	1381.4	–	–
Pigeon-11	–	–	–
Pigeon-12	–	–	–

Table 6.2: The performance of various solvers on 3D Pigeonhole Problem instances encoded in the Fasano/Padberg formulation. "–" denotes that optimality of the incumbent solution has not been proven within an hour.

6.2.1 Extensions to the Formulation of Fasano/Padberg

Arguably, the formulation could be strengthened by the addition of further valid constraints. We have identified three classes of such constraints. Compound constraints stop combinations of certain boxes being placed next to each other if they would violate the domain constraint. For example:

$$\delta^h_{ki} + \delta^h_{mj} + \lambda^h_{ij} \leq 2 \quad \text{if } l^k_i + l^m_j > L^h_0 \quad \forall k, m \in \{1,2,3\}, h \in \{x,y,z\} \\ \text{and } 1 \leq i \neq j \leq n \tag{6.14}$$

The example for 2 boxes can be easily extended to 3 or more boxes "in a row."

One can also attempt to break symmetries in the problem. If boxes i and j (where $i < j$) are identical (i.e. $L^a_i = L^a_j \ \forall a$):

$$\lambda^h_{ij} = 0 \text{ and } \sum_h \delta^h_{1i} \geq \sum_h \delta^h_{1j} \tag{6.15}$$

Furthermore, if a box has two or more sides of the same length then we can limit the rotations. We define function f, which provides a canonical mapping of $\{x,y,z\}$ to $\{1,2,3\}$:

$$\sum_h (f(h) \cdot \delta^h_{ki}) \geq \sum_h (f(h) \cdot \delta^h_{mi}) \quad \text{if } l^k_i = l^m_i \quad \forall 1 \leq i \leq N \\ \text{and } 1 \leq k < m \leq 3 \tag{6.16}$$

6.2.2 Box Grouping

If we 'merge together' boxes that share an identical face then we can create a larger 'compound box,' which helps to generate higher volume utilisations earlier in the MIP search procedure in order to aid the bounding.

If boxes i and j share two identical side lengths, e.g. $L_{1i} = L_{1j}$ and $L_{2i} = L_{2j}$ then a virtual compound box, b, can be created with dimensions L_{1i}, L_{2i} and $L_{3i} + L_{3j}$. This applies to any combination of the dimensions of i and j. The obvious constraints follow:

$$\sum_H \delta_{1i}^H + \sum_H \delta_{1b}^H \leq 1 \tag{6.17}$$

$$\sum_H \delta_{1j}^H + \sum_H \delta_{1b}^H \leq 1 \tag{6.18}$$

Of course, compound boxes can be made up of more than two boxes, and in fact can be recursively built out of boxes and compound boxes so long as the constraints reflect this. Naturally, the problem becomes much larger and the time taken to generate the virtual boxes gets significantly longer for each level of recursion taken to group boxes together.

6.2.3 Over-zealous Symmetry-Breaking for Heuristic Bounds

It was found that if the problem was over-constrained such that some positioning precedences were enforced then good lower bounds could be found on the (maximisation) problem. The 'true optimum' was often cut off, but the bounds were found to help the MIP tree search nonetheless. Each combination of pairs of constraints (6.19 – 6.21) and then also all three constraints were applied, in turn, early in the search procedure, solved for up to 5 minutes and then relaxed again, to help with the bounding.

$$\lambda_{ij}^1 = 0 \quad \forall 1 \leq i < j \leq n \tag{6.19}$$

$$\lambda_{ij}^2 = 0 \quad \forall 1 \leq i < j \leq n \tag{6.20}$$

$$\lambda_{ij}^3 = 0 \quad \forall 1 \leq i < j \leq n \tag{6.21}$$

Nevertheless, whilst the addition of these constraints and added heuristic bounds improves the performance somewhat (in testing the time taken to solve solutions which were solved to optimality within one hour was reduced by, on average, 30% — problems that could not be solved optimally within the hour with the original model could also not be solved optimally with the improved models), the formulation remains impractical for many instances.

6.3 The Van Loading Problem

We now describe an extension to the Container Loading Problem which we call the Van Loading Problem (VLP). As with the CLP we are to maximise the profit of packing a single three-dimensional container of a fixed size, whereas in a slight deviation to the problem description given in section 2.4.1 the profit (i.e. knapsack weight) of each box to be packed is independent of the volume of the box. We also introduce an extra property of the boxes, M, which represents the mass ('mass' is used to disambiguate with the weight property in the 'classical definition' of the knapsack problem) which is also independent of the volume of the boxes. The container has a payload capacity, Q, and an accompanying constraint which represents the maximum summed mass of the boxes allowed in the container. This is shown in constraint 6.22.

$$\sum_{i=1}^{N} \sum_{h} M_i \delta_{1i}^{h} \leq Q \tag{6.22}$$

6.4 A Space-Indexed Formulation

We hence propose a new formulation, where the large box is discretised into units of space, whose dimensions are given by the largest common denominator of the respective dimensions of the small boxes. The small boxes are grouped by their dimensions into types $t \in \{1, 2, \ldots, n\}$. That is, boxes with identical dimensions and identical allowed rotations are assigned a single type. A_t is the number of boxes of type t available. The formulation uses the space-indexed binary variable:

$$\mu_{x,y,z}^{t} = \begin{cases} 1 & \text{if a box of type } t \text{ is placed such that its lower-back-left} \\ & \text{vertex is at coordinates } x, y, z \\ 0 & \text{otherwise} \end{cases} \tag{6.23}$$

Without allowing for rotations, the formulations reads:

$$\text{Maximise} \quad \sum_{x,y,z,t} \mu^t_{xyz} W_t \tag{6.24}$$

s.t.

$$\sum_t \mu^t_{xyz} \leq 1 \qquad \forall x, y, z \tag{6.25}$$

$$\mu^t_{xyz} = 0 \qquad \forall x, y, z, t \text{ where } x + L^1_t > L^1_0 \text{ or } y + L^2_t > L^2_0 \text{ or } z + L^3_t > L^3_0 \tag{6.26}$$

$$\sum_{x,y,z} \mu^t_{xyz} \leq A_t \qquad \forall t \tag{6.27}$$

$$\sum_{x',y',z'} f(x, y, z, x', y', z', t) \leq 1 \qquad \forall x, y, z, t \tag{6.28}$$

where

$$f(x, y, z, x', y', z', t) = \begin{cases} \mu^t_{x'y'z'} & \text{if } x \leq x' \leq x + L^1_t \text{ and} \\ & \quad y \leq y' \leq y + L^2_t \text{ and} \\ & \quad z \leq z' \leq z + L^3_t \\ 0 & \text{otherwise} \end{cases} \tag{6.29}$$

The constraints are very natural: No region in space may be occupied by more than one box type (6.25), boxes must be fully contained within the container (6.27), there may not be more than A_t boxes of type t (6.27), and boxes cannot overlap (6.28). There is one non-overlapping constraint (6.28) for each discretised unit of space and type of box. In order to support rotations, new box types need to be generated for each allowed rotation and linked via set packing constraints similar to constraint (6.27). In order to extend the formulation to the van loading problem, it suffices to add the payload capacity constraint:

$$\sum_{x,y,z,t} \mu^t_{xyz} M_t \leq Q \tag{6.30}$$

6.4.1 Adaptive Discretisation

In order to reduce the number of regions of space, and thus the number of variables in the formulation, a sensible space-discretisation method should be employed. The greatest common divisor (GCD) reduction can be applied on a per-axis basis, finding the greatest common divisor between the length of the container for an axis and all the valid lengths of boxes that can be aligned along that axis and scaling by the inverse of the GCD. This is trivial to do and is useful when all lengths are multiples of each other, which may be common in certain problems. In other problems this may not reduce the number of variables at all. In this case we can apply

a non-linear space-discretisation approach. To do this we use the same values as before on a per-axis basis, i.e. the lengths of any box sides that can be aligned along the axis. We then use dynamic programming to generate all valid locations for a box to be placed. For example, given an axis of length 10 and box lengths of 3, 4 and 6, we can place boxes at positions 0, 3, 4, 6, and 7 (8, 9 and 10 are of course also possible, but no length is small enough to still lie within the container if placed at these points). This has reduced the number of regions along that axis from 10 to 5. An improvement of this scale may not be particularly common in practice, though it is obvious that the non-linear space-discretisation approach will always reduce the number of regions by at least as much as the GCD scaling, and often helps reduce the space by at least a small amount, which can often have a substantial impact on solvers' performance.

6.5 Computational Experience

We have tested the formulations above on three sets of instances:

- 3D Pigeonhole Problem instances, Pigeon-n, where $n+1$ unit cubes are to be packed into a container of dimensions $(1+\epsilon) \times (1+\epsilon) \times n$.
- SA and SAX datasets, which are used to test the dependence of solvers' performance on parameters of the instances, notably the number of boxes, heterogeneity of the boxes, and physical dimensions of the container. There is 1 pseudo-randomly generated instance for every combination of container sizes ranging from 5–100 in steps of 5 units cubed and the number of boxes to pack ranging from 5–100 in steps of 5. The SA datasets are perfectly packable, i.e. are guaranteed to be possible to load the container with 100% utilisation with all boxes packed. The SAX are similar but have no such guarantees; the summed volume of the boxes is greater than that of the container.
- Van-loading instances, generated so as to be similar to those used in pricing sub-problems of branch-and-cut-and-price approaches to vehicle routing with both load weight and volume considerations [GILM06, Lüb10]. The Van-loading and Van-loading-X instances use containers of 10x10x30 and 10x10x50 units, respectively, representing the approximate ratios of a small commercial van and a larger freight truck. 2, 4 or 6 out of 6 pseudo-randomly chosen orientations are allowed. The load weights and knapsack values are generated pseudo-randomly and independently of the volume of the small boxes. 10 instances were generated for each class of problem.

All the tests were performed on a 64-bit computer running Linux, which was equipped with 2 quad-core processors (Intel Xeon E5472) and 16 GB memory. The solvers tested were IBM ILOG CPLEX 12.2.0, Gurobi Solver 4.0, and SCIP 2.0.1 [Ach09] with CLP as the linear programming solver. Instances are available[1] on-line.

[1] http://www.cs.nott.ac.uk/~sda/pigeon

For the 3D Pigeonhole Problem, results obtained within one hour using the three solvers and the Fasano/Padberg formulation are shown in Table 6.2, while Table 6.3 compares the results of Gurobi Solver on both formulations. None of the solvers managed to prove optimality of the incumbent solution for twelve unit-cubes within an hour using the Fasano/Padberg formulation, although the instance of linear programming (after pre-solve) had only 616 rows and 253 columns and 4268 non-zeros. In contrast, the space-indexed formulation allowed Gurobi Solver to solve Pigeon-10000000 within an hour, where the instance of linear programming had 10000002 rows, 10000000 columns, and 30000000 non-zeros. Some of the 3D Pigeonhole instances have been accepted to appear in the forthcoming MIPLIP 2010, a library for testing the capabilities of current mathematical solvers.

For the SA and SAX datasets, Figures 6.1–6.2 summarise solutions obtained within an hour using the Fasano/Padberg formulation, the space-indexed formulation, and a metaheuristic approach described by Allen et al. [ABK11]. In the case of the SAX dataset, the volume utilisation is given with respect to the tightest upper bound found by any of the solvers. Finally, for the Van-loading instances, the results of Gurobi Solver with the space-indexed model after one hour can be seen in Table 6.4. As there were 10 instances tested for each class of problem, the mean number of boxes, number of unique box types and mean gap between solution value and bound are also given.

Overall, the space-indexed relaxation provides a particularly strong upper bound. The mean integrality gap, i.e. (root linear programming relaxation value - optimum value) / optimum value, has been 10.49% and 0.37% for the Fasano/Padberg and the space-indexed formulation, respectively, on the SA and SAX instances solved to optimality within the time limit of one hour. The mean gap, or the ratio of the difference between the linear programming relaxation value and the value of the best solution found to the bound, after an hour on Van-loading instances was 1.5%. This is enouraging, albeit perhaps not particularly surprising, as similar discretised relaxations proved to be very strong in scheduling problems corresponding to one-dimensional packing [SW92, vdA94, PS07] and can be shown to be asymptotically optimal for various geometric problems both in two dimensions [Pap81] and higher [Zem85].

6.6 Conclusions

We have presented a space-indexed linear programming formulation of the container loading problem with a pseudo-polynomial numbers of variables. The space-indexed formulation provides very strong bounds, and hence performs particularly well on instances, where it can be solved. Future work could focus on the development of solvers specific to the relaxation. One could either implement a matrix-free interior point method exploiting the structure of the relaxation, or employ a column generation schema. The Dantzig-Wolfe decomposition could, perhaps, divide the problem into sub-problems pertaining to each type of box (6.28) and a

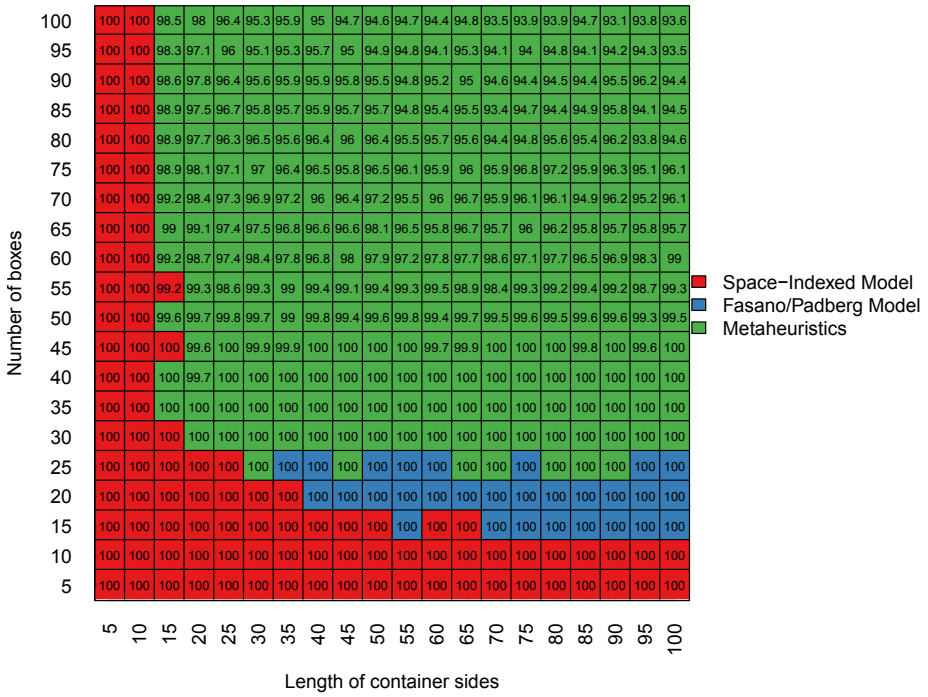

Figure 6.1: The best solutions obtained within an hour per solver per instance from the SA dataset. Each square represents 1 pseudo-randomly generated instance. The best solution is defined as the highest quality solution found, with ties being broken by solve time to best solution. The number is the volume utilisation (in percent) with the tight upper bound of 100.

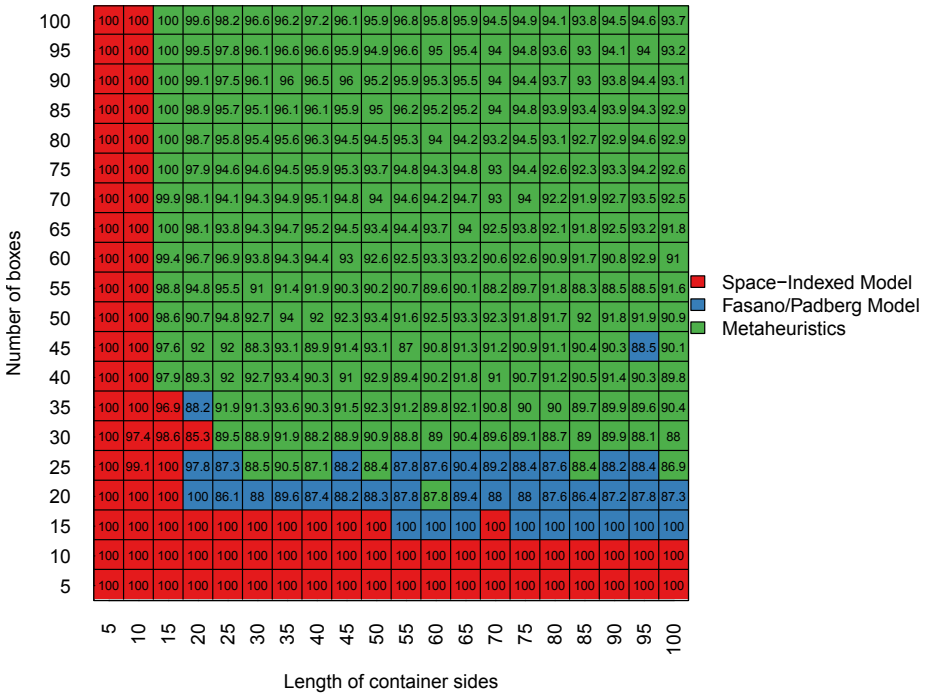

Figure 6.2: The best solutions obtained within an hour per solver per instance from the SAX dataset. Each square represents 1 pseudo-randomly generated instance. The best solution is defined as the highest quality solution found, with ties being broken by solve time to best solution. The number displayed is $100(1 - s/b)$ for solution with value s and upper bound b.

	Time (s)	
	Fasano/Padberg	Space-indexed
Pigeon-01	< 1	< 1
Pigeon-02	< 1	< 1
Pigeon-03	< 1	< 1
Pigeon-04	< 1	< 1
Pigeon-05	< 1	< 1
Pigeon-06	< 1	< 1
Pigeon-07	1.5	< 1
Pigeon-08	7.4	< 1
Pigeon-09	88.6	< 1
Pigeon-10	1381.4	< 1
Pigeon-100	–	< 1
Pigeon-1000	–	1.0
Pigeon-10000	–	1.8
Pigeon-100000	–	4.2
Pigeon-1000000	–	45.1
Pigeon-10000000	–	664.0
Pigeon-100000000	–	–

Table 6.3: The performance of Gurobi 4.0 on 3D Pigeonhole Problem instances encoded in the Fasano/Padberg and the space-indexed formulations. "–" denotes no integer solution has been found within the time limit of one hour.

master problem enforcing the set-packing constraint (6.25). Another interesting direction for future research is automated reformulations. Just as modern integer programming solvers often fail on small instances of the Fasano/Padberg formulation, they fail on certain formulations of scheduling problems. Arguably, one could try to discretise such relaxations automatically, starting with the identification of what variables play the role of relative position in time or space. This could improve the performance of general-purpose integer linear programming solvers considerably.

Dataset	Box Types	Boxes (Mean)	Gap (Mean)
Van-loading-01	3	78	0.10%
Van-loading-02	5	92	0.31%
Van-loading-03	8	81	0.42%
Van-loading-04	10	73	0.32%
Van-loading-05	12	73	0.09%
Van-loading-06	15	70	0.42%
Van-loading-07	20	68	0.50%
Van-loading-08	30	71	0.99%
Van-loading-09	40	69	1.26%
Van-loading-10	50	73	1.10%
Van-loading-X-01	3	133	0.16%
Van-loading-X-02	5	155	0.16%
Van-loading-X-03	8	132	0.56%
Van-loading-X-04	10	119	0.87%
Van-loading-X-05	12	123	0.29%
Van-loading-X-06	15	115	0.36%
Van-loading-X-07	20	116	0.30%
Van-loading-X-08	30	122	0.29%
Van-loading-X-09	40	119	0.18%
Van-loading-X-10	50	121	0.28%

Table 6.4: The performance of the space-indexed model on the Van-loading and Van-loading-X instances after one hour. Each row represents 10 pseudo-randomly generated instances.

7

Conclusions

> It is easier to measure something than to understand what you have measured.
>
> Anon.

7.1 Discussion

Although two-dimensional cutting and packing has been well researched throughout the past few decades, the improvements in packing methods for three dimensions and higher have been comparatively sparse. The need for improved volume utilisation in delivery vehicles will continue to become more important as large companies are driven to lower costs and reduce pollution through transportation. Space allocation in warehousing will become even more of an issue as online mail-order companies such as Amazon continue to grow in popularity.

There have been several methods applied to packing models in three dimensions proposed previously, with varying success and under many different assumptions.

This thesis has shown that there is room for improvement in the current methods reported in the literature, and that gains are still to be had with further research.

We have seen that there exist methods of generating fast and reasonably high quality solutions to the problems by way of the 3BF framework and appropriate heuristics built into this. If real-time packing configurations are not required we have seen that using metaheuristic enhancements to these procedures can drive even further improvements in solution qualities. These methods provide an end-user such as a logistics company with the flexibility to choose the trade-off between fast/real-time packing and repacking of a delivery, or given a slightly longer, but reasonable, time (a few minutes for example) a higher quality solution may be obtained.

The design and implementation of representations for three-dimensional packing has, typically, been fairly unspecified in terms of scalability, and the theoretical limitations of the different representations has mostly been omitted from the literature. We have attempted to contribute something to this with the analysis of the generic 3BF framework, from which most constructive algorithms can be found to derive. This gives practitioners a toolbox of different data structures and problem representations with distinct properties, and properly characterised theoretical analysis when deciding on how to approach a novel three- or higher-dimensional packing problem.

When one can characterise the properties of a problem that is representative of the majority of new problems that will be solved over time, then we know that using a specialised packing heuristic that has been crafted to meet the requirements of the particular problem can be of great utility. This process of "hand crafting" packing heuristics can, as we have seen previously, be automatically reproduced or tuned somewhat by the means of searching through the space of heuristics rather than solutions. The heuristics generated from these methods can, of course, be stored and applied to future problems of a similar nature. Given a long enough "evolution period" and the necessary inputs and training data, the heuristics that are generated can be both fast and efficient.

If an optimal, or near-optimal, solution is required then we have seen that there are methods of generating solutions with mathematical guarantees on solution quality exist. These circumstances may arise more commonly in industry where orders for large deliveries, such as those between warehouses, are given in advance or "batched together" so that the current mathematical programming solvers have long enough to generate incredibly high quality solutions. With the appropriate advances in mathematical solver technologies, the increase in computational power and improved models these methods promise to deliver plenty of potential in the future.

Given the information above and the detailed experimental results presented in the previous chapters, practitioners should glean some extra insight into which techniques are appropriate for a given problem, and how well these solutions will perform under different conditions. A solid framework for (meta-)heuristic solvers has been given, as well as two mathematical programming models which can be implemented in most modern mathematical solvers with relative ease.

7.2 Future work

As with any research project with finite time available to it, there are areas that have not managed to fall into the scope of this thesis but are nonetheless interesting and quite possibly worthy of extended future efforts. These are broken down on a per-chapter basis.

Chapter 3. Given the 3BF framework, novel heuristics and local search neighbourhoods could be derived and applied to three-dimensional cutting and packing problems. If properties of a

problem are known in advance, e.g. the heterogeneity of the boxes to be packed, the ratios of side lengths of the boxes and/or container, stability or load-bearing constraints etc., then custom heuristics could be created and fitted into the framework to solve these problems. A more conclusive study of neighbourhood moves for the metaheuristic stage should be executed in order to ascertain the effectiveness of the current implementation.

Chapter 4. The data structures given for the implementation of the Skyline ADT are by no means definitive, and other data structures from the computational geometry literature could most likely be incorporated into the ADT. An in-depth assessment of other approaches could be a distinct contribution — it is quite possible that there may be exact approaches (i.e. those which do not cut off feasible areas of the solution space) which extend into higher dimensions better than that of the current interval-tree representation. It is conjectured that a better (possibly plane-sweep-style) algorithm for the special case of D-dimensional rectilinear polyhedra, or even D-dimensional simple gap splitting/merging could be derived for the interval-tree representation which is currently limited by the procedures mentioned in the chapter. It is envisioned that the given data structures could also be applied in other related fields such as project scheduling with similar performance benefits, but of course this would need confirming with the appropriate empirical testing.

Chapter 5. The research area of automatically generating heuristics for three-dimensional packing is still a rather new one. That said, we have shown that small, incremental changes to the current methodology can produce significantly larger performance benefits. It is not surprising that improved terminals create improved heuristics, after all: "garbage in, garbage out." Other terminals, possibly with problem specific information, could be implemented in order to further improve the heuristics generated. A more intelligent search methodology could be employed, such as using an enhanced search technique (such as tabu search) and by employing novel neighbourhood moves. This not only applies to generating packing heuristics, but also to genetic programming models on a wider scale.

Chapter 6. The models presented both in the literature and in this thesis are fairly natural, with few constraints. Further work into finding effective constraints for both styles of models presented could lead to significant gains in the tractability of the problems. Similarly, where the models become infeasibly large to solve other techniques such as column generation schemes could be utilised. This could start to close the gap between the problem sizes solvable using exact methods, and problems that can only be sensibly tackled by metaheuristic methods. Another natural extension to the models would be that of load-bearing or gravitational constraints, which should model fairly naturally but will, most likely, increase the complexity of the search for optimality.

7.3 Final Remarks

Research into three-dimensional packing problems is still in its infancy when compared to its lower-dimensioned counterparts, with many gains still to be had from both an academic perspective and in terms of commercial exploitation. This thesis will, hopefully, have given some insight into the state-of-the-art methods for solving these problems and provided a strong enough motivation for doing so. We have seen how different approaches can be employed to great effect on problems of varying sizes, and have pushed the boundaries of the state-of-the-art in several of these methodological areas.

References

[AB98] R. Aboudi and P. Barcia. Determining cutting stock patternswhen defects are present. *Annals of Operations Research*, 82:343–354, 1998.

[AB10] S. D. Allen and E. K. Burke. Automatic generation of three-dimensional packing heuristics. Technical report, University of Nottingham, 2010.

[ABHK09] S. Allen, E. K. Burke, M. Hyde, and G. Kendall. Evolving reusable 3D packing heuristics with genetic programming. In *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 931–938, New York, NY, USA, 2009. ACM.

[ABK11] S. D. Allen, E. K. Burke, and G. Kendall. A hybrid placement strategy for the three-dimensional strip packing problem. *European Journal of Operational Research*, 209(3):219 – 227, 2011.

[ABKG10] S. D. Allen, E. K. Burke, G. Kendall, and S. Groenemeyer. Strip packing: What can we learn from project scheduling? Talk given at the ALIO-INFORMS Joint International Meeting 2010, June 6–9, Buenos Aires, Argentina, June 2010.

[ABM11] S. D. Allen, E. K. Burke, and J. Mareček. A space-indexed formulation of packing boxes into a larger box. Technical report, University of Nottingham, 2011.

[ABar] S. D. Allen and E. K. Burke. A data structure for higher-dimensional rectilinear packing. *INFORMS J. on Computing*, to appear.

[Ach09] T. Achterberg. SCIP: solving constraint integer programs. *Math. Program. Comput.*, 1(1):1–41, 2009.

[AGM02] M. Adler, P. Gibbons, and Y. Matias. Scheduling space-sharing for internet advertising. *Journal of Scheduling*, 5(2):103–119, 2002.

[All09] S. D. Allen. A heuristic/metaheuristic approach to the three-dimensional strip-packing problem. Talk given at the Student Conference on Operational Research, SCOR 2010, March 27–29, Lancaster, UK, March 27–29 2009.

[All10a] S. D. Allen. *CrateViewer: visualising {one, two, three}-dimensional packings*. University of Nottingham, March 2010.

[All10b] S. D. Allen. Data structures for higher-dimensional rectilinear packing. Talk given at the Student Conference on Operational Research, SCOR 2010, April 9–11, Nottingham, UK, April 2010.

[AO79] A. Albano and R. Orsini. An heuristic solution of the rectangular cutting-stock problem. *Comput. J.*, 23:338–343, 1979.

[BD82] E. Bischoff and W. B. Downsland. An application of the microcomputer to product design and distribution. *Journal of the Operational Research Society*, 33(3):271–280, 1982. Cited By (since 1996): 28.

[BDM+99] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

[Bea85] J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33(1):49–64, 1985.

[BEDP08] M. Bader-El-Den and R. Poli. Generating sat local-search heuristics using a gp hyper-heuristic framework. In *Artificial Evolution*, pages 37–49. Springer, 2008.

[BG99] A. Bortfeldt and H. Gehring. Two metaheuristics for strip packing problems. In *Proceedingsband der 5th International Conference of the Decision Sciences Institute, Athen*, volume 2, page 11531156, 1999.

[BG01] A. Bortfeldt and H. Gehring. A hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research*, 131(1):143 – 161, 2001.

[BGM03] A. Bortfeldt, H. Gehring, and D. Mack. A parallel tabu search algorithm for solving the container loading problem. *Parallel Computing*, 29(5):641 – 662, 2003. Parallel computing in logistics.

[BHI+07] N. Bansal, X. Han, K. Iwama, M. Sviridenko, and G. Zhang. Harmonic algorithm for 3-dimensional strip packing problem. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1197–1206, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[BHK+03] E. K. Burke, E. Hart, G. N. Kendall, J. Newall, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In F. Glover and G. Kochenberger, editors, *Handbook of Meta-Heuristics*, pages 457–474. Kluwer, 2003.

[BHKW11] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward. Automating the packing heuristic design process with a genetic programming hyper-heuristic. *Evolutionary Computation*, to appear, 2011.

[Bis06] E. E. Bischoff. Three-dimensional packing of items with limited load bearing strength. *European Journal of Operational Research*, 168(3):952–966, 2006.

[BJR80] B. S. Baker, E. G. C. Jr., and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9(4):846–855, 1980.

[BK05] E. Burke and G. Kendall. *Search methodologies: introductory tutorials in optimization and decision support techniques*. Springer Verlag, 2005.

[BKCS10] A. Bekrar, I. Kacem, C. Chu, and C. Sadfi. An improved heuristic and an exact algorithm for the 2d strip and bin packing problem. *International Journal of Product Development*, 10(1):217–240, 2010.

[BKW04] E. K. Burke, G. Kendall, and G. Whitwell. A new placement heuristic for the orthogonal stock cutting problem. *Oper. Res.*, 52(4):655–671, Jul–Aug 2004.

[BKW09] E. K. Burke, G. Kendall, and G. Whitwell. A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock-cutting problem. *INFORMS J. on Computing*, 21(3):505–516, 2009.

REFERENCES

[BL03] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2):268–281, 2003.

[BM90] E. E. Bischoff and M. D. Marriott. A comparative evaluation of heuristics for container loading. *European J. Operational Research*, 44:266–276, 1990.

[BM07] A. Bortfeldt and D. Mack. A heuristic for the three-dimensional strip packing problem. *European J. Oper. Res.*, 183(3):1267–1279, 2007.

[BMP99] O. Bournez, O. Maler, and A. Pnueli. Orthogonal polyhedra: Representation and computation. In F. W. Vaandrager and J. H. van Schuppen, editors, *HSCC*, volume 1569 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 1999.

[BR95] E. Bischoff and M. Ratcliff. Issues in the development of approaches to container loading. *Omega*, 23:377–390(14), August 1995.

[CC09] M. Chlebík and J. Chlebíková. Hardness of approximation for orthogonal rectangle packing and covering problems. *J. Discrete Algorithms*, 7(3):291–305, 2009.

[CCT+03] A. Crispin, P. Clay, G. Taylor, T. Bayes, and D. Reedman. Genetic algorithms applied to leather lay plan material utilization. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 217(12):1753–1756, 2003.

[CFC94] C. H. Cheng, B. R. Feiring, and T. C. E. Cheng. The cutting stock problem – a survey. *International Journal of Production Economics*, 36(3):291–305, october 1994.

[CG84] E. G. Coffman, Jr. and E. N. Gilbert. Dynamic first fit packings in two or more dimensions. *Inf. and Control*, 61:1–14, 1984.

[CGJ78] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.*, 7(1):1–17, 1978.

[CGJ84] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing—an updated survey. In *Algorithm design for computer system design*, volume 284 of *CISM Courses and Lectures*, pages 49–106. Springer, Vienna, 1984.

[CGJ87] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Bin packing with divisible item sizes. *J. Complexity*, 3:406–428, 1987.

[CGJT80] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9:808–826, 1980.

[CH95] N. Christofides and E. Hadjiconstantionu. An exact algorithm for orthogonal 2-d cutting problems using guillotine cuts. *European Journal of Operations Research*, 83:21–38, 1995.

[Cha83] B. Chazelle. The bottom-left bin-packing heuristic: An efficient implementation. *IEEE Trans. Computers*, 32(8):697–707, 1983.

[CJCM08] F. Clautiaux, A. Jouglet, J. Carlier, and A. Moukrim. A new constraint programming approach for the orthogonal packing problem. *Comput. Oper. Res.*, 35(3):944–959, 2008.

[CJGJ96] E. Coffman Jr, M. Garey, and D. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[CJL89] E. Coffman Jr and J. Lagarias. Algorithms for packing squares: A probabilistic analysis. *SIAM Journal on Computing*, 18:166, 1989.

[CL91] E. G. Coffman Jr. and G. S. Lueker. *Probabilistic analysis of packing and partitioning algorithms*. Series in Discrete Mathematics and Optimization. A Wiley-Interscience Publication John Wiley & Sons, Inc., New York, 1991.

[CLS95] C. S. Chen, S. M. Lee, and Q. S. Shen. An analytical model for the container loading problem. *European Journal of Operational Research*, 80(1):68 – 76, 1995.

[CLT88] E. G. Coffman Jr., J. Y. Leung, and D. W. Ting. Bin packing: maximizing the number of pieces packed. *Acta Informatica*, 9(3):263–271, 1977/788.

[CMV94] S. Chiang, R. Mansharamani, and M. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 33–44. ACM, 1994.

[CMWX08] G. Cintra, F. Miyazawa, Y. Wakabayashi, and E. Xavier. Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. *European Journal of Operational Research*, 191(1):61–85, 2008.

[CPM00] G. Chryssolouris, N. Papakostas, and D. Mourtzis. A decision-making approach for nesting scheduling: a textile case. *International Journal of Production Research*, 38(17):4555–4564, 2000.

[CPT08] T. G. Crainic, G. Perboli, and R. Tadei. Extreme point-based heuristics for three-dimensional bin packing. *INFORMS J. on Computing*, 20(3):368–384, 2008.

[CS90] E. G. Coffman, Jr. and P. W. Shor. Average-case analysis of cutting and packing in two dimensions. *European J. Operational Research*, 44:134–144, 1990.

[CS93] E. G. Coffman Jr. and P. W. Shor. Packings in two dimensions: asymptotic average-case analysis of algorithms. *Algorithmica*, 9(3):253–277, 1993.

[CSY02] J. Cagan, K. Shimada, and S. Yin. A survey of computational approaches to three-dimensional layout problems. *Computer-Aided Design*, 34(8):597–611, 2002.

[Dan98] G. Dantzig. *Linear programming and extensions*. Princeton Univ Pr, 1998.

[dBCvKO08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, April 2008.

[DD92] K. Dowsland and W. Dowsland. Packing problems. *European Journal of Operational Research*, 56(1):2–14, 1992.

[DG74] R. G. Dyson and A. S. Gregory. The cutting stock problem in the flat glass industry. *Operational Research Quarterly*, 25(1):41–53, March 1974.

[Dow93] K. A. Dowsland. Some experiments with simulated annealing techniques for packing problems. *Journal of Operational Research*, pages 389–399, 1993.

[Dyc90] H. Dyckhoff. A typology of cutting and packing problems. *European J. Operational Research*, 44:145–159, 1990.

[Dyc92] U. Dyckhoff, H.and Finke. *Cutting and Packing in Production and Distribution: Typology and Bibliography.* (eds. Mueller, W.A. and Schuster, P.) Springer-Verlag, New York, 1992.

[EG75] P. Erdős and R. L. Graham. On packing squares with equal squares. *J. Combinatorial Theory Ser. A*, 19:119–123, 1975.

[Ele02] M. Eley. Solving container loading problems by block arrangement. *European Journal of Operational Research*, 141(2):393 – 409, 2002.

[EP09] J. Egeblad and D. Pisinger. Heuristic approaches for the two- and three-dimensional knapsack packing problem. *Comput. Oper. Res.*, 36(4):1026–1049, 2009.

[Fas99] G. Fasano. Cargo analytical integration in space engineering: A three-dimensional packing model. In T. A. Ciriani, S. Gliozzi, E. L. Johnson, and R. Tadei, editors, *Operational Research in Industry*, pages 232–246. Purdue University Press, 1999.

[Fas04] G. Fasano. A mip approach for some practical packing problems: balancing constraints and tetris-like items. *4OR*, 2(2):161–174, 2004.

[FDHI10] G. Fuellerer, K. Doerner, R. Hartl, and M. Iori. Metaheuristics for vehicle routing problems with three-dimensional loading constraints. *European Journal of Operational Research*, 201(3):751–759, 2010.

[Fog02] D. Fogel. An introduction to simulated evolutionary optimization. *Neural Networks, IEEE Transactions on*, 5(1):3–14, 2002.

[FS97] S. P. Fekete and J. Schepers. A new exact algorithm for general orthogonal d-dimensional knapsack problems. Technical report, Mathematisches Institut, Universit"at zu K"oln, 1997.

[Fuk08] A. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, 2008.

[GB97] H. Gehring and A. Bortfeldt. A genetic algorithm for solving the container loading problem. *International Transactions in Operational Research*, 4(5-6):401 – 418, 1997.

[GG65] P. Gilmore and R. Gomory. Multistage cutting stock problems of two and more dimensions. *Ops. Res.*, 13:94–120, 1965.

[GILM06] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search algorithm for a routing and container loading problem. *Transportation Science*, 40(3):342–350, 2006.

[GJ77a] M. Garey and D. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM J. Comput.*, 6(3):416–426, 1977.

[GJ77b] M. Gonzalez Jr. Deterministic processor scheduling. *ACM Computing Surveys (CSUR)*, 9(3):173–204, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Co., San Francisco, Calif., 1979. A guide to the theory of NP-completeness, A Series of Books in the Mathematical Sciences.

[GL93] F. Glover and M. Laguna. *Modern heuristic techniques for combinatorial problems*, chapter Tabu search, pages 70–150. John Wiley & Sons, Inc., 1993.

[GMM90] H. Gehring, K. Menschner, and M. Meyer. A computer based heruistic for packing pooled shipment containers. *European J. Operational Research*, 44:277–288, 1990.

[GR80] J. A. George and D. F. Robinson. A heuristic for packing boxes into a container. *Computers and Operations Research*, 7(3):147–156, 1980. Cited By (since 1996): 79.

[GUA06] C. Geiger, R. Uzsoy, and H. Aytuğ. Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *Journal of Scheduling*, 9(1):7–34, 2006.

[Hah68] S. Hahn. On the optimal cutting of defective sheets. *Operations Research*, 16(6):1100–1114, 1968.

[Har00] S. Hartmann. Packing problems and project scheduling models: an integrating perspective. *J Oper Res Soc*, 9:1083–1092, 2000.

[HH09] W. Huang and K. He. A caving degree approach for the single container loading problem. *European Journal of Operational Research*, 196(1):93 – 101, 2009.

[HH11] K. He and W. Huang. An efficient placement heuristic for three-dimensional rectangular packing. *Computers & Operations Research*, 38(1):227 – 233, 2011. Project Management and Scheduling.

[Hif01] M. Hifi. Exact algorithms for large-scale unconstrained two and three staged cutting problems. *Comput. Optim. Appl.*, 18(1):63–88, 2001.

[HKE89] C. Han, K. Knott, and P. Egbelu. A heuristic approach to the three-dimensional cargo-loading problem. *International journal of production research*, 27(5):757–774, 1989.

[Hol92] J. Holland. *Adaptation in natural and artificial systems.* MIT Press Cambridge, MA, USA, 1992.

[HRK96] W. Hower, M. Rosendahl, and D. Köstner. Evolutionary algorithm design. In *Artificial Intelligence in Design 96*, pages 663–680, Dordrecht, Germany, 1996. Kluwer Academic Publishers.

[HT99] E. Hopper and B. C. H. Turton. A genetic algorithm for a 2d industrial packing problem. *Comput. Ind. Eng.*, 37(1-2):375–378, 1999.

[HT01a] E. Hopper and B. Turton. A review of the application of meta-heuristic algorithms to 2d strip packing problems. *Artificial Intelligence Review*, 16(4):257–300, 2001.

[HT01b] E. Hopper and B. C. H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European J. Oper. Res.*, 127(1):34–57, January 2001.

[HWPS98] I. Harjunkoski, T. Westerlund, R. P
"orn, and H. Skrifvars. Different transformations for solving non-convex trim-loss problems by minlp. *European Journal of Operational Research*, 105(3):594–603, 1998.

[HZWC10] C. He, Y. Zhang, J. Wu, and C. Chang. Research of three-dimensional container-packing problems based on discrete particle swarm optimization algorithm. In *Test and Measurement, 2009. ICTM'09. International Conference on*, volume 2, pages 425–428. IEEE, 2010.

[Jak96] S. Jakobs. On genetic algorithms for the packing of polygons. *European J. Oper. Res.*, 88(1):165–181, January 1996.

REFERENCES

[JHD97] M. Jouaneh, A. Hammad, and P. Datseris. A flexible automated foam cutting system. *International Journal of Machine Tools and Manufacture*, 37(4):437–449, 1997.

[JJB07] D. Jakobović, L. Jelenković, and L. Budin. Genetic programming heuristics for multiple machine scheduling. *Genetic Programming*, pages 321–330, 2007.

[JRZ97] M. Johnson, C. Rennick, and E. Zak. Skiving addition to the cutting stock problem in the paper industry. *Siam Review*, 39(3):472–483, 1997.

[JSO06] K. Jansen and R. Solis-Oba. An asymptotic approximation algorithm for 3d-strip packing. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 143–152, New York, NY, USA, 2006. ACM.

[KD02] L. Kos and J. Duhovnik. Cutting optimization with variable-sized stock and inventory status data. *International journal of production research*, 40(10):2289–2301, 2002.

[KI04] K. Karabulut and M. M. Inceoglu. A hybrid genetic algorithm for packing in 3d with deepest bottom left with fill method. In *ADVIS*, pages 441–450, 2004.

[KIN+09] M. Kenmochi, T. Imamichi, K. Nonobe, M. Yagiura, and H. Nagamochi. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, 198(1):73–83, 2009.

[Kir84] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5):975–986, 1984.

[Koz92] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[LC90] K. Li and K.-H. Cheng. On three-dimensional packing. *SIAM J. Comput.*, 19(5):847–867, 1990.

[LC92] K. Li and K.-H. Cheng. Heuristic algorithms for on-line packing in three dimensions. *J. Algorithms*, 13(4):589–605, 1992.

[LH10] R. Lougee-Heimer. The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, 2010.

[LLM02] L. Lins, S. Lins, and R. Morabito. An n-tet graph approach for non-guillotine packings of n-dimensional boxes into an n-container. *European Journal of Operational Research*, 141(2):421 – 439, 2002.

[LM80] O. Larsen and G. Mikkelsen. An interactive system for the loading of cargo aircraft. *European Journal of Operational Research*, 4(6):367 – 373, 1980.

[LMM02] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 127(2):241–252, September 2002.

[LMV99] A. Lodi, S. Martello, and D. Vigo. Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem. In *MIC-97: meta-heuristics international conference*, pages 125–139, 1999.

[LMV02] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Appl. Math.*, 123(1-3):379–396, 2002.

[LRW03] A. Lim, B. Rodrigues, and Y. Wang. A multi-faced buildup algorithm for three-dimensional packing problems. *Omega*, 31(6):471 – 481, 2003.

[LRY05] A. Lim, B. Rodrigues, and Y. Yang. 3-d container packing heuristics. *Applied Intelligence*, 22(2):125–134, 2005.

[Lüb10] M. Lübbecke. Personal communication, 2010.

[MA93] R. Morábito and M. N. Arenales. An and/or graph approach to the container loading problem. NOTAS DO ICMSC 001, University of São Paulo - São Carlos, São Carlos - Brazil, abril 1993.

[MAA92] R. Morábito, M. Arenales, and V. F. Arcaro. An and-or-graph approach for two-dimensional cutting problems. *European J. Operational Research*, 58:263–271, 1992.

[MBG04] D. Mack, A. Bortfeldt, and H. Gehring. A parallel hybrid local search algorithm for the container loading problem. *International Transactions in Operational Research*, 11(5):511–533, 2004.

[McD99] C. McDiarmid. Pattern minimisation in cutting stock problems. *Discrete Applied Mathematics*, 98(1-2):121–130, 1999.

[MG98] R. Morabito and V. Garcia. The cutting stock problem in a hardboard industry: A case study. *Computers and Operations Research*, 25(6):469–485, 1998.

[MH78] R. C. Merkle and M. E. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions On Information Theory*, 24:525–530, 1978.

[MM68] A. Meir and L. Moser. On packing of squares and cubes. *Journal of combinatorial theory*, 5(2):126–134, 1968.

[MO05] A. Moura and J. Oliveira. A grasp approach to the container-loading problem. *Intelligent Systems, IEEE*, 20(4):50 – 57, 2005.

[Mos89] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[MPT99] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.

[MPV00] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Oper. Res.*, 48(2):256–267, 2000.

[MPV+07] S. Martello, D. Pisinger, D. Vigo, E. D. Boef, and J. Korst. Algorithm 864: General and robot-packable variants of the three-dimensional bin packing problem. *ACM Trans. Math. Softw.*, 33(1):7, 2007.

[MR06] I. Mukherjee and P. K. Ray. A review of optimization techniques in metal cutting processes. *Computers & Industrial Engineering*, 50(1-2):15 – 34, 2006.

[MS02] S. Menon and L. Schrage. Order allocation for stock cutting in the paper industry. *Operations Research*, 50(2):324–332, 2002.

[MT90] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[MW04] F. K. Miyazawa and Y. Wakabayashi. Packing problems with orthogonal rotations. In *LATIN*, pages 359–368, 2004.

[NI02] K. Nonobe and T. Ibaraki. Formulation and tabu search algorithm for the resource constrained project scheduling problem. *Essays and Surveys in Metaheuristics*, pages 557–588, 2002.

[NT00] S. Nonås and A. Thorstenson. A combined cutting-stock and lot-sizing problem. *European Journal of Operational Research*, 120(2):327–342, 2000.

[NW88] G. Nemhauser and L. Wolsey. *Integer and combinatorial optimization*, volume 18. Wiley New York, 1988.

[OGF00] J. Oliveira, A. Gomes, and J. Ferreira. Topos–a new constructive algorithm for nesting problems. *OR Spectrum*, 22(2):263–284, 2000.

[Olt05] M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[Pad00] M. Padberg. Packing small boxes into a big box. *Math. Methods Oper. Res.*, 52(1):1–21, 2000.

[Pap81] C. H. Papadimitriou. Worst-case and probabilistic analysis of a geometric location problem. *SIAM J. Comput.*, 10(3):542–557, 1981.

[PE06] D. Pisinger and J. Egeblad. Heuristic approaches for the two- and three- dimensional knapsack packing problems. Technical report, Dept. of Computer Science, University of Copenhagen, 2006.

[Pis02] D. Pisinger. Heuristics for the container loading problem. *European J. Oper. Res.*, 127(2):382–392, September 2002.

[PRK04] J. Puchinger, G. Raidl, and G. Koller. Solving a real-world glass cutting problem. *Evolutionary Computation in Combinatorial Optimization*, pages 165–176, 2004.

[PS98] C. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover Pubns, 1998.

[PS07] Y. Pan and L. Shi. On the equivalence of the max-min transportation lower bound and the time-indexed lower bound for single-machine scheduling problems. *Math. Program.*, 110(3, Ser. A):543–559, 2007.

[RBN09] M. Riff, X. Bonnaire, and B. Neveu. A revision of recent approaches for two-dimensional strip-packing problems. *Engineering Applications of Artificial Intelligence*, 22(4-5):823–827, 2009.

[Rob84] S. A. Roberts. Application of heuristic techniques to the cutting-stock problem for worktops. *J. Opl Res. Soc.*, 35(5):369–377, May 1984.

[SAdC10] E. Silva, F. Alvelos, and J. V. de Carvalho. An integer programming model for two- and three-stage two-dimensional cutting stock problems. *European Journal of Operational Research*, 205(3):699 – 708, 2010.

[SP92] P. E. Sweeney and E. R. Paternoster. Cutting and packing problems: a categorized, application-oriented research bibliography. *J. Operational Research Society*, 43(7):691 – 706, 1992.

[SSW94] Z. Sinuany-Stern and I. Weiner. The one dimensional cutting stock problem using two objectives. *Journal of the Operational Research Society*, pages 231–236, 1994.

[SW92] J. P. Sousa and L. A. Wolsey. A time indexed formulation of non-preemptive single machine scheduling problems. *Mathematical Programming*, 54:353–367, 1992.

[Tan10] C.-H. Tang. A scenario decomposition-genetic algorithm method for solving stochastic air cargo container loading problems. *Transportation Research Part E: Logistics and Transportation Review*, In Press, Corrected Proof:–, 2010.

[TH08] J. Tay and N. Ho. Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Computers & Industrial Engineering*, 54(3):453–473, 2008.

[TMC+04] J. Tavares, P. Machado, A. Cardoso, F. Pereira, and E. Costa. On the evolution of evolutionary algorithms. *Genetic Programming*, pages 389–398, 2004.

[TMFÁR05] H. Terashima-Marín, E. J. Flores-Álvarez, and P. Ross. Hyper-heuristics and classifier systems for solving 2d-regular cutting stock problems. In H.-G. Beyer and U.-M. O'Reilly, editors, *GECCO*, pages 637–643. ACM, 2005.

[TTS94] A. Tarnowski, J. Terno, and G. Scheithauer. A polynomial time algorithm for the guillotine pallet loading problem. *INFOR special issue: Knapsack, packing and cutting, part II*, 32, 1994. Published and sponsored by the Canadian Operational Research Society and Canadian Information Processing Society.

[vdA94] J. M. van den Akker. *LP-based solution methods for single-machine scheduling problems*. Technische Universiteit Eindhoven, Eindhoven, 1994. Dissertation.

[VFFS89] E. Vasko Floyd, J. Francis, and K. Stott. A practical solution to a fuzzy two-dimensional cutting stock problem. *Fuzzy Sets and Systems*, 29(3):259–275, 1989.

[Wal02] P. Walsh. *Advanced 3-D Game Programming using DirectX 8.0*. Wordware Publishing Inc, Texas USA, 2002.

[WGC+10] L. Wang, S. Guo, S. Chen, W. Zhu, and A. Lim. Two natural heuristics for 3d packing with practical loading constraints. *PRICAI 2010: Trends in Artificial Intelligence*, pages 256–267, 2010.

[WHS07] G. Wäscher, H. Haußner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 127(3):1109–1130, December 2007.

[WLGDS10] Y. Wu, W. Li, M. Goh, and R. De Souza. Three-dimensional bin packing problem with variable bin height. *European Journal of Operational Research*, 202(2):347–355, 2010.

[WW88] P. Widmayer and D. Wood. A time- and space-optimal algorithm for boolean mask operations for orthogonal polygons. *CVGIP*, 41(1):14–27, January 1988.

[WZC09] L. Wei, D. Zhang, and Q. Chen. A least wasted first heuristic algorithm for the rectangular packing problem. *Computers & Operations Research*, 36(5):1608–1614, 2009.

[YY87] A. M. Yaglom and I. M. Yaglom. *Challenging mathematical problems with elementary solutions*. Dover Publications, Inc., New York, NY, USA, 1987.

[Zem85] E. Zemel. Probabilistic analysis of geometric location problems. *SIAM J. Algebraic Discrete Methods*, 6(2):189–200, 1985.