KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

fwo

Simple, Flexible, and Open-Source 2D Cutting & Packing Algorithms

Jeroen Gardeyn

Dissertation presented in partial fulfillment
of the requirements for the degree of
Doctor of Engineering Technology (PhD)

November 2025

Supervisors:
Prof. Tony Wauters
Prof. Greet Vanden Berghe

SIMPLE, FLEXIBLE, AND OPEN-SOURCE 2D CUTTING & PACKING ALGORITHMS

Jeroen GARDEYN

Supervisors:
Prof. Tony Wauters
Prof. Greet Vanden Berghe

Members of the
Examination Committee:
Prof. Elke Gruyaert, chair
Prof. Pieter Vansteenwegen
Dr. Thijs Steel
Bern De Lille
(ARDIS, Belgium)
Prof. José Fernando Oliveira
(Universidade do Porto, Portugal)
Prof. Ted Ralphs
(Lehigh University, United States of America)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

November 2025

© 2025 Jeroen Gardeyn
Uitgegeven in eigen beheer, Jeroen Gardeyn, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Acknowledgements

A PhD is a journey marked by both periods of rapid progress and long stretches where one feels stuck. Throughout this journey, I have been fortunate to receive the support of many people who helped me through the difficult times and shared in the good ones.

First, I would like to thank Greet and Tony for giving me the opportunity to pursue a PhD, and more generally for the research group they founded and continue to lead. They created an environment where meetings happen spontaneously, supervisors are always accessible, and freedom and independence are strongly encouraged. Within this setting, I truly felt able to leave my mark—not only on the direction of my own research but also on the group as a whole.

I am especially grateful to Luke. Beyond thoroughly reviewing and annotating every draft I sent his way, the most valuable moments were our one-on-one meetings. We spent countless hours brainstorming and discussing how best to explain ideas concisely, present arguments convincingly, and weave them into a coherent and engaging story. His input elevated this thesis to a level of refinement that would have been impossible without his help. In many ways, he felt like a third supervisor – one who probably deserves to be acknowledged as a co-author.

Although a PhD is an individual pursuit, the collegial atmosphere in the CODeS lab made it a genuinely enjoyable place to work. Thank you to Alessio, Augustin, Bryan, Carlo, Chiara, David, Everton, Fabien, Hatice, Jan, Jonas, Khadija, Lisa, Manos, Michiel, Niels, Pieter, Reinout, Reshma, Robin, Vinicius, and Wim. I especially cherish the numerous conference trips with Jonas and Bryan, which felt more like holidays with close friends than work obligations.

Finally, I want to thank my family and friends who have been a constant source of support, encouragement, and stability throughout this journey. I now recognize that this is not something to take for granted, and I am deeply grateful for it.

I will always look back on this chapter of my life with great fondness.

Jeroen Gardeyn Ghent, November 2025

Abstract

Cutting and packing (C&P) problems involve placing items into larger containers while optimizing objectives such as minimizing waste or maximizing material yield. They arise in numerous industrial and logistical contexts where solution quality directly affects operational efficiency. However, their combinatorial nature makes them notoriously difficult to solve. The number of possible configurations grows exponentially with problem size, challenging the design of methods that can deliver high-quality solutions in a reasonable time. Despite decades of research, the quest for more effective algorithms remains pressing, with the pursuit of marginal gains often leading to increasingly fragile and convoluted methods that fail to generalize.

This thesis advances the state of the art in 2D C&P problems and focuses on developing simple, flexible, open-source algorithms. It comprises three core contributions, each accompanied by a robust open-source implementation. First, we introduce `GDRR`, a goal-driven ruin and recreate heuristic for the 2D rectangular bin packing problem with guillotine constraints. We then focus on irregular C&P problems, where polygonal shapes add a significant geometric challenge on top of the underlying combinatorial challenges. To address this, we present `jagua-rs`, a dedicated *Collision Detection Engine* that can efficiently and accurately verify whether items are colliding, thereby lowering the barrier to entry for 2D nesting. Finally, building on this engine, we introduce `sparrow`, a complete heuristic that sets a new academic and open-source baseline for 2D irregular strip packing problems.

A lack of reproducibility hinders researchers from verifying or building on prior work, impeding scientific progress. Within the field of *Computational Operations Research*, reproducibility unfortunately remains the exception rather than the rule. We contend that a culture of openly sharing source code would largely mitigate this issue. This thesis takes a strong stance on reproducibility: the codebases of the implementations are treated as first-class research artifacts – on par with, and complementary to, the methodological descriptions.

This dissertation puts forward the idea that pursuing simple, flexible methods – paired with polished open-source implementations – presents one of the most promising paths to lasting impact for the research community. In this way, we aim for our contributions not merely to be a single step forward, but to catalyze future research and advances.

List of Abbreviations

2BP two-dimensional bin packing problem. viii, 6, 11–13, 16, 17, 23, 30, 32, 35, 36

2DISPP two dimensional irregular strip packing problem. 7, 67–72, 74, 87, 91, 97, 100

2VSBP two-dimensional variable-sized bin packing problem. viii, xiii, 12, 14, 32, 34, 35

AABB axis-aligned bounding box. 51, 54, 55

C&P cutting and packing. iii, xi–xiii, 1, 5, 6, 39–44, 55, 58, 61–64, 66, 67, 69, 71, 79, 102, 103, 116, 117

CDE collision detection engine. xi, 6, 40, 43, 44, 47, 49–51, 53, 55, 58, 61–64, 67, 73, 75, 76, 82, 89, 103, 104

COR Computational Operations Research. 3, 4

CPU central processing unit. 29, 75, 90

GLS guided local search. 82, 85

MIP mixed-integer programming. 3

NFP no-fit polygon. 41–43, 72, 79

PoI pole of inaccessibility. 55, 57, 61

SIMD single instruction, multiple data. 75

SotA state of the art. 66, 69, 70, 91, 100

Contents

Acknowledgements i

Abstract iii

List of Abbreviations v

Contents vii

List of Figures xi

List of Tables xiii

1 Introduction 1

I Rectangular Cutting & Packing Problems 9

2 A Goal-Driven Ruin and Recreate Heuristic for 2D Bin Packing 10

2.1 Introduction . . . 11
2.2 Literature Review . . . 13
2.3 Solution Representation . . . 14
2.4 Ruin and Recreate Heuristic . . . 16
2.4.1 Goal-driven Approach . . . 16
2.4.2 Ruin . . . 19
2.4.3 Recreate . . . 21
2.4.3.1 Insertion Options . . . 22
2.4.3.2 Blinks . . . 24
2.4.4 Evaluation . . . 25
2.4.4.1 Solution Quality . . . 25
2.4.4.2 Late-Acceptance Hill-Climbing . . . 26
2.4.5 Multithreading . . . 28

2.5 Computational Results . . . 28
2.5.1 Parameters . . . 29
2.5.2 Results for 2BP|O|G . . . 30
2.5.3 Results for 2BP|R|G . . . 32
2.5.4 Results for 2VSBP|O|G . . . 32
2.5.5 Results for 2VSBP|R|G . . . 35
2.5.6 Impact of Multiple Threads . . . 35
2.5.7 Impact of Calculation Time . . . 36
2.6 Conclusion . . . 36

II Irregular Cutting & Packing Problems 37

3 An Open-Source Collision Detection Engine for 2D Nesting Problems 38
3.1 Introduction . . . 39
3.2 General Approaches to Geometry in 2D C&P Problems . . . 41
3.3 Collision Detection Engine . . . 43
3.4 Polygon Collision . . . 45
3.5 Hazards . . . 47
3.6 CDE Implementation . . . 49
3.6.1 Two-phased Approach . . . 49
3.6.2 Quadtree . . . 50
3.6.2.1 Querying the Quadtree . . . 51
3.6.2.2 Updating the Quadtree . . . 52
3.6.3 Edge Intersection . . . 53
3.6.4 Inclusion . . . 54
3.7 Fail-fast Surrogate . . . 55
3.7.1 Poles (of Inaccessibility) . . . 57
3.7.2 Piers . . . 58
3.8 Polygon Simplification . . . 58
3.9 Codebase . . . 61
3.10 Validation & Evaluation . . . 63
3.11 Conclusion . . . 64

4 An Open-Source Heuristic to Reboot 2D Nesting Research 65
4.1 Introduction . . . 66
4.2 Problem Definition . . . 67
4.3 State of 2D Irregular Strip Packing . . . 69
4.4 A Sequence of Feasibility Problems . . . 71
4.5 Detecting Collisions . . . 72
4.6 Quantifying Collisions . . . 73
4.6.1 Area of Overlap . . . 74
4.6.2 Decaying Penetration Depth . . . 76

4.6.3 Shape-based Penalty 78
4.7 Navigating a Continuous Search Space 79
4.8 Solving the Feasibility Problem 82
4.8.1 Guided Local Search 82
4.8.2 Separation Procedure 83
4.8.3 Balancing Exploration and Exploitation 86
4.9 Solving the 2D Irregular Strip Packing Problem 87
4.9.1 Exploration Phase 87
4.9.2 Compression Phase 88
4.10 Implementation 89
4.11 Computational Experiments 89
4.11.1 Experimental Setup 90
4.11.2 Performance Analysis 90
4.11.3 Time-sensitivity Analysis 93
4.12 Real-world Benchmarks & Future Research Directions 97
4.13 Conclusions and Reflections 100

5 Conclusions and Reflections **102**

Bibliography **107**

List of Publications **121**

List of Figures

1.1 A representative developmental flow of an algorithm/model to solve a specific combinatorial optimization problem, presented as a graph. 4

2.1 Visual representation of non-guillotine and guillotine patterns . 11
2.2 Visual representation of the data structure for a single bin . . . 14
2.3 High-level overview of `GDRR` 16
2.4 Progression of `GDRR` over time 17
2.5 Three examples of node removal 19
2.6 Four options of inserting an item into a leftover node. 23
2.7 Impact of multithreading on solution quality (600s of runtime) 35
2.8 Impact of calculation time on solution quality (8 threads) . . . 36

3.1 Example solution of a leather nesting instance. 39
3.2 Fundamental interaction between the CDE and an optimization algorithm for irregular C&P problems. 44
3.3 Types of collision between a pair of simple polygons. 45
3.4 Different types of hazards. 47
3.5 The `collision` function. 49
3.6 An abstract representation of the two-phased approach. 49
3.7 Quadtree for an interior hazard with varying maximum depths d. 51
3.8 Querying an edge through the quadtree: $\texttt{QT}(e)$. 52
3.9 Three ways to split a node in the quadtree. 53
3.10 The `edge_intersection` function. 53
3.11 The `inclusion` function. 55
3.12 The `fail_fast_collision` procedure. 56
3.13 Example of a fail-fast surrogate. 57
3.14 Three options for simplification. 59
3.15 Example of polygon simplification for different values of α. . . 60

3.16 High-level overview of the different layers involved in solving a 2D irregular C&P problem and the scope of `jagua-rs`. 62

4.1 The 2D irregular strip packing problem. 67
4.2 Example of an infeasible and feasible configuration of a set of items within containers of equal dimensions. 72
4.3 Two shapes in different configurations, with their respective poles drawn. 74
4.4 Visualization of Algorithm 5 76
4.5 Comparing the standard (δ) and decaying (δ') penetration depth functions. 77
4.6 Visualization of Algorithm 6 77
4.7 The `perform_sampling` procedure. 81
4.8 The `separate` procedure and all its underlying components presented in a hierarchical overview. Every component references its corresponding algorithm ($A.*$) or figure ($F.*$). 85
4.9 Distribution of `sparrow`'s solution quality ρ across the academic benchmark instances. 92
4.10 Scaling of the expected solution quality $\mathbb{E}(\rho)$ of `sparrow` for the three most challenging instances across different time limits. . . 94
4.11 The best solutions obtained by `sparrow` (Table 4.3) and their respective densities. 95
4.11 The best solutions obtained by `sparrow` (Table 4.3) and their respective densities. (continued) 96
4.12 Ten new benchmark instances for the 2DISPP. 98
4.12 Ten new benchmark instances for the 2DISPP. (continued) . . 99

List of Tables

2.1 Results for the 2DBP|O|G benchmark from Berkey and Wang [13] and Lodi et al. [63] 31
2.2 Results for the 2DBP|R|G benchmark from Berkey and Wang [13] and Lodi et al. [63] 33
2.3 Results for the 2VSBP|O|G benchmark from Hopper and Turton [51] 34
2.4 Results for the 2VSBP|O|G benchmark from Pisinger and Sigurd [78] 34
2.5 Results for the 2VSBP|O|G benchmark from Ortmann et al. [76] 34

3.1 Strengths and weaknesses of the general approaches for collision detection in 2D irregular C&P problems. 43

4.1 Configuration of `sparrow`'s parameters throughout the experiments. 90
4.2 Comparison of the expected solution quality $\mathbb{E}(\rho)$ of `sparrow` on academic benchmarks. 93
4.3 The best solution densities ever produced by `sparrow` for instances in the benchmark set 94

Chapter 1

Introduction

Cutting and packing (C&P) problems involve placing a set of smaller *items* within larger *containers* while optimizing a given objective. Various one-, two- or three-dimensional (1D, 2D, 3D) C&P problems arise in industrial and logistical contexts, where solution quality directly affects operational efficiency. Effective solutions enable containers to be *packed* densely and raw materials to be *cut* apart with minimal waste. C&P covers a broad class of combinatorial optimization problems, ranging from the classic 1D cutting stock problem formulated in the 1960s [54] to modern applications such as extracting the maximum value from rough diamonds or minimizing the number of cowhides needed to produce leather furniture.

The combinatorial nature of these problems causes the number of possible configurations to grow exponentially with the size of the problem instance, making it challenging to design methods that deliver high-quality solutions within a reasonable time. Over the last half-century, researchers have developed a wide variety of approaches to tackle a broad spectrum of C&P problems. Yet, persisting challenges ensure that the quest for more effective algorithms remains as relevant as ever.

Flexibility & simplicity

One common way to contribute to combinatorial optimization research is to engage in *horse-racing*: trying to improve upon the previous best-known results on established academic benchmarks. While better solutions are the ultimate goal of optimization, this approach often produces increasingly fragile and convoluted methods that deliver only marginal gains. Many such methods are so specialized that even slight changes in the problem definition render them ineffective, and they rarely generalize to real-world settings – much like a race car that performs well on a track but struggles with potholes on everyday roads.

Another, arguably underemphasized, way to contribute is to pursue *flexible* and *simple* methods. Methods that:

- Are immediately understandable without multiple readings of the introducing publication.
- Deliver consistently competitive performance without extensive fine-tuning or reliance on specific problem characteristics.
- Adapt gracefully to small changes in the problem definition.
- Are distilled to their essence, with each component meaningfully contributing to overall performance.

Few researchers would dispute that these properties are desirable, both in others' work and as goals in their own. Flexible, simple methods can reach a wider audience and provide a foundation for more specialized approaches to address a broad range of real-world problems. Yet they are seldom prioritized. In our view, the reasons are twofold.

First, flexibility and simplicity combined with competitive performance do not arise naturally. Flexibility requires careful, disciplined design choices throughout development, sometimes even at the expense of potentially performance-enhancing features. Simplicity, meanwhile, results from rigorously removing unnecessary complexity, unafraid to *'kill your darlings'*.

Second, progress along these dimensions is harder to quantify objectively and, consequently, harder to legitimize as a scientific contribution. Arguing for the methodological merits of an algorithm demands careful reasoning and persuasion, whereas presenting numerical performance results is comparatively straightforward.

Reproducibility

Reproducibility – the ability to independently verify scientific claims – is a cornerstone of the scientific method. Yet science in general faces a reproducibility *crisis*, with many studies, including those in high-impact peer-reviewed journals, found to be irreproducible [6]. Within computer science, Collberg and Proebsting [23] concluded that the causes of non-reproducibility are often sociological rather than technical.

Although their report is almost a decade old, the trends they identified remain highly relevant to Computational Operations Research (COR), the field in which this thesis is situated. Some voices within COR do advocate for stronger reproducibility practices [8, 55, 59, 71, 73], but a broader cultural shift is still lacking. Reproducibility remains the exception rather than the rule.

In several computer science (related) fields, there has been a strong shift toward open science in recent years. In System Security, for example, leading publication venues now require submissions to include a *research artifact* to ensure broad reproducibility and replicability [1, 70, 89].

COR lags behind when it comes to reproducibility. This is surprising, as it should in theory be one of the easiest domains to ensure reproducibility. Experiments are typically computational in nature, performed in controlled environments, do not require exotic hardware or costly consumables, and commonly used software – such as MIP solvers – often provides free academic licenses.

By contrast, chemistry experiments demand expensive equipment and consumables, and pharmaceutical research faces complex regulatory and ethical constraints. In comparison, enforcing reproducibility in COR should be straightforward.

To illustrate, consider a typical research project aimed at solving a combinatorial optimization problem. Researchers iteratively design a mathematical model or search algorithm, iterating between conceptualization and implementation. Towards the end of the project, computational experiments are performed and the final methodology is described. The description and results are ultimately disseminated to the academic community in the form of a scientific publication.

Figure 1.1: A representative developmental flow of an algorithm/model to solve a specific combinatorial optimization problem, presented as a graph.



However, computational results are never a direct product of the methodological description – they arise from the implementation itself. Reproducibility hinges on the assumption that an equivalent implementation can be reconstructed solely from the published methodology. In practice, this premise is naive. Methodological descriptions are high-level, *lossy* abstractions of the final implementation. Replicating the computational results from such descriptions is often impossible. The workflow depicted in Figure 1.1 is an over-simplification, but the key idea remains: there is no direct path from 'methodological description' to 'computational results'.

The implementation, which is the crucial link to enable reproducibility in COR, is most often excluded from publication. Academia readily publishes methodological advances validated by computational experiments but, for some reason, is very reluctant to share the code that produced these results.

The responsibility for this situation cannot be fully placed on the shoulders of individual researchers though. Reproducibility has to be encouraged and enforced by the community at large. Currently, most top-tier journals do neither require reproducible results nor mandate inclusion of complete computational results as supplementary material.[1] While bottom-up efforts toward open science are emerging, a cultural shift must be supported from the top down, with leading journals and conferences requiring reproducibility as a condition for publication.

I strongly believe that any publication whose contributions rely – even partly – on computational evidence should include the source code. Claims based on irreproducible results are scientifically unsound. In 2025, open science should be the default, not the exception.

[1] A notable exception being INFORMS Journal on Computing

This thesis

This thesis focuses on the category of 2D C&P problems for which it aims to advance the academic state of the art by presenting novel algorithms and methodologies with a strong emphasis on **flexibility** and **simplicity**. Each of the three main chapters is accompanied by an **open-source** implementation that can be used to (i) replicate our computational results, (ii) serve as a foundation for future research, or (iii) be deployed and adapted for real-world applications. These publicly available codebases[2] are not merely 'supplementary material' but each represent an integral and important part of this thesis.

Concretely, we propose the following research hypothesis:

The pursuit of simple, flexible methods – supported by polished open-source implementations – offers one of the most promising paths toward achieving lasting value and impact for the research community.

Although this hypothesis serves as the guiding principle for the following research, it cannot be proven within the bounds of this thesis itself. It is only in the years following the publication of this thesis that its validity will ultimately be determined.

This thesis is divided into two parts, which not only differ in content but also reflect different phases of the PhD. We briefly situate the context of each part and the chapters within them.

PART I: Rectangular C&P Problems

This first part of this thesis stems from a two-year collaboration with ARDIS[3] within a VLAIO[4] project (code: `HBC.2018.2297`). The goal was to design an algorithm to serve as the base for a new commercial solver for 2D rectangular C&P problems. The solver needed to perform well across a wide variety of instances while supporting diverse constraints and objectives. Flexibility and simplicity were therefore essential and guided every design decision.

[2] https://github.com/JeroenGar
[3] https://www.ardis.eu/en-US/home
[4] Flanders Innovation & Entrepreneurship

The methods developed in this project have been distilled to their core and adapted into a heuristic for the academic 2D bin packing (2BP) problem with guillotine cuts. In the bin packing problem, both the containers (bins) and the items are fixed in size and the objective is to pack all items into the minimum number of bins. Chapter 2 presents this heuristic, `GDRR`, which supports both 90° rotation of items in combination with a heterogeneous set of bins.

These first two years also shaped my perspective on the field. I found that flexibility and simplicity were rarely prioritized in research, and I was often shocked by the lack of verifiability in published results. Even setting aside the lack of source code, the individual solutions underlying reported results are almost never shared, leaving their correctness unverified and preventing any assessment beyond the authors' chosen metrics.

PART II: Irregular C&P Problems

The second part of this research was carried out under a four-year *strategic basic research* grant from FWO[5] (codes: `1S71222N`, `1S71224N`). This grant provided me both the time to reflect on the broader research landscape of 2D C&P problems and the impetus to take a strong stance on reproducibility and open-source practices, reinforced by FWO's open-access policy[6], which states that *"research financed with public funds should be publicly available"*.

We sought to identify which subfield of 2D C&P suffered most from the issues outlined earlier and could benefit from four years of dedicated research. We quickly arrived at 2D irregular C&P – or *nesting* – problems. These problems do not restrict the shapes of items and containers to rectangles but allow arbitrary polygons instead. The irregularity of the shapes makes nesting problems among the most challenging C&P problems. These problems pose not only an **optimization** challenge of finding good solutions but also a **geometric** one: verifying feasibility requires ensuring that items do not *collide* with each other or with the container. This added difficulty raises the barrier to entry and amplifies reproducibility issues.

In Chapter 3, we address the geometric challenge directly by introducing a *Collision Detection Engine* (CDE) for 2D irregular C&P. Released as the software library `jagua-rs`, it provides a backend for optimization algorithms to verify feasibility.

[5] Research Foundation - Flanders
[6] https://www.fwo.be/en/about-fwo/research-policy/open-access/

In Chapter 4, with `jagua-rs` in place, we shift focus to the combinatorial challenge and tackle the canonical 2D irregular strip packing problem (2DISPP). The 2DISPP consists of packing a set of irregular items into a strip of fixed width and infinite height, minimizing the height used. We present `sparrow`, an open-source heuristic built on top of `jagua-rs`.

Part I

Rectangular Cutting & Packing Problems

Chapter 2

A Goal-Driven Ruin and Recreate Heuristic for the 2D Variable-Sized Bin Packing Problem with Guillotine Constraints

This chapter is a minimally edited version of the following published work:

J. Gardeyn and T. Wauters. "A goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints". In: *European Journal of Operational Research* 301.2 (2022), pp. 432–444. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2021.11.031

This chapter is presented together with the following open-source software:

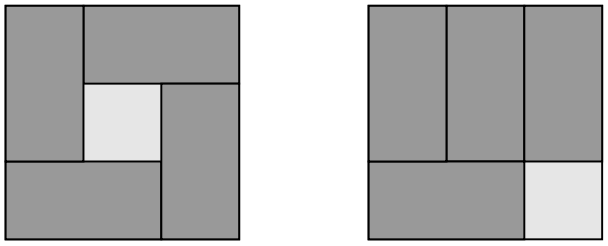
J. Gardeyn. "`gdrr-2bp`: a Rust implementation of the goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints." 2022. URL: https://github.com/JeroenGar/gdrr-2bp

2.1 Introduction

The Two-Dimensional Bin Packing Problem (2BP) consists of packing a heterogeneous set of small rectangular items into one or more larger rectangular bins. In general, a 2BP solution is considered feasible if (1) the items are fully inside the bin, (2) items are not overlapping and (3) the edges of the items are parallel to the edges of the bin. The goal is to minimize the amount of unused bin area incurred by packing all the items.

Lodi et al. [63] defined four variants of the 2BP problem using a three-field notation. This notation defines whether the items are allowed to rotate 90 degrees and/or whether guillotine cuts are required. This chapter focuses on variants of the 2BP which require guillotine cuts (2BP|*|G). Both fixed orientation (2BP|O|G) and the variant that allows 90° rotation of items (2BP|R|G) are addressed.

The guillotine constraint defines that all cuts must be edge-to-edge cuts. Figure 2.1(a) shows a simple visual example of a pattern which does not comply with the guillotine constraint. It is not possible to cut this pattern by exclusively using edge-to-edge cuts. The pattern in Figure 2.1(b), on the other hand, satisfies the guillotine constraint by first making a horizontal cut followed by several vertical cuts. Guillotine cuts have many real-world applications as they are often required in glass- and wood-cutting industries.



(a) Non-guillotine pattern (b) Guillotine pattern

Figure 2.1: Visual representation of non-guillotine and guillotine patterns

This work introduces a novel heuristic GDRR, based on the ruin and recreate paradigm combined with a goal-driven approach. The heuristic attempts to improve the solution at each iteration by removing and reinserting items into the bins in a greedy fashion. GDRR is goal-driven in the sense that it iteratively lowers the available bin area. Each time the algorithm finds a feasible solution with a certain total bin area, it is henceforth forced to find new solutions which use less bin area. As a result, most of the time, the heuristic will be unable to fit all items into the available bins and will therefore be working with infeasible solutions. After each run, items which could not be placed are automatically reconsidered for reinsertion during the next run. In order to reach complete solutions, the objective function penalizes unassigned items in order to steer the heuristic towards feasibility.

GDRR is also capable of addressing the variant with a heterogeneous set of bins. This generalization is commonly referred to as the 2BP with variable-sized bins. Since the classification made by Lodi et al. [63] does not cover this variant, it will henceforth be referred to as the 2VSBP.

In summary, the proposed heuristic addresses four variants of the 2BP:

- **2BP|O|G**: identical bins, no rotation
- **2BP|R|G**: identical bins, 90° rotation
- **2VSBP|O|G**: heterogeneous set of bins (variable-sized), no rotation
- **2VSBP|R|G**: heterogeneous set of bins (variable-sized), 90° rotation

The remainder of the chapter is organized as follows. In Section 2.2 a compact literature review is provided which includes papers covering both the traditional 2BP and the variant with a heterogeneous set of bins. Section 2.3 introduces the data structure for solution representation. Next, a detailed explanation of the complete ruin and recreate heuristic is given in Section 2.4. A comparison of the heuristic against the current state of the art is conducted in Section 2.5. Finally, Section 2.6 summarizes the chapter and highlights some future research directions.

2.2 Literature Review

General typologies for cutting and packing problems are published in Dyckhoff [27] and Wäscher et al. [95]. Surveys for the 2BP have been provided by Lodi et al. [64] and Lodi et al. [62]. These surveys cover heuristics, exact methods and bounds for the 2BP. A more recent survey focusing on exact methods was conducted by Iori et al. [53].

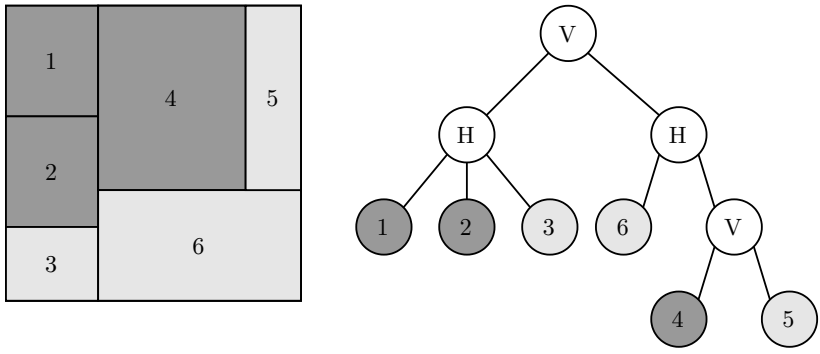
Gilmore and Gomory [44] were the first to tackle the 2BP and proposed a column generation approach for the multidimensional bin packing problem. Many heuristics for 2BP|*|G have since been published in the literature and this review will focus on the most competitive among these. Polyakovsky and M'Hallah [79] developed a 'guillotine bottom left' constructive heuristic to pack one bin at a time in combination with a pseudo-parallel agent-based system wherein each agent has its own characteristics. Later, Charalambous and Fleszar [18] introduced a constructive heuristic which supports item rotation. At each iteration, their method packs one bin by creating multiple simple cutting patterns and selects the one with the highest quality by way of a sufficiency criterion. To improve solution quality, they proposed a post-optimization routine which prioritizes items that most often do not fit during the constructive phase. Later, Fleszar [34] proposed three new insertion heuristics and a justification improvement heuristic. Lodi et al. [65] introduced a heuristic based on an enumeration tree for the 2BP|O|G. At each level of the tree, a new bin is packed. When filling a new bin, a set of selection and guillotine split rules are considered. These different strategies create new nodes in the tree. In an enhanced version of the heuristic, they attempt to reduce the search space to improve performance. More recently, Cui et al. [24] present a sequential value correction heuristic which repeatedly creates cutting patterns while trying to maximize the total item value. The values of the items are updated after each iteration based on information from current and past solutions. Finally, a heuristic which uses a pattern generation procedure that creates triple-block patterns was introduced by Cui et al. [25]. The procedure is followed by an improvement phase which uses an Integer Linear Programming model to select the best patterns.

The concept of heterogeneous sets of bins (variable-sized) was first introduced by Friesen and Langston [37]. They proposed three heuristics for one-dimensional bin packing problems. Literature on the 2VSBP|*|G is more scarce compared to the variant with identical bins. To the best of our knowledge, only the variant without rotation of items has been studied. Ortmann et al. [76] proposed a two-stage heuristic, which begins with a strip packing procedure. Next, attempts are made to repack these strips into smaller and smaller bins. Later, Hong et al. [50] designed a hybrid heuristic for the 2VSBP|O|G. They proposed a mixed bin packing algorithm which is used in combination with iterative simulated annealing runs and a binary search. Finally, Wei et al. [96] introduced a heuristic with a goal-driven approach for the 2VSBP|O|F (no guillotine constraint). Their algorithm packs bins with a sequential packing heuristic, followed by a local search to improve the solution. It also performs a binary search on bin combinations and increases the search effort each time.

It should be clear that many of these solution methods are either based on exact methods or focus heavily on the constructive aspect. Exact methods, however, often lack the flexibility required to adapt to real-world scenarios and generally do not scale well for larger problems. On the other hand, although constructive heuristics are usually more flexible and very fast, their solutions are often lacking in terms of quality. The approach proposed in this chapter attempts to find a balance by focusing on heuristic improvement strategies.

2.3 Solution Representation

Figure 2.2: Visual representation of the data structure for a single bin



Each cutting pattern is represented as a rooted tree, as shown in Figure 2.2. The tree consists of three different types of node. There are item (dark gray), leftover (light gray) and structure nodes (white). Leaves are always either an item or leftover node. All non-leaf nodes define the structure of a cutting pattern in a hierarchical way and are denoted as either vertical (V) or horizontal (H). The orientation of the node corresponds to the direction in which its children are cut. This orientation is identical for every node of a level and each new level switches the orientation compared to the previous one.

A similar, but not identical, data structure was used before by Clautiaux et al. [20], Fleszar [34] and Kröger [56]. As noted by Fleszar [34], this representation always satisfies the guillotine constraint and never allows for overlapping items.

Note that the data structure does not define any coordinates. It only describes relative positions of items and leftovers. This allows for simpler insertion or removal of items (or groups of items) compared to more rigid structures where a cutting pattern is defined by a set of items and their locations.

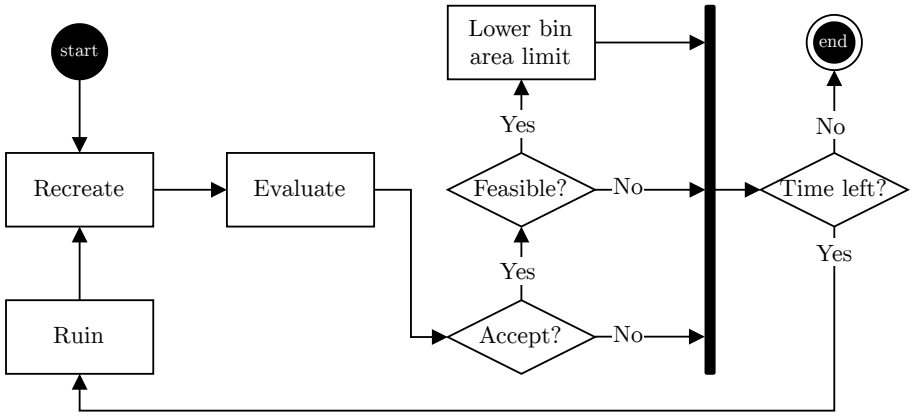
As was first described by Gilmore and Gomory [44], in real-world scenarios there is often a limit on the number of possible stages. A stage includes one or more horizontal or vertical cuts, but never a mixture of the two. Progressing from one stage to another involves rotating the cut orientation. The number of stages therefore represents the number of blade rotations needed to cut the pattern. In this representation, since the cutting orientations are switched at every level, each level represents a stage. While not the focus of this chapter, limiting the number of stages in this data structure involves simply defining a tree's maximum depth.

Due to the nature of the heuristic, there is a need to be able to represent incomplete solutions. Therefore, a solution $S = \{C, E\}$ not only contains a set of cutting patterns C, it also consists of a set of excluded items E. A solution is deemed feasible if all items are included ($E = \varnothing$).

2.4 Ruin and Recreate Heuristic

Figure 2.3 provides a high-level overview of the heuristic. At each iteration of the local search, the solution is partially destroyed and rebuilt in an attempt to find improvements. Every time a new feasible solution is found, the bin area limit is reduced. The general idea behind this is discussed in Section 2.4.1. Detailed explanations of the ruin and recreate procedures are given in Sections 2.4.2 and 2.4.3. Afterwards, the modified solution is evaluated and has the possibility of being accepted. This is covered in Section 2.4.4. The entire process is repeated until a given time limit is reached.

Figure 2.3: High-level overview of `GDRR`



2.4.1 Goal-driven Approach

In the 2BP, the quality of a solution is based on the sum of areas of the bins required to pack all items. Due to the nature of the problem, all feasible solutions which result in the same total bin area are deemed to be equal in quality. This means that, once a complete solution is found with a certain total bin area, solutions with equal (or greater) total bin area are no longer of interest.

Wei et al. [96] proposed a goal-driven approach (GDA) heuristic for the 2BP with variable-sized bins. It generates all combinations of bins that have a total area between a lower and upper bound, sorts them by area and attempts to find the best combination that produces a feasible solution. The sorted list is traversed using a binary search. At each iteration, the GDA heuristic must use that specific combination of bins. Although this particular algorithm does not incorporate the guillotine constraint, the general idea is applicable.

Similar to GDA, the proposed approach could also be viewed as being goal-driven. The recreate phase, responsible for rebuilding the ruined solution, must adhere to a certain bin area limit. A new bin can only be opened if the total bin area of the solution remains below this threshold. Therefore, most of the time, the recreate phase will be unable to fit all items into the available bins and will produce infeasible solutions. A cost function guides the heuristic towards a solution capable of fitting all items.

Figure 2.4: Progression of `GDRR` over time

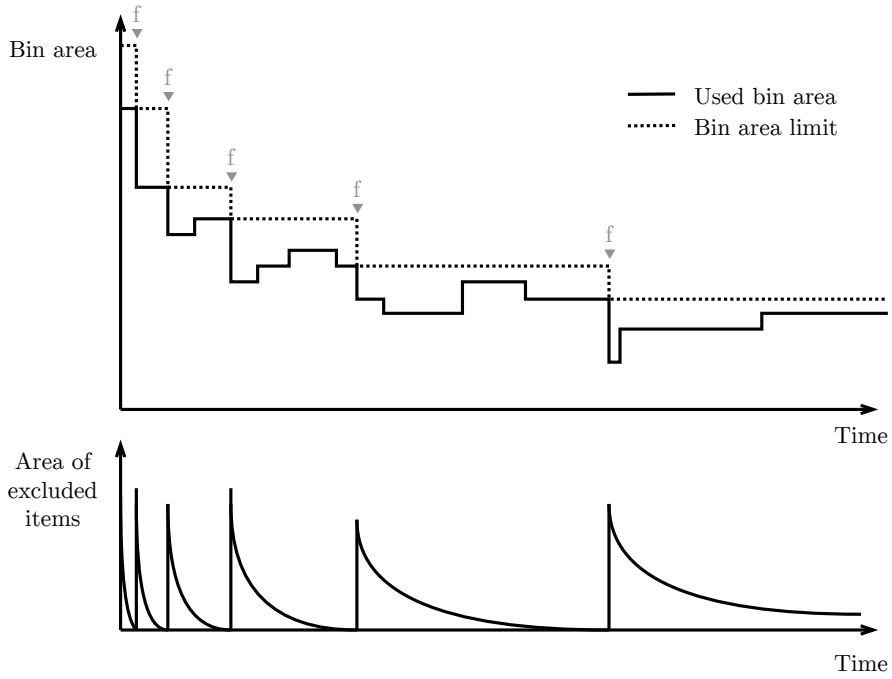


Figure 2.4 represents one possible course of the heuristic over time. At all times, the goal is to find a set of cutting patterns capable of packing all items with a total bin area below the current limit. Whenever a new feasible solution (denoted by 'f') is found, the limit is lowered to the total bin area of this new solution. A formal description of the complete `GDRR` heuristic and how the bin area limit tightens is presented in Algorithm 3 at the end of Section 2.4.

Unlike Wei et al. [96], the search does not begin from scratch each time, but rather destroys the solution until at least one bin is freed up entirely, satisfying the new bin area limit. Therefore, high-quality features from a previous solution can be carried over when the limit is lowered. This can result in a significant amount of saved computational time. However, the heuristic may be more likely to get stuck in local optima when solutions are already tightly packed.

Another difference lies in the combination of available bins. GDA tries to make a feasible solution with a fixed combination of bins. By contrast, `GDRR` is free to use whichever combination of bins yields good results according to the cost function. `GDRR` only needs to ensure that the total bin area is below the current limit. Furthermore, it can sometimes happen that there is a feasible solution for a combination of bins A, but not for a combination of bins B despite the total bin area of A being smaller than B. This situation most often occurs with instances that contain small bins (few items per bin) or items with extreme dimensions. Due to the binary search nature of GDA, these combinations might not be reachable.

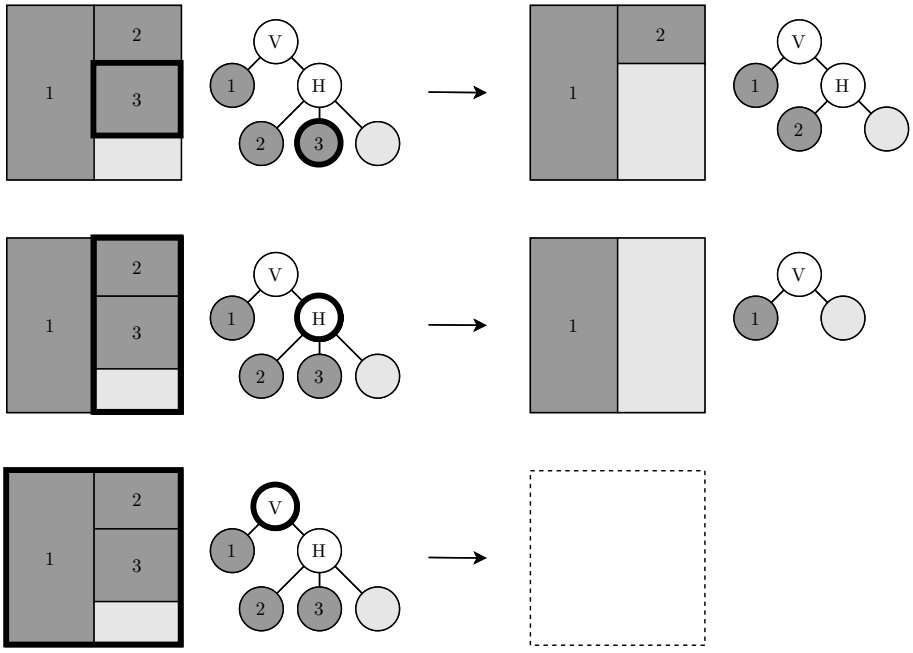
It should become clear that, by design, the heuristic will be working with infeasible solutions most of the time. This might seem counterproductive at first, but such a strategy has two major advantages compared to incorporating bin area into a cost function. First of all, items which are not included in the solution have a new chance at being inserted every iteration. This drastically increases the likelihood of finding better solutions. If this were not the case, reaching complete solutions with less bin area would be unlikely, especially when the cutting patterns are complex. Second, as shown in Figure 2.4, the total area of excluded items can be used as a very gradual indicator of a solution's quality. This is in stark contrast to the abrupt drops when using bin area as a cost factor.

In conclusion, although the proposed heuristic is also goal-driven, it significantly differs from the approach introduced by Wei et al. [96]. They merely share the same philosophy of constraining the use of bins in some way.

2.4.2 Ruin

The ruin procedure removes a number of nodes from the solution. All item and structure nodes are candidates for removal. Leftover nodes are ineligible since removing them would not have any effect. They can only be deleted as a result of removing a parent node.

Figure 2.5: Three examples of node removal



Three cases of node removal are shown in Figure 2.5. These figures illustrate the situation before (left) and after (right) the removal of a node. In each example, the node with a bold edge is the one selected for removal. In the first example, the node containing item 3 is removed. As a result, the leftover node is expanded to account for the freed-up space. In the second example, a first-level structure node is deleted. This results in items 2 and 3 being removed since they are children. The final example shows the root node being selected for removal. This results in the entire cutting pattern being removed. It should be clear that impact of a single removal can range from individual items to entire cutting patterns. The closer a structure node is to the root of the tree, the more significant an impact its removal will have.

There are two main motivations for allowing structure nodes to be removed. First, removing these nodes produces large regions of contiguous leftover space. This, in turn, provides the recreation phase more flexibility, resulting in diverse neighboring solutions. The second main advantage lies in the fact that it is probable that some of the removed items will form good patterns again at another location. In the second example, for instance, items 2 and 3 are likely to form a compact pattern again because they share the same width. This is particularly apparent in the later stages of the search, when the solution is already quite good in terms of quality. It could be viewed as a form of related removal.

However, dramatically destroying the solution can hinder the chances of finding better solutions. Radically ruining a solution too often can result in a lot of wasted time. Because the most impactful nodes are closest to the root of the tree, they are less numerous compared to the less impactful nodes at the bottom of the tree. Due to the fact that the ruin method selects nodes at random, removals with a significant impact have a lower chance of occurring. This could be viewed as a sort of built-in balancing mechanism.

Algorithm 1 Ruin phase

Input: A solution $S = \{C, E\}$ where C is a set of cutting patterns and E is a set of excluded items, bin area limit $\mathcal{A}_{lim}$, and the average number of nodes to be removed μ
Output: A modified (ruined) solution S'

function RUIN($S, \mathcal{A}_{lim}, \mu$)
  $S' = \{C', E'\} \leftarrow S$
  $i \in_R \{1, ..., 2\mu - 1\}$
  while $i > 0$ **or** $\sum_{c \in C'} \mathcal{A}_c \geq \mathcal{A}_{lim}$ **do**
    $c \in_R C'$
    $N_c \leftarrow$ item and structure nodes of c
    $n \in_R N_c$
    $c' \leftarrow$ REMOVE(c, n)
    $C' \leftarrow C' \setminus \{c\}$
    if c' is not empty **then**
      $C' \leftarrow C' \cup \{c'\}$
    $E_n \leftarrow$ items in n
    $E' \leftarrow E' \cup E_n$
    $i \leftarrow i - 1$
  return S'

Algorithm 1 provides the pseudocode for the ruin phase. The number of nodes to be removed, i, is uniformly randomized ($\in_R$) at each iteration and averages out at μ (line 3). The procedure continues ruining until i nodes have been removed or while the total bin area of the modified solution exceeds the bin area limit $\mathcal{A}_{lim}$ (line 4). For each removal, a cutting pattern c is selected uniformly at random (line 5). A random item or structure node n is selected from c (lines 6-7). This node is removed from c, resulting in c' (lines 8-9). Finally, both the set of cutting patterns and the set of excluded items associated with the modified solution S' are updated (lines 10-13).

2.4.3 Recreate

During the recreate phase, the (ruined) solution needs to be rebuilt. This is done by inserting items into either leftover nodes or new bins. Of course, due to the bin area limit, it is highly unlikely that it will be possible to insert all items.

Algorithm 2 provides the pseudocode for the recreate phase. During this phase, all currently excluded items have the opportunity to be inserted into the solution (lines 3-4). Set $\tilde{E}$ corresponds to all items which have yet to receive a chance at insertion. At each iteration, the most restricted item is selected as the next insertion candidate (line 5). The restrictiveness of an item is determined by its number of possible insertion places: the more possibilities, the less restricted[1]. When multiple items are deemed equally restricted, a random item is selected. The definition of insertion options is provided in Section 2.4.3.1 in addition to how their costs are calculated. For the selected item, e_{mr}, all insertion options are generated (line 6). If there are no available options (line 7), an attempt is made to open a new bin (lines 8-12). Only bins which will not result in exceeding the bin area limit $\mathcal{A}_{lim}$ are eligible (lines 8-9). If possible, a new and empty cutting pattern is created with a random eligible bin (lines 10-11) and the set of insertion options is updated (line 12). If the item can be inserted (line 13), all options are evaluated (line 14) and eventually the item is inserted into the solution (lines 15-18). Most of the time, the cheapest option is selected in a greedy fashion, but sometimes the best options are 'blinked' over. The concept of blinks is explained in Section 2.4.3.2. Regardless of whether a successful insertion takes place, item e_{mr} is removed from $\tilde{E}$ as it has now received a chance at insertion (line 19). Once every item has been given a chance to be inserted, a recreated solution S' is returned (line 20).

[1]Determining the most restricted item can be performed in linear time ($O(n)$) by caching the insertion options for all excluded items in a table and performing updates throughout the recreate phase.

Algorithm 2 Recreate phase

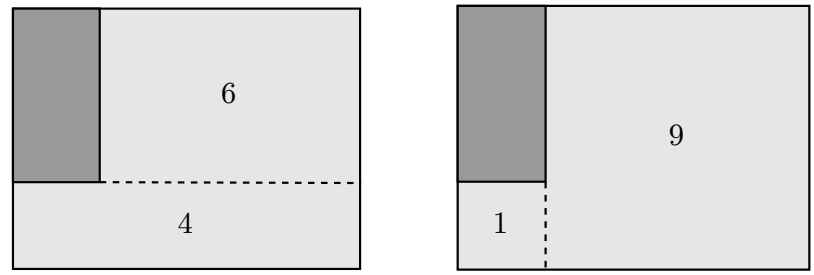
Input: A solution $S = \{C, E\}$ where C is a set of cutting patterns and E is a set of excluded items, bin area limit $\mathcal{A}_{lim}$
Output: A modified (recreated) solution S'

function RECREATE($S, \mathcal{A}_{lim}$)
  $S' = \{C', E'\} \leftarrow S$
  $\tilde{E} \leftarrow E'$
  while $\tilde{E} \neq \emptyset$ **do**
    $e_{mr} \leftarrow$ most restricted item $\in \tilde{E}$
    $O \leftarrow$ insertion options for e_{mr} for all cutting patterns $\in C'$
    if $O = \emptyset$ **then** ▷ *attempt to open new bin*
      $\Delta \leftarrow \mathcal{A}_{lim} - \sum_{c \in C'} \mathcal{A}_c$
      if there is a bin with $\mathcal{A} < \Delta$ **then**
        $c \leftarrow$ empty cutting pattern using random bin with $\mathcal{A}_c < \Delta$
        $C' \leftarrow C' \cup \{c\}$
        $O \leftarrow O \cup \{$insertion options for e_{mr} in $c\}$
    if $O \neq \emptyset$ **then** ▷ *insert the item*
      $o \leftarrow$ select best (with blinks) $\in O$
      $c \leftarrow$ cutting pattern of o
      $c' \leftarrow$ INSERT(c, o)
      $C' \leftarrow C' \setminus \{c\} \cup \{c'\}$
      $E' \leftarrow E' \setminus \{e_{mr}\}$
    $\tilde{E} \leftarrow \tilde{E} \setminus \{e_{mr}\}$
  return S'

2.4.3.1 Insertion Options

An insertion option defines how exactly an item is inserted into a leftover node. If 90° rotation is allowed, there are two possible orientations for each item. Due to the guillotine constraint, each feasible orientation of an item in a leftover node, in turn, generates two insertion options depending on the direction of the first cut (vertical or horizontal). As a result, there are at most four different ways of inserting an item into a leftover node. An illustrative example of these four possibilities is shown in Figure 2.6. For each excluded item, all leftover nodes capable of fitting the item are considered.

Figure 2.6: Four options of inserting an item into a leftover node.



(a) $cost = 12^2 - (6^2 + 4^2) = 92$ (b) $cost = 12^2 - (9^2 + 1^2) = 62$

(c) $cost = 12^2 - (8^2 + 2^2) = 74$ (d) $cost = 12^2 - (6^2 + 4^2) = 92$

Note: $width_{n_o^u} = 4$, $height_{n_o^u} = 3$, $\alpha = 2$, 2BP|R|G

All insertion options have an associated cost. This cost is used as a measure of an option's desirability and is calculated using Equation 2.1. Inserting items not only consumes existing leftover nodes, but also generates new ones. The cost of an insertion option o corresponds to the difference in total leftover value before and after the insertion. Each leftover node thus has a corresponding value (Equation 2.2) and the aim is to minimize the loss in value resulting from insertion.

n_o^u : leftover node used by insertion option o
N_o^c : set of leftover nodes created by insertion option o
α : leftover area value exponent

$$cost(o) = value(n_o^u) - \sum_{n \in N_o^c} value(n) \tag{2.1}$$

$$value(n) = (width_n \cdot height_n)^\alpha \tag{2.2}$$

As shown in Equation 2.2, the value of a leftover node increases polynomially with respect to area. This is to encourage the heuristic to preserve large contiguous leftover areas instead of many smaller ones. Generally speaking, it will be easier to pack items into few, but larger, leftover nodes rather than many small ones. In Figure 2.6, option (b) has the lowest cost since it preserves the largest leftover value.

The value of exponent α has an impact on the behavior of the algorithm because it defines the ordering of insertion options. Making this value too large can lead to the creation of very few well-sized leftover areas. It will ignore many medium-sized leftover nodes and only care about a few large ones. On the other hand, setting the value of α too small can be detrimental for preserving large leftover areas.

2.4.3.2 Blinks

Always selecting the best option would result in a deterministic recreate phase and may lead to making the same bad decisions over and over again. To combat this, it is sometimes necessary to turn a blind eye to an insertion option. The concept of blinks was first introduced by Christiaens and Vanden Berghe [19] in a state-of-the-art heuristic for solving Vehicle Routing Problems. The general idea, however, is also applicable here.

In a pure best-fit, the best option is always selected. But with blinks, there is a small chance for an option to be ignored. Blinking rate β defines the likeliness of skipping over an option. Equation 2.3 shows the chance of an option being selected based on its rank r. A value for β of 0.05, for example, means each option has a 5% chance of being ignored.

$$p(r) = (1 - \beta)\beta^{(r-1)} \qquad r \in \{1, ..., \infty\} \tag{2.3}$$

Blinking exhibits the same behavior as Heuristic-Biased Stochastic Sampling, introduced by Bresina [15], with an exponential bias function. However, as Christiaens and Vanden Berghe [19] note, blinking is more efficient since there is no need to sort options based on their fitness.

2.4.4 Evaluation

After each ruin and recreate iteration, the solution needs to be evaluated. This is accomplished by determining the quality of the solution in question, as explained in Section 2.4.4.1. Based on this quality, the solution is either accepted or rejected using the Late-Acceptance Hill-Climbing metaheuristic, which is described in Section 2.4.4.2.

2.4.4.1 Solution Quality

Due to bin area limit constraints, the recreate phase will most likely be unable to incorporate all items into the solution. The goal, therefore, is to guide the heuristic towards a feasible solution. Equation 2.4 describes how two solutions are compared against each other and resembles a lexicographic ordering.

$$
\begin{array}{ll}
E_S & : \text{set of excluded items in solution } S \\
N_S^l & : \text{set of leftover nodes from cutting patterns in } C \text{ in solution } S \\
a_e(S) & : \sum_{e \in E_S} width_e \cdot height_e \\
v_l(S) & : \sum_{n \in N_S^l} value(n)
\end{array}
$$

$$
cmp(S_1, S_2) = \begin{cases}
\text{-1} & \text{if } a_e(S_1) < a_e(S_2) & \triangleright\ S_1 \textit{ superior} \\
1 & \text{if } a_e(S_1) > a_e(S_2) & \triangleright\ S_2 \textit{ superior} \\
\text{-1} & \text{if } a_e(S_1) = a_e(S_2) \text{ and } v_l(S_1) > v_l(S_2) & \triangleright\ S_1 \textit{ superior} \\
1 & \text{if } a_e(S_1) = a_e(S_2) \text{ and } v_l(S_1) < v_l(S_2) & \triangleright\ S_2 \textit{ superior} \\
0 & \text{otherwise} & \triangleright\ \textit{equal}
\end{cases} \tag{2.4}
$$

A solution is always superior if it excludes less item area. If two solutions include the same item area, the one with the highest total leftover value is superior. Therefore, leftover value can be viewed as a tiebreaker in the evaluation of solutions. Note that, despite the fact that the objective is to minimize the total bin area, this factor is not explicitly included in the cost function. Total bin area is irrelevant so long as it remains below the limit. As mentioned in Section 2.4.1, the limit is lowered every time a new feasible solution is reached.

2.4.4.2 Late-Acceptance Hill-Climbing

`GDRR` employs the Late Acceptance Hill-Climbing (LAHC) metaheuristic, introduced by Burke and Bykov [17]. This local search algorithm accepts non-improving moves when a candidate solution is better than the best solution a certain number of iterations ago. The pseudocode in Algorithm 3 shows how LAHC is implemented.

The main argument for using LAHC is its scale independence. Given that LAHC is insensitive to the scale of the objective function, the metaheuristic does not need to be tuned on an instance-specific basis. This is a major advantage over metaheuristics that make use of cooling schemes, such as Simulated Annealing.

Furthermore, since solutions are accepted based on their superiority with respect to previous solutions, no absolute difference in cost is needed. The only requirement is being able to compare two costs. No delta value is necessary. This allows for the easy use of tiebreakers, such as the one outlined in Section 2.4.4.1, where the amount of included item area always dominates leftover value.

The final argument for using LAHC is that there is only a single parameter to tune: the history length ($\mathcal{L}_h$). This parameter defines how far back to look when evaluating a solution.

Algorithm 3 provides a high-level overview of `GDRR`. As mentioned in Section 2.4.1, the heuristic will continuously attempt to fit all items inside the bins while respecting the bin area limit constraint. This limit is lowered each time the heuristic is able to construct a feasible solution.

The LAHC fitness array is initially filled entirely with the starting solution (line 2). This array keeps track of the last $\mathcal{L}_h$ best solutions. During each local search iteration, the solution is partially destroyed and rebuilt (lines 6-7) according to the procedures described in Sections 2.4.2 and 2.4.3. Index v corresponds to the index of the first element in the fitness array. Instead of shifting all elements of the fitness array every time a new solution is accepted, the head of the array is moved (line 8). This is similar to how a circular buffer works. In order for a modified solution (S') to be accepted, it must either be superior or equal to the best solution $\mathcal{L}_h$ iterations ago (S_v) or improve upon the local optimum (S^*) (lines 9-10). The fitness array is only updated with improving values (line 11-12). The implementation of LAHC is analogous to that outlined in Burke and Bykov [17], but with one major exception. Counter i is only incremented when a solution is accepted (line 13). This is in contrast to Burke and Bykov's original LAHC, where the counter is always incremented. The main motivation behind this decision is to avoid the heuristic from becoming a pure hill-climbing algorithm in later stages and thus easily getting stuck in local optima.

Algorithm 3 GDRR

Input: A starting solution $S = \{C, E\}$ where C is a set of cutting patterns and E is a set of excluded items, bin area limit $\mathcal{A}_{lim}$, history length $\mathcal{L}_h$, the average number of nodes to remove μ, and the current best solution S_{best}
Output: The best (complete) solution S_{best}

1: **function** GDRR($S, \mathcal{A}_{lim}, \mathcal{L}_h, \mu, S_{best}$)
2: **for all** $k \in \{0, ..., \mathcal{L}_h - 1\}$ **do** $S_k \leftarrow S$
3: $i \leftarrow 0$
4: $S^* = \{C^*, E^*\} \leftarrow S$ ▷ *local optimum (incomplete)*
5: **while** no stopping criteria met **do**
6: $S' \leftarrow$ RUIN($S^*, \mathcal{A}_{lim}, \mu$)
7: $S' \leftarrow$ RECREATE($S', \mathcal{A}_{lim}$)
8: $v \leftarrow i \bmod \mathcal{L}_h$
9: **if** $cmp(S', S_v) \leq 0$ **or** $cmp(S', S^*) \leq 0$ **then** ▷ *solution accepted*
10: $S^* \leftarrow S'$
11: **if** $cmp(S^*, S_v) < 0$ **then** ▷ *fitness array updated*
12: $S_v \leftarrow S^*$
13: $i \leftarrow i + 1$
14: **if** $E^* = \emptyset$ **then** ▷ *complete solution reached*
15: $S_{best} \leftarrow S^*$
16: $\mathcal{A}'_{lim} \leftarrow \sum_{c \in C^*} \mathcal{A}_c$
17: $S_{next} \leftarrow$ RUIN($S_{best}, \mathcal{A}'_{lim}, 0$)
18: $S_{best} \leftarrow$ GDRR($S_{next}, \mathcal{A}'_{lim}, \mathcal{L}_h, \mu, S_{best}$)
19: **return** S_{best}

If a solution is obtained that has no excluded items (line 14), `GDRR` has reached its current goal. When this occurs, the best complete solution S_{best} is updated (line 15), a new bin area limit $\mathcal{A}'_{lim}$ is calculated based on the total bin area of S_{best} (line 16) and a starting solution for the next goal S_{next} is created which complies with this new limit (line 17). Finally, `GDRR` is 'restarted' for the next goal (line 18). Once a stopping criterion is met, the best complete solution is returned (line 19).

In essence, the heuristic continuously strives towards feasibility with an iteratively lowering bin area limit. To initiate the optimization, Algorithm 3 can be called with an empty S (no cutting patterns and all items excluded) and with $\mathcal{A}_{lim}$ set to ∞. There is no need for a separate algorithm to create an initial solution.

2.4.5 Multithreading

Multithreading can help to improve the performance of the heuristic and makes use of the capabilities of modern processors. Each thread runs its own separate instance of GDRR, but a single global bin area limit is shared across all threads. This means that once one thread reaches a feasible solution, the limit is lowered across all threads. Solutions themselves are, by contrast, not shared between threads. This is mainly to preserve diversity.

The multithreaded implementation is largely analogous to Algorithm 3. The only major difference is that the bin area limit is not only lowered when the heuristic reaches a feasible solution (lines 12-14), but rather whenever any thread reaches a feasible solution at the current limit. It goes without saying that the best feasible solution among all threads is saved.

Threads are almost entirely independent and only influence each other indirectly. There are, for example, no explicit mechanisms in place to avoid duplicate searched areas. Due to the vast search space and the heuristic containing a number of stochastic elements (e.g. blinks), there is no real need for such mechanisms.

Multithreading is primarily used to maintain diversity, not necessarily to accelerate execution speed. The use of multithreading is particularly beneficial for instances with many different bins. At later stages of the optimization, solutions are often too fixed to significantly change the combination of used bins. Multithreading helps overcome this issue since different threads can have solutions that are close in terms of quality, but consist of very different combinations of bins. This level of diversity would be difficult to achieve with only a single thread, as it would require very extreme ruin procedures.

2.5 Computational Results

To evaluate the performance of GDRR, the algorithm was tested on a number of different benchmarks from the literature. These benchmarks contain datasets with both identical and heterogeneous sets of bins. Oliveira et al. [75] provides a central repository for a large number of cutting and packing datasets from the literature. All datasets included in the repository are formatted in the same structure, which saves a lot of time when testing across a number of different benchmarks.

Section 2.5.1 defines how the parameters of GDRR were configured for the computational experiments. The performance of the algorithm is then evaluated in Sections 2.5.2-2.5.5. Finally, Sections 2.5.6 and 2.5.7 analyze the impact of multithreading and calculation time on the performance of GDRR. All experiments were conducted on an Intel Xeon Gold 6240 CPU (2.6GHz) with 8GB of system memory. For the comparisons in Sections 2.5.2-2.5.5, GDRR was configured to use 8 threads and 600 seconds of runtime. Solutions for all datasets tested in the comparison are included as an online supplement [43] and an open-source reimplementation of GDRR has been made available [38].

2.5.1 Parameters

GDRR has a number of parameters which must be configured. The tuning of these parameters was conducted using the dataset from Ortmann et al. [76]. This dataset was chosen because it contains a large variety of instances and has variable-sized bins. Heterogeneous bins enable gradual improvements in solution quality to be visible. All parameters are tuned to achieve best results when optimizing for 600 seconds and 8 threads. Parameters are fixed and varied one by one to examine their individual effect on the behavior of the algorithm. However, the history length ($\mathcal{L}_h$) and the average number of removed nodes (μ) were tuned as a pair since they are so closely related. The parameters listed below are used throughout the experiments:

- **Leftover valuation power:** α
  In Section 2.4.3.1, parameter α was defined as the power with which to multiply the area of a leftover node in order to derive its value. The leftover value is not only used to evaluate insertion options, but also as a tiebreaker when determining a solution's quality (Section 2.4.4.1). A greater α will favor few, but large, leftover areas. When α is smaller, many medium-sized leftover pieces are not penalized as much. Experiments show that the value of α has a minimal impact so long as it is greater than 1. A value of 1.2 resulted in the best overall performance.
  - $\alpha = 1.2$
- **History Length:** $\mathcal{L}_h$
  $\mathcal{L}_h$ determines the length of the history queue in LAHC. The best value for $\mathcal{L}_h$ depends largely on the size of the instance. The larger an instance, the shorter the optimal history length. This is because larger instances result in less ruin and recreate iterations per second and generally have more diverse neighborhoods. Preliminary experiments resulted in the following rules of thumb:

  - $0 < \#items \leq 100$: $\mathcal{L}_h = 2000$
  - $100 < \#items \leq 300$: $\mathcal{L}_h = 1000$
  - $300 < \#items \leq 500$: $\mathcal{L}_h = 500$

- **Average number of removed nodes:** μ
  As with $\mathcal{L}_h$, the optimal value of μ depends on the instance size. Ruining the solution less significantly results in more iterations per second, which is beneficial for larger instances. Preliminary experiments showed that the following values for μ are a suitable choice:

  - $0 < \#items \leq 100$: $\mu = 8$
  - $100 < \#items \leq 300$: $\mu = 6$
  - $300 < \#items \leq 500$: $\mu = 4$

- **Blink rate :** β
  Across the dataset, a blinking chance of 5% was deemed to be a good balance between greediness and randomness.

  - $\beta = 0.05$

2.5.2 Results for 2BP|O|G

To compare the performance of GDRR for the 2BP with guillotine cuts and no rotation of items, the well-known instances from Berkey and Wang [13] (classes 1-6) and Lodi et al. [63] (classes 7-10) are used. Each class has ten instances of 20, 40, 60, 80 and 100 items. As a result, the dataset contains a total of 500 instances.

A comparison between GDRR and other heuristics in the literature is presented in Table 2.1. The sum of bins for each instance category is displayed. As Cui et al. [25] note, the majority of instances from this dataset have already been solved to optimality. Therefore, the differences between algorithms might seem small. The comparison includes a partial enumeration heuristic (ENH) by Lodi et al. [65], a constructive heuristic (CHBP) by Charalambous and Fleszar [18], a hybrid heuristic algorithm (HHA) by Hong et al. [50], the critical fit insertion heuristic (CFIH) by Fleszar [34] and the hybrid heuristic with triple block patterns (HHTB) by Cui et al. [25]. For all of these heuristics, the best results reported in each publication are presented in the comparison.

Table 2.1: Results for the 2DBP|O|G benchmark from Berkey and Wang [13] and Lodi et al. [63]

Class	Number of items	GDRR	ENH [65]	CHBP [18]	HHA [50]	CFIH [34]	HHTB [25]
1	20	**71**	**71**	**71**	**71**	**71**	**71**
	40	**134**	**134**	**134**	**134**	135	**134**
	60	**200**	**200**	201	201	201	**200**
	80	**275**	**275**	**275**	**275**	**275**	**275**
	100	**317**	**317**	321	320	322	**317**
	All	**997**	**997**	1002	1001	1004	**997**
2	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	20	20	**19**	20	20
	60	**25**	**25**	26	**25**	27	**25**
	80	**31**	32	33	**31**	32	**31**
	100	**39**	**39**	**39**	**39**	40	**39**
	All	**124**	126	128	**124**	129	125
3	20	**52**	54	**52**	**52**	53	**52**
	40	**94**	96	97	**94**	96	**94**
	60	**139**	140	140	140	141	**139**
	80	**189**	190	196	192	195	**189**
	100	**223**	225	230	228	226	**223**
	All	**697**	705	715	706	711	**697**
4	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	**19**	**19**	**19**	**19**	**19**
	60	**24**	25	25	25	25	25
	80	**31**	33	33	32	33	**31**
	100	**37**	39	39	38	39	**37**
	All	**121**	126	126	124	126	122
5	20	**65**	66	**65**	**65**	66	**65**
	40	**119**	**119**	121	**119**	120	**119**
	60	**180**	181	183	181	182	**180**
	80	**247**	**247**	**247**	**247**	248	**247**
	100	**282**	286	288	287	290	**282**
	All	**893**	899	904	899	906	**893**
6	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**17**	19	19	18	19	**17**
	60	**21**	22	22	22	22	**21**
	80	**30**	**30**	**30**	**30**	**30**	**30**
	100	**32**	35	34	35	34	**32**
	All	**110**	116	115	115	115	**110**
7	20	**55**	**55**	**55**	**55**	56	**55**
	40	**111**	113	112	**111**	115	**111**
	60	**158**	159	160	159	161	**158**
	80	**230**	232	233	232	232	231
	100	**271**	275	275	273	274	**271**
	All	**825**	834	835	830	838	826
8	20	**58**	**58**	**58**	**58**	60	**58**
	40	**113**	**113**	114	**113**	116	**113**
	60	**161**	162	163	162	165	**161**
	80	**224**	226	226	225	227	**224**
	100	**277**	280	279	279	281	**277**
	All	**833**	839	840	837	849	**833**
9	20	**143**	**143**	**143**	**143**	**143**	**143**
	40	**278**	**278**	279	**278**	**278**	**278**
	60	**437**	**437**	438	**437**	**437**	**437**
	80	**577**	**577**	**577**	**577**	**577**	**577**
	100	**695**	**695**	**695**	**695**	**695**	**695**
	All	**2130**	**2130**	2132	**2130**	**2130**	**2130**
10	20	**42**	44	44	43	43	43
	40	**74**	**74**	**74**	**74**	75	**74**
	60	**101**	102	103	102	104	**101**
	80	**128**	130	130	130	132	**128**
	100	**158**	159	163	160	163	159
	All	**503**	509	514	509	517	505
Total bins:		**7233**	7281	7311	7275	7325	7238

The proposed heuristic achieves the best scores in each category. However, it should be noted that, with the exception of ENH, GDRR has longer runtimes than most of the other algorithms. Since many of these algorithms are based heavily on constructive approaches their runtimes are around or under the 1-minute mark.

2.5.3 Results for 2BP|R|G

To compare the performance of the heuristic for the 2BP with guillotine cuts and 90° rotation of items, once again the dataset by Berkey and Wang [13] and Lodi et al. [63] is used. In addition to some of the heuristics from Section 2.5.2, this comparison also includes an agent-based approach (A-B) by Polyakovsky and M'Hallah [79] and the sequential value correcting heuristic (SVCRG) by Cui et al. [24]. The results are summarized in Table 2.2.

Unlike in Section 2.5.2, GDRR does not always come out on top here. On average the proposed heuristic achieves the best quality solutions, but there are some classes where HHTB is superior.

2.5.4 Results for 2VSBP|O|G

The performance on instances with a heterogeneous set of bins is tested using three well-known benchmarks. In this comparison, the average density is compared across different algorithms. Density ρ corresponds to the percentage of area of the bins that are packed with items. A density of 95%, for example, corresponds to 5% wasted bin area.

$$\rho = \frac{\text{total item area}}{\text{total bin area}} \cdot 100\%$$

All the results from other heuristics in the tables are taken from the comparison performed by Hong et al. [50]. Table 2.3 provides a comparison for the instances created by Hopper and Turton [51] which consist of three problem categories, each containing five instances. The second dataset was introduced by Pisinger and Sigurd [78] and is divided into ten classes, the results for which are summarized in Table 2.4. Note that this dataset does not define limits on the quantity of bins of each type. The last dataset, introduced by Ortmann et al. [76], contains instances based on nice (relatively normal) and pathologically (unusual, more extreme) sized items. These instances range from 25-500 items and 2-6 different bin sizes. Table 2.5 provides the results compared to other heuristics. For all of these datasets and classes, GDRR always achieves the best solutions in terms of quality.

Table 2.2: Results for the 2DBP|R|G benchmark from Berkey and Wang [13] and Lodi et al. [63]

Class	Number of items	GDRR	CHBP [18]	A-B [79]	SVCRG [24]	CFIH [34]	HHTB [25]
1	20	**66**	**66**	**66**	**66**	**66**	**66**
	40	**128**	129	129	**128**	129	**128**
	60	**195**	**195**	**195**	**195**	**195**	**195**
	80	**270**	271	**270**	**270**	271	**270**
	100	**313**	314	314	314	314	**313**
	All	**972**	975	974	973	975	**972**
2	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	**19**	**19**	20	**19**	20
	60	**25**	**25**	**25**	29	**25**	**25**
	80	**31**	**31**	**31**	33	32	**31**
	100	**39**	**39**	**39**	40	**39**	**39**
	All	**124**	**124**	**124**	132	125	125
3	20	**47**	48	49	**47**	48	**47**
	40	**92**	94	94	**92**	94	**92**
	60	**134**	136	137	135	136	**134**
	80	**183**	186	188	184	185	**183**
	100	**219**	223	223	223	223	**219**
	All	**675**	687	691	681	686	**675**
4	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	**19**	**19**	**19**	**19**	**19**
	60	**23**	25	25	25	25	25
	80	**30**	33	31	31	33	31
	100	**37**	38	**37**	38	38	**37**
	All	**119**	125	122	123	125	122
5	20	**59**	**59**	60	**59**	**59**	**59**
	40	**114**	115	115	**114**	117	**114**
	60	**173**	176	176	174	175	**173**
	80	239	240	243	240	241	**238**
	100	**277**	282	284	281	281	**277**
	All	862	872	878	868	873	**861**
6	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**16**	18	18	19	18	18
	60	**21**	**21**	**21**	**21**	**21**	**21**
	80	**30**	**30**	**30**	**30**	**30**	**30**
	100	**32**	34	34	34	34	**32**
	All	**109**	113	113	114	113	111
7	20	**52**	**52**	**52**	**52**	**52**	**52**
	40	102	104	103	103	106	**101**
	60	**145**	148	148	**145**	151	**145**
	80	208	211	211	210	214	**207**
	100	249	255	252	251	258	**248**
	All	756	770	766	761	781	**753**
8	20	**53**	**53**	**53**	**53**	**53**	**53**
	40	104	105	104	104	105	**103**
	60	**146**	150	150	147	152	**146**
	80	204	210	209	207	211	**203**
	100	**252**	258	256	254	258	**252**
	All	759	776	772	765	779	**757**
9	20	**143**	**143**	**143**	**143**	**143**	**143**
	40	**275**	**275**	**275**	**275**	**275**	**275**
	60	**435**	**435**	**435**	**435**	**435**	**435**
	80	**573**	**573**	**573**	**573**	**573**	**573**
	100	**693**	**693**	**693**	**693**	**693**	**693**
	All	**2119**	**2119**	**2119**	**2119**	**2119**	**2119**
10	20	**41**	**41**	**41**	**41**	42	**41**
	40	**72**	73	73	73	73	73
	60	**99**	100	100	100	101	**99**
	80	**124**	130	129	126	129	126
	100	**155**	159	161	159	159	**155**
	All	**491**	503	504	499	504	494
Total bins:		**6986**	7064	7063	7035	7080	6989

Table 2.3: Results for the 2VSBP|O|G benchmark from Hopper and Turton [51]

Class	GDRR	FFDH[21]	BFDH[22]	JOIN[68]	FCOG[63]	BFDH*[14]	SAS[72]	SASm[76]	BFS[76]	HHA[50]
M1	**98.4**	93.5	93.5	86.8	94.9	94.9	83.5	91.6	95.5	**98.4**
M2	**97.2**	87.5	88.8	82.8	89.6	89.6	81.7	88.0	90.0	95.6
M3	**98.0**	92.0	92.6	85.5	93.6	93.6	86.8	91.8	94.9	97.4
$\mu[\rho]$:	**97.85**	91.00	91.63	85.03	92.70	92.70	84.00	90.47	93.47	97.13

Note: $\mu[\rho]$ is the average density of the solutions as a percentage.

Table 2.4: Results for the 2VSBP|O|G benchmark from Pisinger and Sigurd [78]

Class	GDRR	FFDH[21]	BFDH[22]	JOIN[68]	FCOG[63]	BFDH*[14]	SAS[72]	SASm[76]	BFS[76]	HHA[50]
I	**94.1**	86.3	86.6	83.0	87.3	87.4	79.4	86.3	88.6	91.5
II	**96.5**	83.7	83.7	80.9	85.2	85.0	80.1	82.2	85.1	96.3
III	**91.5**	81.1	81.7	76.7	81.9	81.9	69.6	75.7	82.2	86.6
IV	**93.5**	80.0	80.0	79.1	82.7	82.1	76.0	78.0	82.0	91.7
V	**89.3**	80.4	81.1	77.6	81.3	81.2	72.6	78.3	81.4	84.6
VI	**92.7**	79.1	79.3	78.3	80.7	80.5	76.1	77.1	79.5	90.2
VII	**90.1**	79.9	80.2	79.6	80.8	80.6	74.0	80.4	80.9	86.9
VIII	**89.4**	80.7	81.1	74.2	81.3	81.2	76.4	79.5	81.6	85.9
IX	**75.6**	72.8	72.6	72.1	72.6	72.8	71.6	72.9	73.0	74.3
X	**93.0**	83.3	83.8	79.3	85.4	84.6	73.2	79.4	85.5	90.3
$\mu[\rho]$:	**90.66**	80.73	81.01	78.08	81.92	81.73	74.90	78.98	81.98	87.83

Note: $\mu[\rho]$ is the average density of the solutions as a percentage.

Table 2.5: Results for the 2VSBP|O|G benchmark from Ortmann et al. [76]

Class	GDRR	FFDH[21]	BFDH[22]	JOIN[68]	FCOG[63]	BFDH*[14]	SAS[72]	SASm[76]	BFS[76]	HHA[50]
Nice25i	**99.8**	73.9	73.6	70.6	73.9	73.6	68.3	71.8	73.6	94.3
Nice50i	**93.5**	76.7	76.7	73.3	77.9	77.7	70.8	73.1	77.8	89.2
Nice100i	**91.8**	79.4	79.4	77.5	79.9	79.4	75.7	76.3	79.4	88.3
Nice200i	**92.6**	82.0	82.0	81.5	84.5	84.5	78.5	79.4	84.5	91.0
Nice300i	**93.3**	85.8	85.8	83.3	86.0	86.5	80.2	81.7	86.8	92.0
Nice400i	**94.3**	85.1	85.1	82.7	86.6	85.7	80.2	81.0	85.7	92.8
Nice500i	**94.1**	87.2	87.2	84.8	87.7	87.7	81.8	83.8	87.7	93.1
Nice	**94.04**	81.44	81.40	79.10	82.35	82.15	76.50	78.15	82.21	91.52
Path25i	**100**	76.3	76.3	73.9	77.9	77.6	72.2	74.4	78.3	97.2
Path50i	**98.1**	76.4	78.6	74.2	81.6	79.4	72.4	75.6	81.9	94.9
Path100i	**95.5**	79.7	79.7	77.8	83.2	81.3	72.4	75.8	83.5	93.5
Path200i	**94.6**	84.1	84.0	81.8	88.0	85.9	77.7	79.0	87.5	93.3
Path300i	**95.5**	82.9	82.9	82.7	87.0	86.0	81.4	81.7	87.3	94.9
Path400i	**95.4**	82.7	82.7	82.3	89.6	87.0	79.9	80.1	88.5	94.8
Path500i	**96.6**	82.9	82.9	81.2	88.7	86.4	81.3	82.4	86.7	94.0
Path	**96.43**	80.71	81.01	79.13	85.14	83.37	76.76	78.43	84.81	94.66
$\mu[\rho]$:	**95.24**	81.22	81.35	79.26	83.89	82.91	76.78	78.43	83.66	93.10

Note: $\mu[\rho]$ is the average density of the solutions as a percentage.

2.5.5 Results for 2VSBP|R|G

To the best of our knowledge, there are currently no algorithms available in the literature with which results for the 2BP variant with 90° rotation and heterogeneous sets of bins can be compared. Since `GDRR` supports this variant, experiments were run on the same three datasets in Section 2.5.4. The solutions are included in the online supplement [43] to facilitate comparisons in future research.

2.5.6 Impact of Multiple Threads

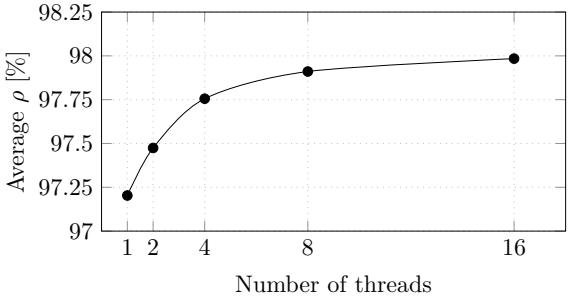


Figure 2.7: Impact of multithreading on solution quality (600s of runtime)

To test the impact of multithreading on the solution quality, experiments were run with a subset of instances from Ortmann et al. [76]. From each class, the first instance with 6 different bins was selected. This decision was made because a greater variety of bins allows for more gradual differences in solution quality to be visible. The results are shown in Figure 2.7. As expected, the first few extra threads result in a significant performance improvement. Increasing the number of threads from one to four, for example, results in an improvement of 0.5% in average density. However, adding additional threads quickly leads to diminishing returns.

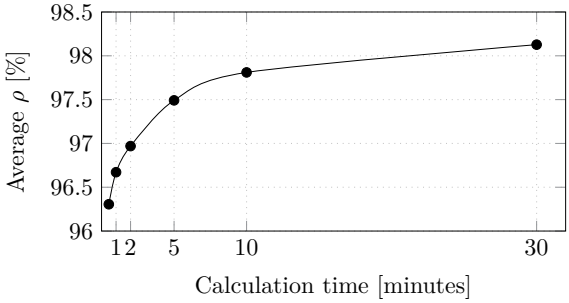


Figure 2.8: Impact of calculation time on solution quality (8 threads)

2.5.7 Impact of Calculation Time

The impact of available calculation time on solution quality was tested using the same subset of instances detailed in Section 2.5.6. Figure 2.8 shows how the heuristic scales with respect to calculation time. Despite improvements slowing down, there are still gains to be had, even after 30 minutes. This is most likely because the history length ($\mathcal{L}_h$) was scaled with respect to the calculation time. When running `GDRR` for 5 minutes, for example, $\mathcal{L}_h$ was set to half of the value described in Section 2.5.1.

2.6 Conclusion

This chapter introduced `GDRR`, a heuristic for solving the 2D bin packing problem with guillotine constraints. Variants of the problem with a heterogeneous set of bins (variable-sized) and 90° rotation of items are also supported. The heuristic contains a ruin and recreate procedure which iteratively attempts to improve the solution. The search can be described as being goal-driven since it continuously strives to create feasible solutions with an ever decreasing limit of available bin area. To escape local optima, `GDRR` employs the Late Acceptance Hill-Climbing metaheuristic. Unlike most other heuristics for this problem, the focus lies primarily on the improvement phase rather than the constructive aspect. It is capable of consistently producing the best results in terms of solution quality across several benchmarks in the literature.

Future research may include exploring the 2BP variant without guillotine constraint and other cutting and packing problems with real-world constraints. Finally, it would also be interesting to investigate the applicability of a goal-driven approach in other types of problems.

Part II

Irregular Cutting & Packing Problems

Chapter 3

Decoupling Geometry from Optimization in 2D Nesting Problems: an Open-Source Collision Detection Engine

This chapter is a minimally edited version of the following published work:

J. Gardeyn, G. Vanden Berghe, and T. Wauters. "Decoupling Geometry from Optimization in 2D Irregular Cutting and Packing Problems: an Open-Source Collision Detection Engine". In: *INFORMS Journal on Computing* (2025). article in advance. DOI: 10.1287/ijoc.2024.1025

This chapter is presented together with the following open-source software:

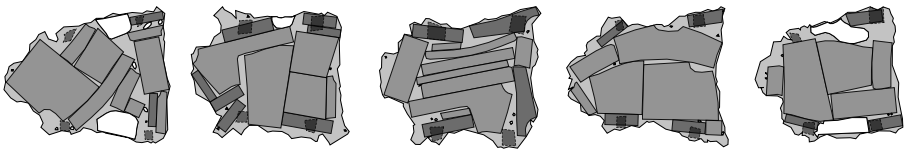
J. Gardeyn. "`jagua-rs`: a fast and fearless Collision Detection Engine for 2D nesting problems". 2024. URL: https://github.com/JeroenGar/jagua-rs

3.1 Introduction

2D cutting and packing (C&P) problems involve placing a set of smaller items within the bounds of a larger container. This broad class of optimization problems can be very challenging to solve, especially when the items or containers are *irregular* (non-rectangular) in shape. This difficulty has led to a highly active yet fractured research domain featuring many approaches to many problem variants.

While the objectives and particulars of these irregular C&P — or *nesting* — problems can differ, they all share the need for a specific feasibility check: determining whether or not an item can be cut from or placed at a certain position without overlapping with anything else. Unlike, say, the traveling salesman problem in which feasibility simply means ensuring all locations are visited, here the irregularity of the items and containers makes implementing this check in a precise and efficient manner a challenging endeavor. Therefore, in addition to the **optimization challenge** of finding a good solution according to some objective function, irregular C&P problems also pose an additional **geometric challenge** of accurately and efficiently verifying the feasibility of candidate solutions.

Figure 3.1: Example solution of a leather nesting instance.



Note: Dataset introduced by Baldacci et al. [7].

By way of example, let us consider the leather nesting problem depicted in Figure 3.1. The aim is to use the leather hides as efficiently as possible (the optimization challenge), while ensuring that all items are fully contained within the hides and are not colliding with each other or any defects (the geometric challenge). When a human is tasked to solve such a problem, they can rely on their innate spatial reasoning skills to ensure no *collisions* occur and even subconsciously rule out many invalid placements before ever considering them. However, translating this particular human intuition into a robust and efficient algorithm is far from trivial. The general problem of collision detection also presents a significant challenge in other domains, such as computer graphics and simulations. Indeed, this seemingly simple problem is so challenging that entire books have been dedicated to the subject [30].

From a conceptual and algorithmic perspective, the geometric and optimization aspects of such problems are clearly distinct: the feasibility of a solution implies nothing about its quality or the strategy used to generate it. At present, however, there are no publicly available tools to handle collision detection in irregular 2D C&P problems, meaning that researchers and practitioners need to simultaneously address the geometric and the optimization challenges. This is an inherently inefficient situation that forces researchers to spend a significant proportion of their time reinventing the wheel. Not only do they need to address the optimization intricacies of their new 2D irregular C&P variant or the methodological challenges of developing a new approach for an existing problem, but they also need to spend effort integrating this with a feasibility check despite the fact that this geometric challenge is a feature of all problem variants. Besides the wasted effort, this also creates a significant barrier to entry that impedes further advances in the field.

In this chapter, we will decouple the geometric from the optimization challenge by introducing an open-source, easy-to-use and adaptable *collision detection engine* (CDE) that is capable of handling the geometric component of 2D irregular C&P problems. The primary aim of the CDE is to (in)validate potential item placements as efficiently and accurately as possible.

We envisage two target audiences for this tool. First, there are those who simply want to focus on the combinatorial aspects of their 2D irregular C&P problem at hand. They stand to benefit greatly from having a freely available engine to incorporate into their own methodology, which essentially outsources the geometric challenge and enables them to focus their efforts on developing smart optimization algorithms. The second target audience are those who, rather than solving optimization problems themselves, might have good ideas concerning how to further refine or extend this open-source project. Our CDE will therefore not represent an endpoint, but should instead serve as the starting point for these two target audiences: either a tool to be readily used or one to be further improved upon.

The remainder of this chapter is structured as follows. In Section 3.2 we review the general approaches to collision detection, highlighting their strengths and limitations. The main body of the chapter is then dedicated to the components of the engine itself, outlined in Section 3.3, followed by a set of core principles and design philosophies which are each time accompanied by concrete implementations in Sections 3.4-3.8. Finally, Sections 3.9-3.10 introduce our open-source CDE and discuss the factors that affect its performance.

3.2 General Approaches to Geometry in 2D C&P Problems

The literature contains many approaches to deal with the geometric challenge of 2D irregular C&P problems. The terms *collide*, *overlap* and *intersect* are used interchangeably in the academic literature, but all refer to the same concept: two entities occupying the same space. This chapter will primarily use the word *collision*.

A valuable introduction to geometry in irregular 2D C&P and an overview of the most frequently used methods is provided by Bennell and Oliveira [10]. In the interest of brevity, we will not exhaustively repeat the content of the tutorial. Instead, we provide a short summary of the most relevant approaches reviewed by Bennell and Oliveira [10] along with a discussion of their main strengths and weaknesses. The three most common approaches for dealing with geometry in irregular C&P problems are: **raster**, **no-fit polygon** and **trigonometry**.

The simplest method is arguably the raster approach, where irregular shapes are discretized and represented as a set of pixels within a grid. Collision detection is performed by simply checking the occupation of the pixels. This strategy is flexible, robust and relatively straightforward to implement efficiently. To guarantee feasibility, discretizing an irregular shape into a set of pixels inevitably requires containers to be deflated and items to be inflated. As a result, a low grid resolution can severely affect the maximum achievable solution quality. However, increasing the resolution is accompanied by a significant rise in computational cost and memory usage. Hence, determining a suitable grid resolution is a delicate balancing act. Furthermore, each possible rotation of an item requires a distinct discretization, which can become problematic when continuous rotation of items is allowed.

A second commonly used approach is the no-fit polygon (NFP), which encompasses the set of locations where two polygons will collide. A unique NFP must be generated for each pair of polygons and every possible rotation. Detecting collisions between two polygons entails checking whether or not a single reference point of one of the polygons lies inside the respective NFP. Section 3.4 will clarify why this is a relatively fast operation. Additionally, the boundary of the NFP defines where two polygons touch each other. This is very valuable information in an optimization context, where compact solutions are likely being targeted. Similar to the raster approach, NFPs are not particularly suitable when rotation freedom is high. For n polygons with m possible rotations, $(m * n)^2$ unique NFPs need to be computed.

The most significant challenge for this approach undoubtably remains the development of robust NFP generators, particularly for non-convex and complex shapes. Over the years, numerous papers have been published claiming to improve NFP generation robustness or efficiency. Yet, at the time of writing this article, a publicly available implementation without significant shortcomings remains elusive. Even the authors of the most popular NFP generator on GitHub [69] acknowledge that their implementation is not feature-complete and only works well for shapes without holes or concavities.

The third general approach for dealing with geometry in irregular C&P problems is trigonometry. Its precision, flexibility and simplicity make it an attractive choice. However, approaches that rely directly on trigonometry are often quickly written off as being too slow for practical use. For example, Bennell and Oliveira [10], state:

> *"Hence, [trigonometry] does not lend itself to nesting algorithms based on any iterative search process since the time spent on evaluating the geometry can restrict the time, and therefore the extent, of the search. However, constructive algorithms, for instance approaches that build solutions based on a previously defined sequence of pieces, may quite efficiently use [trigonometry] to tackle the geometric issues of nesting problems."*

The general claim being made here is certainly not incorrect. By default, trigonometry is slow and highly sensitive to the number of edges in a polygon and therefore seldom employed. For this reason, most state-of-the-art solution strategies for irregular C&P problems are either based on a raster or on NFPs.

Another approach discussed by Bennell and Oliveira [10], the phi-function, also deserves to be mentioned. While interesting, the limited accessibility of this method and lack of general phi-function 'generators' have hindered its widespread adoption.

A common theme across all approaches is the severe lack of publicly available source code of the implementations. This impedes scientific progress and leads to a lot of wasted time and effort. We are determined to change the status quo and are guided by the principle of open science.

Table 3.1: Strengths and weaknesses of the general approaches for collision detection in 2D irregular C&P problems.

Approach	Precision	Robustness	Implementation ease*
Raster	⋆	⋆⋆⋆	⋆⋆⋆
NFP	⋆⋆⋆	⋆	⋆⋆
Trigonometry	⋆⋆⋆	⋆⋆⋆	⋆

Note: *to reach a sufficiently fast implementation.

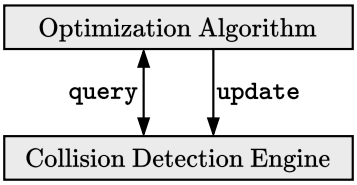
Table 3.1 presents the strengths and weaknesses of the three approaches discussed based on the criteria of precision, robustness and effort required to reach a sufficiently fast implementation. Deciding on which approach to use as the basis of our implementation is not straightforward as no approach is without shortcomings. However, given that the primary aim of our CDE is for its users to trust that they can safely delegate the geometric challenge, maximum precision and robustness are crucial. Trigonometry is the only approach which does not compromise with respect these two properties. Furthermore, increased implementation effort is a manageable drawback. The open-source nature of our proposed implementation means that the bulk of the development effort can be shared and reused. Additionally, improvements in computational efficiency can easily be quantified, allowing for incremental improvements.

3.3 Collision Detection Engine

When solving irregular C&P problems, the computational bottleneck will typically pertain to validating potential item placements, regardless of the search strategy one uses to improve the solution quality. The speed at which these feasibility checks can be performed therefore inherently constrains the design of any optimization algorithm. However, whether a solution is feasible or not implies nothing about its quality or the optimization strategy used to generate it. Decoupling these two challenges by way of a reusable and extendable *engine* to handle the geometry would enable researchers to focus with full confidence on the optimization challenge.

Figure 3.2 illustrates how a collision detection engine (CDE) fits into the process of solving a 2D irregular C&P problem. The CDE enables the geometric aspect of the problem to be abstracted away by assuming the responsibility of validating item placements. To perform this task, two fundamental interactions between the CDE and the optimization algorithm must take place.

Figure 3.2: Fundamental interaction between the CDE and an optimization algorithm for irregular C&P problems.



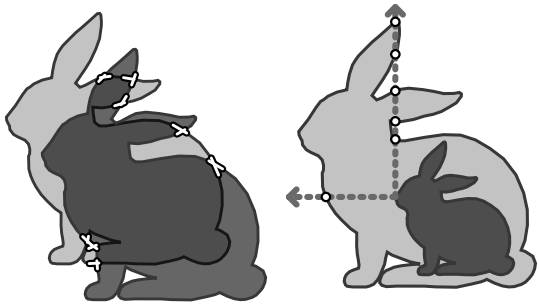
First, the CDE must be able to resolve **queries** by definitively answering whether or not an item can be placed at a certain position without causing a collision. The interior of every item is represented by a base shape ($\mathcal{S}$), with every placement of this item corresponding to a transformation of its base shape ($\mathcal{S}^T$). A placement is defined by a *rigid transformation*, which is a combination of (i) a translation along the x and y dimensions, (ii) a rotation by angle θ and (iii) a reflection (or flip). Determining if a placement is feasible involves checking if $\mathcal{S}^T$ is situated entirely inside the container and does not collide with any other items.

The second interaction concerns changes to the environment (for example placing or removing items). These changes must be communicated to the CDE so it can **update** its internal state to reflect the new environment. In practice, more sophisticated or tightly integrated interactions are possible, but Figure 3.2 outlines the bare minimum of interaction required.

Although collision detection also plays a central role in various other domains (video games, computer graphics, simulation), we cannot simply rely on any existing off-the-shelf engines. To achieve maximum performance, these engines are typically tailored to the specific needs and unique characteristics of their respective domain. By focusing on irregular C&P problems, we can make a number of assumptions which will enable us to design a specialized CDE.

First, we can assume a dense environment (unlike video games), which means that the probability of collision is high. Second, we can assume a largely static environment (unlike simulations), which means that queries will be more frequent than updates. Third, we are only interested in the boolean answer to the query. Therefore we only need to determine whether a collision occurs, but not necessarily the details concerning where and how exactly it occurs.

Figure 3.3: Types of collision between a pair of simple polygons.



(a) (b)

Note: Collision due to (a) intersecting edges or (b) inclusion.

3.4 Polygon Collision

At a fundamental level, detecting collisions entails determining whether or not two arbitrary polygons collide. More specifically, we aim to detect collisions between **simple polygons**: polygons that do not intersect themselves and have no holes. Convexity of the polygons is not a requirement. Given a simple polygon α, let us define some notation:

$$\begin{gathered} \mathcal{S}_\alpha : \text{set of all points in the interior of } \alpha \\ E_\alpha = \{..., e, ...\} : \text{set of edges of } \alpha \\ e : \text{set of all points on the corresponding edge} \end{gathered} \tag{3.1}$$

Two shapes $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ collide if a point p exists which is in both $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$. If no such point exists, the polygons do not collide:

$$\mathcal{S}_\alpha \cap \mathcal{S}_\beta = \emptyset \iff \nexists p : p \in \mathcal{S}_\alpha \ \wedge \ p \in \mathcal{S}_\beta \tag{3.2}$$

The remainder of this section describes a trigonometric approach for detecting collisions between two polygons.

Figure 3.3 illustrates the distinct types of collision that exist between two simple polygons. The first type of collision occurs when the polygons have edges that intersect each other. If there are no intersecting edges, the two polygons are either disjoint or one polygon is located entirely inside the bounds of the other. The second type of collision accounts for the second case, where one polygon is fully contained within the other. Every collision between two simple polygons can be expressed as one of these two types [80].

Figure 3.3(a) shows an example of collision due to intersecting edges, where the intersecting edges are highlighted. To detect this type of collision, all unique pairs of edges between both polygons need to be checked with a **line segment intersection** test, an efficient implementation of which was provided by Antonio [4]. Let us now define $\mathcal{E}_\alpha$ as the set of all points that lie on the boundary of simple polygon α:

$$\mathcal{E}_\alpha : \bigcup_{e \in E_\alpha} e \tag{3.3}$$

Collision due to intersecting edges does not occur if none of the edges defining one polygon intersect with any of the edges defining another:

$$\mathcal{E}_\alpha \cap \mathcal{E}_\beta = \emptyset \iff \forall e_\alpha \in E_\alpha, \forall e_\beta \in E_\beta : e_\alpha \cap e_\beta = \emptyset \tag{3.4}$$

Proving there is no collision by intersecting edges involves checking all unique pairs of edges. The computational cost of this proof scales quadratically with the number of edges: $O(|E_\alpha||E_\beta|)$. In the example depicted in Figure 3.3(a) each shape has 140 edges, resulting in a total of 19600 intersection checks required. Despite these shapes obviously colliding, it is worth noting that only a very small minority of the edge-pairs ($\sim 0.08\%$) actually intersect.

An example of collision due to inclusion is illustrated in Figure 3.3(b). It shows a smaller shape $\mathcal{S}_\mu$ colliding with a larger one $\mathcal{S}_\Omega$ without any of their edges intersecting. This is only possible when the smaller shape is fully enclosed within the larger shape ($\mathcal{S}_\mu \subseteq \mathcal{S}_\Omega$). Determining that no inclusion occurs thus requires demonstrating that there exists a point $p \in \mathcal{S}_\mu$ which is $\notin \mathcal{S}_\Omega$:

$$\mathcal{S}_\mu \not\subseteq \mathcal{S}_\Omega \iff \exists p \in \mathcal{S}_\mu : p \notin \mathcal{S}_\Omega \tag{3.5}$$

This can be achieved by performing a **point in polygon** test, with one approach involving the ray-casting algorithm introduced by Shimrat [84]. In Figure 3.3(b), two example rays are cast from a point in $\mathcal{S}_\mu$ to outside $\mathcal{S}_\Omega$. The number of intersections between the ray and the edges of $\mathcal{S}_\Omega$ indicate whether or not $p \in \mathcal{S}_\Omega$. An odd number of intersections means $p \in \mathcal{S}_\Omega$, while an even number of intersections proves $p \notin \mathcal{S}_\Omega$. In the example both rays intersect $\mathcal{S}_\Omega$ an odd number of times (1 and 5 times).

Since only cases of full inclusion remain undetected by the edge intersection test, checking a single point is sufficient to prove no inclusion occurs. Therefore, the computational cost of the inclusion test scales linearly with the number of edges of $\mathcal{S}_\Omega$. In this example, a single point in polygon test requires 140 line segment intersection tests. In practice, however, either shape could assume the role of $\mathcal{S}_\mu$ or $\mathcal{S}_\Omega$. Given that this test is noncommutative, proving there is no inclusion consequently involves two point-in-polygon tests.

In summary, two polygons do not collide if none of their edges intersect and neither polygon is fully positioned inside the other:

$$\mathcal{S}_\alpha \cap \mathcal{S}_\beta = \emptyset \iff \mathcal{E}_\alpha \cap \mathcal{E}_\beta = \emptyset \ \wedge \ \mathcal{S}_\alpha \not\subseteq \mathcal{S}_\beta \ \wedge \ \mathcal{S}_\beta \not\subseteq \mathcal{S}_\alpha \tag{3.6}$$

This is a fundamentally precise and robust way of detecting collisions between any two polygons, but computational efficiency is lacking. For the example depicted in Figure 3.3, where the shapes each have 140 edges, a naive implementation would require a total of 19880 line segment intersection tests (19600 + 140 + 140) to prove no collision occurs.

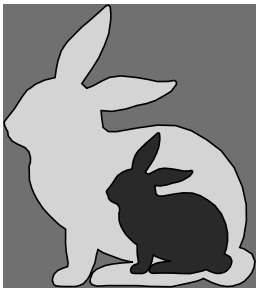
For the edge intersection test, more efficient algorithms exist. One example is by Bentley and Ottmann [11], which lowers the computational cost from a quadratic to quasi-linear time complexity. When considering potentially hundreds of polygons, however, checking individual polygons against one another still represents an unacceptable computational burden.

The following sections will therefore investigate how we can improve the efficiency of this approach while retaining its inherent precision and robustness.

3.5 Hazards

For the CDE to be easily adaptable, it should be agnostic of the specific problem variant being addressed. We therefore need a unifying concept with which to express anything that could cause a collision and render the placement of an item invalid. In this section, we introduce the concept of *hazards*: an abstraction which is capable of expressing spatial constraints in a general way.

Figure 3.4: Different types of hazards.



Note: Two distinct hazards are present. The container induces an ■ exterior hazard and the item induces an ■ interior hazard.

Figure 3.4 demonstrates the concept of hazards with an example of a container (the large bunny) inside of which an item (the smaller bunny) is placed. Two distinct hazardous regions are present: one induced by the container and another induced by the item. Since items must be fully situated inside the container, the hazard induced by the container refers to all of its **exterior** locations. Meanwhile, the hazard induced by the item refers to every location within its **interior**.

In summary, the shape of hazard h induced by entity e is defined as:

$$\mathcal{S}_h = \begin{cases} \mathcal{S}_e & \text{if } e \text{ induces an interior hazard} \\ \mathcal{S}_e^{\complement} & \text{if } e \text{ induces an exterior hazard} \end{cases} \tag{3.7}$$

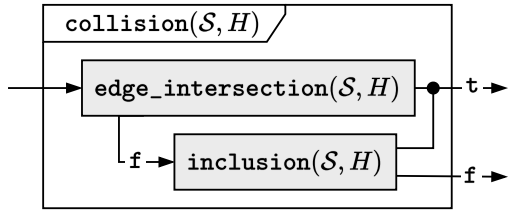
In case of interior hazards, the shape of the hazard is identical to the entity that induces it (an item, for example). For exterior hazards, its shape is the complement of the entity which induces it. Placing an item i with transformed shape $\mathcal{S}_i^T$ into a container with hazard set H is feasible if it does not collide with any of the existing hazards:

$$\mathcal{S}_H = \bigcup_{h \in H} \mathcal{S}_h \tag{3.8}$$

$$\mathcal{S}_i^T \cap \mathcal{S}_H = \emptyset \iff \forall h \in H : \mathcal{S}_i^T \cap \mathcal{S}_h = \emptyset \tag{3.9}$$

The concept of hazards can be further expanded to cover additional constraints regarding the validity of item placements. A quality zone for example, as first explored by Heistermann and Lengauer [49], is a part of the container where the quality differs (lower or higher) from the rest. Such zones can easily be represented as hazards, where zones with inferior or superior quality are converted into interior and exterior hazards, respectively. In a problem context where each item i must be produced from a minimum quality level, not all hazards are universally applicable. When validating a placement, all hazards induced by higher-quality zones than the item requires may safely be ignored and only those induced by lower quality regions need to be taken into account. In this context, each item i thus has its own set of *relevant* hazards: $H_i \subseteq H$.

Figure 3.5: The `collision` function.



Note: Returns whether shape $\mathcal{S}$ collides with any hazards $h \in H$, `t`: true, `f`: false.

3.6 CDE Implementation

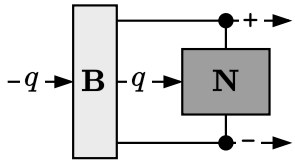
The state of the CDE is defined by its set of registered hazards (H). Figure 3.5 provides a diagram of the `collision` function, which is used to query the CDE. When validating the placement of an item i, its transformed shape ($\mathcal{S}_i^T$) can be passed to this function to determine if it collides with any hazard $h \in H$.

The remainder of this section details how we implemented the CDE and highlights the interactions between its various components. All efficiency enhancements introduced throughout this section yield results that are identical to those produced by the naive implementation from Section 3.4.

3.6.1 Two-phased Approach

Many collision detection techniques in computer graphics and simulations employ a two-phased approach, which splits a procedure into a broad and a narrow phase. The broad phase is responsible for eliminating the bulk of the work, using relatively fast algorithms. The remaining work, which could not be inexpensively eliminated, is then resolved using time-consuming algorithms within a narrow phase.

Figure 3.6: An abstract representation of the two-phased approach.



Note: **B**: broad phase, **N**: narrow phase, q: query.

Figure 3.6 illustrates the interaction between these two phases. Assume we have an abstract query (q) which can be resolved to either a positive ($+$) or negative ($-$) answer. The query is first sent through the broad phase (**B**), which is often able to swiftly resolve it. However, the quick checks employed by the broad phase are not always able to provide a definitive answer. In such cases the query must be passed on to the narrow phase (**N**), which is guaranteed to produce a definitive answer, albeit at a higher computational cost.

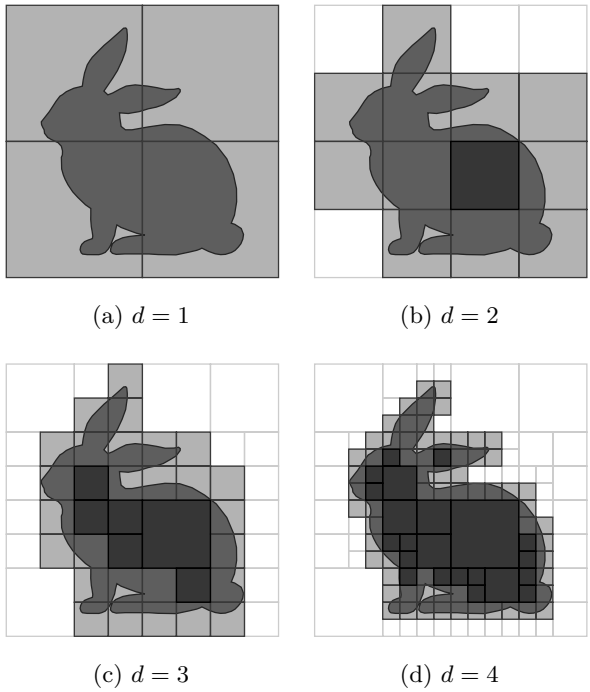
The narrow phase encompasses the exact checks outlined in Section 3.4: line segment intersection tests and point in polygon tests. What remains outstanding is designing a broad phase that focuses on maximizing overall performance. This broad phase should strike the right balance between the ratio of queries it can resolve and its associated average computation cost. The following sections will explore the two-phased approach in the context of the `edge_intersection` and `inclusion` procedures from Figure 3.5.

3.6.2 Quadtree

The broad phase of our CDE revolves around a data structure based on quadtrees to represent the hazards. Introduced by Finkel and Bentley [32], quadtrees are tree-based data structures which recursively divide a two-dimensional space into quadrants. They are used in computer graphics to efficiently store and query spatial data, with a practical example being image compression [85]. Their efficiency stems from the fact that they provide high resolution only in areas where it is needed and do not waste any resources in regions where it is not required.

Figure 3.7 illustrates a quadtree for a single (interior) hazard. The four illustrations depict the data structure with varying maximum tree depths (d). Only the leaf nodes of the tree are visualized, and their colors correspond to the degree to which a hazard is present. As depth increases, only nodes which partially contain hazards are subdivided into four children (if $< d$). Nodes that do not feature any hazards or those fully occupied by them do not require any additional resolution provided by the extra depth. In case of partially present hazards, additional information such as which edges of the hazard intersect with the respective node are also stored. Although not shown in Figure 3.7, it is important to note that a single node can contain information concerning multiple hazards.

Figure 3.7: Quadtree for an interior hazard with varying maximum depths d.



(a) $d = 1$ (b) $d = 2$

(c) $d = 3$ (d) $d = 4$

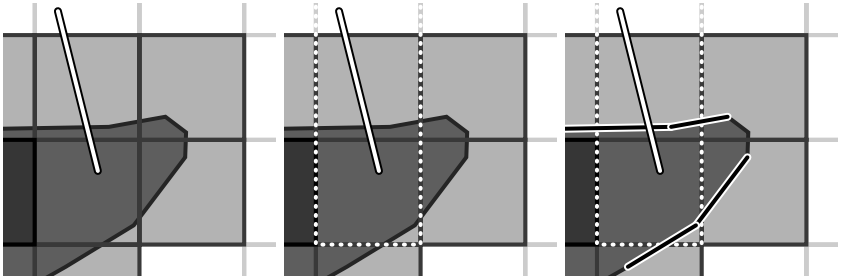
Note: Hazard is [□ not, ▩ partially, ■ fully] present in the node.

The quadtree can be queried to efficiently narrow down which nodes (and hazards) are relevant for a given query. The quadtree is part of the internal state of the CDE and must therefore be updated whenever the hazards change (such as when inserting or removing items). Both of these procedures will be discussed in the following sections.

3.6.2.1 Querying the Quadtree

Figure 3.8(a) shows an example of querying the quadtree with an edge: $\texttt{QT}(e)$. The tree is traversed from the root node down in a depth-first manner. Each node checks if the edge falls within its *axis-aligned bounding box* (AABB). If it does, the node is deemed *relevant*. Figure 3.8(b) shows the relevant leaf nodes, two of which partially contain a hazard, while the topmost one (which is partially shown) is free of hazards. The quadtree is, in this example, unable to definitively determine whether or not the edge collides with any of the hazards.

Figure 3.8: Querying an edge through the quadtree: `QT`(e).



(a) (b) (c)

Note: (a) Edge e is drawn in white. (b) The three relevant leaf nodes, highlighted with a dotted border. (c) `QT`(e) returns indeterminable, the four unresolved edges E_u, highlighted with a white border.

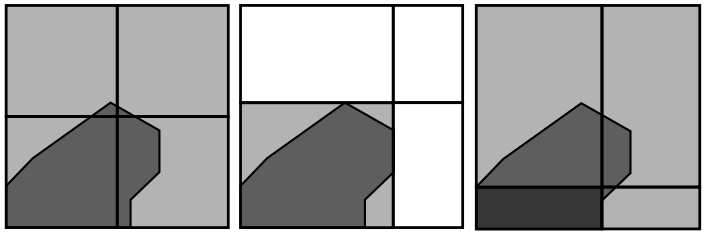
Despite this indeterminable outcome, substantial useful information is gained. Since each node stores which edges from which hazard intersect, a set of unresolved edges E_u can be derived from these relevant nodes. Figure 3.8(c) highlights the four unresolved edges, which will have to be resolved in the narrow phase (described in Section 3.6.3).

Whenever a single fully hazardous node is crossed, `QT` returns `true` (collision). If all crossed nodes are free of hazards, `QT` returns `false` (no collision). In all other cases, `QT` returns `?` (indeterminable).

3.6.2.2 Updating the Quadtree

Any change with respect to the hazards (placing or removing an item) must also be reflected in the quadtree. The quadtree must therefore be able to dynamically register and deregister hazards. Updating a hazard entails traversing the tree and updating all affected nodes. If a registration causes a leaf node ($< d$) to become partially hazardous, its four children will be generated. Likewise, if a deregistration causes a node with children to become non- or fully hazardous, its children will be pruned. In this way, the quadtree continuously adapts itself to its environment, increasing resolution where needed and reducing where it is not.

Figure 3.9: Three ways to split a node in the quadtree.



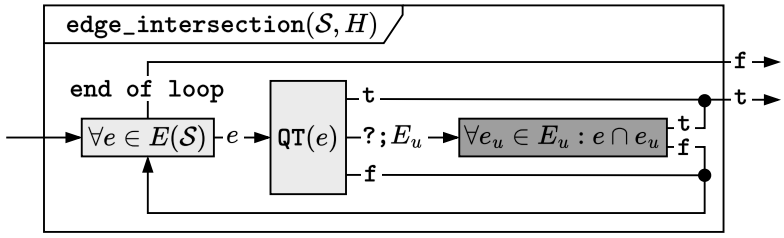
(a) (b) (c)

Note: Node contains [□ no, ▒ partial, ■ full] hazard. (a) Split in the center. (b) Split to prioritize free nodes. (c) Split to prioritize fully hazardous nodes.

When increased resolution is required in a particular node, it is split into four children. The most obvious approach would be to split the node in the center (Figure 3.9(a)). The position of the split could also be determined dynamically, focussing on either generating non-hazardous (Figure 3.9(b)) or fully hazardous nodes (Figure 3.9(c)). This would allow for more nodes to terminate at a shallower depth. However, it also makes the tree structure heavily influenced by hazards that were (de)registered earlier on. This bias could result in undesirable side effects when registering new hazards. By contrast, splitting in the center allows the quadtree to remain entirely unbiased. Since our CDE is operating in an optimization context where hazards are constantly being registered and deregistered, we opted for the center-split approach.

3.6.3 Edge Intersection

Figure 3.10: The `edge_intersection` function.



Note: [□ broad, ▒ narrow] phase, `t`: true, `f`: false, `?`: indeterminable.

The `edge_intersection` function is the first of the two sub-procedures from the `collision` procedure illustrated in Figure 3.5. This function will return `false` if shape $\mathcal{S}$ does not collide with any hazard by means of edge intersection. Figure 3.10 illustrates the procedural flow of the `edge_intersection` function.

Every edge $e \in E(\mathcal{S})$ is tested by first querying the quadtree: $\texttt{QT}(e)$. If the quadtree detects a collision (`t`), `edge_intersection` immediately returns `true`. If the quadtree concludes that no collisions occur (`f`), the function proceeds on to the next edge. If the quadtree is undecided (`?`), the function returns a set of unresolved edges E_u as described in Section 3.6.2.1. A definitive answer is then produced during the narrow phase by performing line segment intersection tests for every edge $e_u \in E_u$. Again, whenever a collision is detected, the procedure immediately returns `true`. If all edges are cleared, the procedure returns `false`.

It is worth noting that the function does not strictly detect intersections between edges, but rather collisions between edges and the quadtree's nodes. Once edge e crosses a single fully hazardous node, the procedure will immediately detect the collision, regardless of whether e is actually intersecting any edges. If sufficient resolution (depth) is available in the quadtree, collisions caused by inclusion will, in many cases, already be detected here. Therefore, the `edge_intersection` function can detect a broader range of collisions than its name implies.

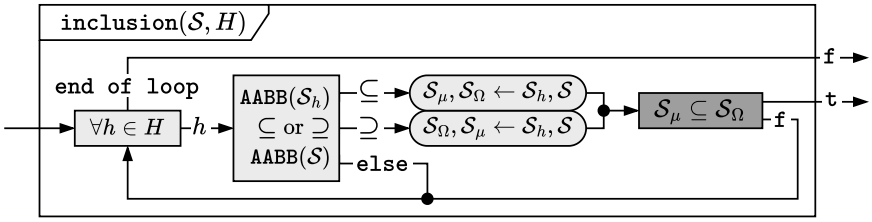
3.6.4 Inclusion

Whenever `edge_intersection` does not detect a collision, the `inclusion` procedure is called to eliminate any possibility of a collision by inclusion (Figure 3.5). Given that all collisions with partial overlap are guaranteed to be detected by an edge intersection test (Section 3.4), the only remaining undetected collisions are those where one shape is fully contained within another. Since we could be dealing with hundreds of distinct hazards, the naive implementation (Section 3.4) can become time-consuming. A more sophisticated method is required.

Figure 3.11 illustrates the flow of the `inclusion` procedure. Once again, performance is greatly improved by leveraging a two-phased approach. The broad phase of the `inclusion` procedure is largely centered around the observation that, in order for one polygon to be fully contained in another, its respective axis-aligned bounding box (AABB) must also be fully contained in the AABB of the other:

$$\mathcal{S}_\mu \subseteq \mathcal{S}_\Omega \implies \texttt{aabb}(\mathcal{S}_\mu) \subseteq \texttt{aabb}(\mathcal{S}_\Omega) \tag{3.10}$$

Figure 3.11: The `inclusion` function.



Note: [□ broad, ■ narrow] phase, `t`: true, `f`: false.

The procedure loops over all hazards and checks if any of their AABBs is fully contained within the AABB of $\mathcal{S}$ or vice versa. If one of the AABBs is fully contained in the other, an inclusion could occur. In this case, it is also known which shape assumes the role of $\mathcal{S}_\mu$ and $\mathcal{S}_\Omega$. This enables us to proceed to the narrow phase, where a single point-in-polygon test from $\mathcal{S}_\mu$ with respect to $\mathcal{S}_\Omega$ is performed.

In theory, any point $p \in \mathcal{S}_\mu$ could be used for the point-in-polygon test. However, in order to avoid any potential numerical issues, we should also ensure that p is not close to the boundary of $\mathcal{S}_\mu$ or $\mathcal{S}_\Omega$. The pole of inaccessibility (PoI) of a polygon, explained in the following section, satisfies both of these requirements and is therefore an ideal candidate.

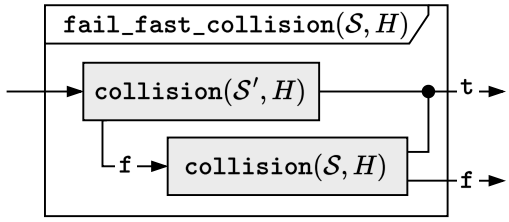
3.7 Fail-fast Surrogate

Recall that the CDE is targeted towards irregular C&P problems, where the goal is to pack the items in a compact fashion (as per Figure 3.1). It is therefore safe to assume that hazards will also be located in close proximity. When validating item placements, the probability of colliding and non-colliding outcomes will not be balanced, with the former greatly outnumbering the latter.

The *fail-fast* principle was first applied by Haralick and Elliott [47] in tree search algorithms. They beautifully captured the essence of this concept when they stated: *"to succeed, try first where you are most likely to fail"*. This section will explore how we can leverage this principle to resolve *obviously* colliding queries with as little effort as possible, thereby enhancing the overall efficiency of the CDE.

Figure 3.12 provides an abstract overview of the fail-fast approach applied to collision detection. Prior to using the actual shape $\mathcal{S}$, collisions are first detected

Figure 3.12: The `fail_fast_collision` procedure.



Note: A collision check on a surrogate shape $\mathcal{S}'$ precedes a check on the actual shape $\mathcal{S}$.

using a surrogate representation $\mathcal{S}'$. When the surrogate is a subset of the original shape ($\mathcal{S}' \subseteq \mathcal{S}$), a colliding surrogate implies a collision of the original:

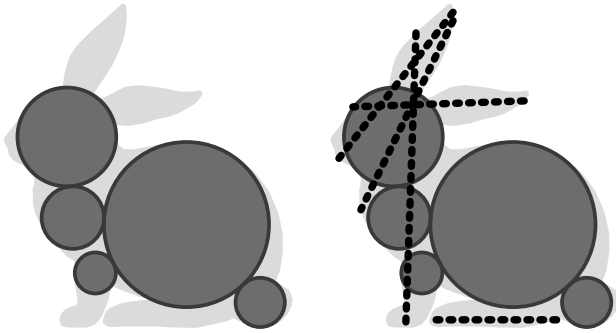
$$\mathcal{S}' \cap \mathcal{S}_H \neq \emptyset \implies \mathcal{S} \cap \mathcal{S}_H \neq \emptyset \tag{3.11}$$

A complete check using the actual shape is therefore only necessary when the algorithm detects no collision with the surrogate. The surrogate model should be designed in a way to maximize the probability of detecting a collision with as little computational effort as possible. For non-colliding queries, this additional check of the surrogate will slightly impair performance. Nevertheless, given the aforementioned imbalance concerning the outcome of collision queries, this trade-off is usually worthwhile.

Converting polygons into a raster of pixels [83], circle covering [81] and semi-discrete representations [67] are three other examples of surrogate representations aimed to address the geometric challenge. The difference between these approaches and our intended use is that they employ the surrogate as a total replacement for the original shape. To ensure feasibility, these *replacing* surrogates must cover all collision cases of the actual shape. Consequently, the shape replacing a hazard must be a superset of the original hazard (${\mathcal{S}'}_h \supseteq \mathcal{S}_h$). This results in an inflation of the items and a deflation of the containers, which in turn causes a loss in precision (and maximum solution quality). By instead using the surrogate merely as a preceding check (fail-fast), this significant drawback does not apply.

The following sections detail our implementation of a fail-fast surrogate, consisting of two distinct components.

Figure 3.13: Example of a fail-fast surrogate.



(a) Poles. (b) Poles & piers.

3.7.1 Poles (of Inaccessibility)

The first component of the surrogate model involves a set of fully contained circles. Figure 3.13(a) shows an example where five such circles are inscribed within a bunny-shaped polygon $\mathcal{S}$. A collision between any of these circles and a hazard also implies a collision with the original $\mathcal{S}$. Detecting collisions using circles has three distinct advantages over line segments: they are (i) cheaper to transform, (ii) cheaper to query the quadtree with and (iii) more likely to collide. Figure 3.13(a) makes clear how just a few of these circles already cover the bunny relatively well.

We employ a concept that originates from geography to generate these circles. The Pole of Inaccessibility (PoI) defines the point in the interior of a shape that is the furthest away from its boundary. It is important to note that the PoI is not synonymous with the centroid of a polygon. A key distinction lies in the fact that, for non-convex polygons, the centroid is not guaranteed to be located in the interior of the shape. From this point on, whenever we refer to the pole, we are referring to a circle with the PoI as its origin and the distance from the PoI to the boundary as its radius.

Agafonkin [2] proposed an efficient algorithm to calculate a polygon's PoI up to a certain (configurable) precision. For use in the proposed surrogate, we extended Agafonkin's algorithm so that it can iteratively generate multiple non-overlapping poles. Each iteration generates the next pole, with all poles from previous iterations being treated as holes in the shape. Figure 3.13(a) shows the first five poles the algorithm generates for the bunny.

3.7.2 Piers

The main body of the bunny in Figure 3.13(a) is captured quite well by the five poles, but its ears and legs are not represented at all. Long and narrow sections are particularly challenging to capture with a limited set of circles. This is not ideal since these 'extremities' are quite likely to collide with hazards. To combat this, we include a second component of surrogate representation to complement the poles.

Figure 3.13(b) shows a set of (dashed) lines crossing the bunny, which we will refer to as *piers*. A pier is a line segment which is fully contained within the original shape, and selected in such a way to maximally complement the aforementioned limitations of the poles. Figure 3.13(b) shows that, with addition of a few piers, the shape of the bunny is now almost entirely captured. The surrogate with both poles and piers enables us to detect virtually all 'obvious' collisions, while being much cheaper to query than the actual shape.

3.8 Polygon Simplification

Despite the numerous efficiency enhancements introduced in the previous sections, both the query- and update-speed of the CDE remain affected by the number of edges in the polygons. This sensitivity is not unique to 2D irregular C&P problems and is often addressed by applying polygon simplification: the act of reducing the number of edges in a polygon while remaining as faithful to the original shape as possible.

For use in the CDE, simplified polygons are effectively a replacing surrogate (like those described in Section 3.7) and must therefore cover all collision cases of the original shape. In order to guarantee feasibility, original hazards have to be fully captured by their simplified versions:

$$\mathcal{S}_h \subseteq \mathcal{S}'_h \tag{3.12}$$

This means that entities are only allowed to be inflated or deflated depending on whether they induce an interior or exterior hazard, respectively:

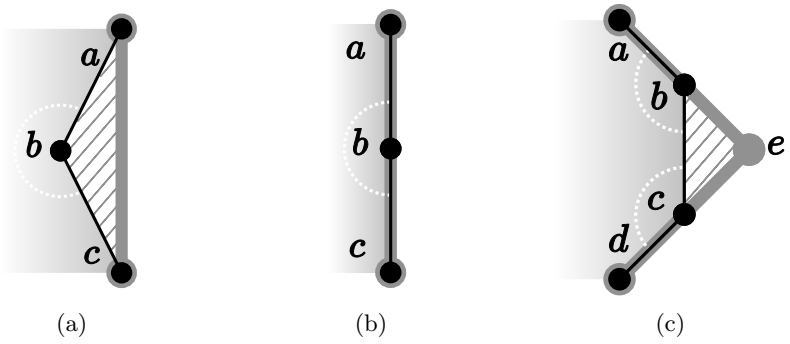
$$\mathcal{S}_e \begin{cases} \subseteq \mathcal{S'}_e & \text{if entity } e \text{ induces an interior hazard} \\ \supseteq \mathcal{S'}_e & \text{if entity } e \text{ induces an exterior hazard} \end{cases} \tag{3.13}$$

The procedure has to ensure that, when a simplification induces a change on the shape, it does so **strictly** outwards or inwards with respect to the original.

This requirement is generally not of concern for other applications of polygon simplification, which is why we cannot simply use any existing implementation such as Douglas and Peucker [26], Fisher et al. [33], and Visvalingam and Whyatt [91] or Wang and Müller [94]. Additionally, inflating hazards effectively restricts the search space and thus has a negative impact on the achievable solution quality. Therefore, we do not wish to simplify the polygons in any way that would significantly alter them. We are, in effect, seeking only to eliminate *insignificant* features.

For purely academic optimization problems, such as the 2D nesting instances introduced by Oliveira et al. [74], a simplification procedure may not be particularly effective because of the relatively simple shapes which have most likely been plotted manually. However, real-world problems, such as the leather nesting dataset introduced by Baldacci et al. [7], sometimes contain more detailed shapes generated by scans and/or arc approximations. The simplification potential for these shapes is significant with almost no associated change in area.

Figure 3.14: Three options for simplification.



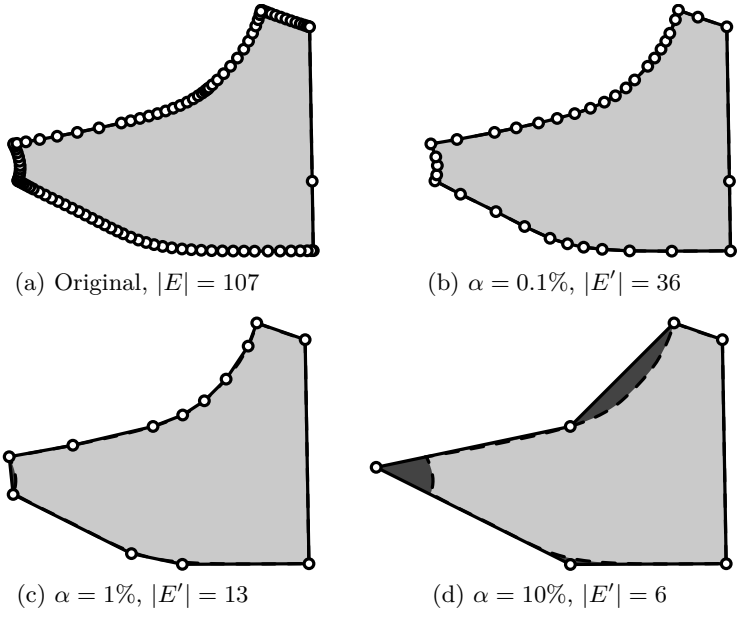
(a) (b) (c)

Note: Simplification by inflation illustrated. The angles are viewed from the interior of the polygon (shaded area). (a) Concave angle $\{a, b, c\} \to \{a, c\}$. (b) Collinear angle $\{a, b, c\} \to \{a, c\}$. (c) Consecutive convex angles $\{a, b, c, d\} \to \{a, e, d\}$.

Our approach consists of three elementary operations shown in the example of Figure 3.14, in which a polygon is being simplified by inflation ($\mathcal{S}' \supseteq \mathcal{S}$). The shaded area represents the interior of the polygon, while the hatched area denotes what is gained through the various simplification procedures. The original vertices and edges are drawn in black, with the replacement ones in gray. Each of these simplifications reduces the total number of edges by one. The same holds for simplification by deflation if one simply reverses the interior and exterior of the polygon.

The simplification procedure is performed in an iterative and greedy way. Each iteration generates all valid simplifications and calculates their associated differences in area. A proposed simplification is rendered invalid if the resulting polygon would intersect itself. The simplification with the smallest area change is selected and applied, each time reducing the number of edges by one. Evidently, simplifications of a collinear angle are always eliminated first since they do not cause any inflation or deflation. This process continues until no further simplification is possible without exceeding a configurable inflation/deflation threshold (α). This threshold is expressed as a percentage of the area of the original polygon, and is inversely correlated with the attainable compactness of the solution.

Figure 3.15: Example of polygon simplification for different values of α.



(a) Original, $|E| = 107$ (b) $\alpha = 0.1\%$, $|E'| = 36$

(c) $\alpha = 1\%$, $|E'| = 13$ (d) $\alpha = 10\%$, $|E'| = 6$

Note: [□ original, ■ simplified] shape, $\mathcal{S}' \supseteq \mathcal{S}$, $|E'|$ denotes the remaining number of edges.

Figure 3.15 illustrates our approach, simplifying a shape from the leather nesting dataset introduced by Baldacci et al. [7]. In Figure 3.15(b) and 3.15(c), the number of edges reduces by 66% and 88% respectively, with the simplified shape very closely resembling the original from Figure 3.15(a). For more extreme simplifications, such as in Figure 3.15(d) where 94% of the edges are eliminated, the proposed approach is no longer very effective. However, in an optimization setting, we are only interested in eliminating largely redundant features, and raising α to 1% would already be too significant to be practical in many cases.

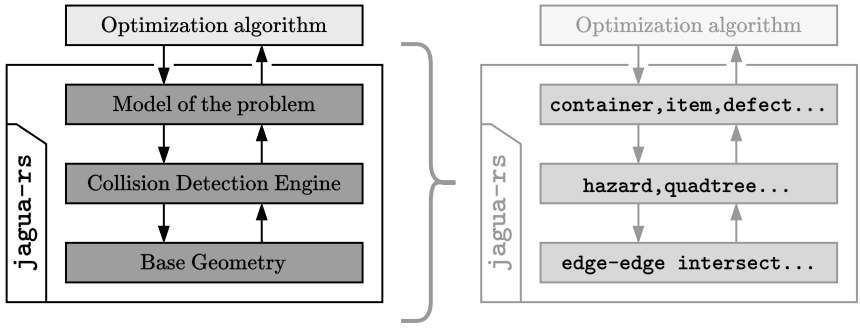
3.9 Codebase

The CDE implementation is called `jagua-rs` and its source code is available at: https://github.com/JeroenGar/jagua-rs. We wrote this project entirely in Rust: a statically-typed and compiled programming language with a strong emphasis on performance, safety and robustness. We have currently implemented both strip- and bin-packing problems, however the project can be extended to support virtually any 2D irregular C&P problem variant.

Given that the CDE operates in a continuous space, it is important to be aware of the inherent limitations of floating-point arithmetic. Throughout our implementation, we have taken great care to ensure that false negatives — undetected collisions — do not occur. For the polygon inclusion test from Section 3.6.4 for example, we eliminate numerical issues that could arise when using a point that is close to the boundary of the shape by always opting to use the PoI. When unable to eliminate, the design philosophy of `jagua-rs` is to always err on the side of caution. Feasibility is guaranteed above all else, even if this means producing false positives in edge cases. An important consequence of this philosophy is that for two shapes to be deemed non-colliding they must be strictly separated, even if only by an infinitesimal margin. This limitation means that 'exact fits' will always be deemed infeasible. This is a deliberate choice and we consider it an acceptable trade-off to ensure the robustness and reliability of the CDE.

The repository also contains a basic optimization algorithm built on top of `jagua-rs`, namely a left-bottom-fill heuristic. This simple constructive heuristic selects the leftmost and bottommost position possible for each item. It is far from the state of the art and simply serves as a reference implementation to demonstrate how researchers can couple their own optimization algorithms with `jagua-rs`.

Figure 3.16: High-level overview of the different layers involved in solving a 2D irregular C&P problem and the scope of `jagua-rs`.



Note: The figure on the right lists some components discussed in previous sections, positioned at their corresponding layer.

Figure 3.16 provides a high-level visualization of the different layers involved in solving a 2D irregular C&P problem and delimits the scope of `jagua-rs`. The optimization algorithm — which decides where to place items and which containers to use — exists outside `jagua-rs`. To give an impression of how someone might employ `jagua-rs` and where any modifications are required, we will briefly discuss three possible workflows.

First, imagine a researcher is developing a new optimization algorithm for the academic strip-packing problem. This type of problem is already modeled in `jagua-rs` and therefore it can be used as a library (as-is), allowing the researcher to focus solely on their optimization algorithm without delving into any of the lower layers.

Second, consider a new real-world problem which features some additional constraints such as the quality zones explained at the end of Section 3.5. In this case, not only does an optimization algorithm need to be developed, but `jagua-rs` also needs to be extended to model the additional spatial constraints (quality zones in this example). Thanks to the fact that the CDE operates on hazards, which are general enough to express quality zones, these modifications are restricted to the top two layers and do not leak into the bottom two.

Third, there are those interested in improving the CDE itself and are not necessarily focused on an optimization challenge. The decoupling illustrated in Figure 3.16 means their contributions to the bottom layers can be readily exploited by the previous two workflows without significant effort or interaction.

3.10 Validation & Evaluation

While alternative approaches for collision detection in 2D irregular C&P do exist (Section 3.2), no decoupled engine is currently available. Even if one were, a direct and meaningful comparison would only be possible if the fundamental interactions (Section 3.3) and limitations (Section 3.2) are similar.

In our opinion, the only irrefutable way to validate (i) the use of trigonometry, (ii) the general design of the CDE and (iii) its implementation, is to build a competitive optimization algorithm on top of `jagua-rs`. Readers interested in seeing `jagua-rs` in action are referred to the `sparrow` project: an open-source heuristic for the 2D strip packing problem [41]. Built on top of `jagua-rs`, `sparrow` demonstrates the CDE's capabilities in practice.

For readers interested in a relative performance evaluation, we use the `criterion.rs` framework by [48] to perform "statistics-driven microbenchmarking" of the most critical operations within the CDE. We encourage readers interested in using the CDE to re-run the tests on their own hardware and problem instances using the latest version of `jagua-rs`. These benchmarks should provide valuable insights into the factors at play and the trade-offs involved when tuning the CDE for maximum performance.

3.11 Conclusion

There are two challenges involved in any irregular cutting and packing problem. First, there is the geometric difficulty of ensuring items are placed in such a way that they do not collide with each other, with the boundary of the container or certain restricted regions within. Second, there is the optimization challenge of finding a good solution in accordance with some objective. Regardless of the method used to reach a high-quality solution, being able to quickly and accurately determine its geometric feasibility is of universal importance. Until now, there has not been a general-purpose and reusable way of handling this shared geometric challenge.

The collision detection engine (CDE) we have introduced in this chapter addresses this gap by decoupling geometry from optimization. The CDE provides researchers and developers with a powerful and adaptable tool that can quickly determine whether or not a certain placement of items is feasible. This enables them to confidently focus their efforts on the optimization challenge at hand, without worrying about the underlying geometry.

This chapter introduced a suite of core principles and design philosophies which we believe are essential when modeling a CDE for use in irregular C&P problems: *hazards, two-phased approach, fail-fast surrogate* and *polygon simplification*. However, such concepts are only useful insofar as they can actually be leveraged to achieve an effective implementation.

Therefore, we also shared one possible interpretation of these concepts which, when combined, resulted in a concrete implementation. `jagua-rs` targets maximum performance and robustness, being able to resolve millions of queries per second. The source code for this project is available at: https://github.com/JeroenGar/jagua-rs

We intend for this chapter and `jagua-rs` to serve as catalysts for further research and development in the field of irregular C&P problems, providing solid foundations on which to build. While readers could certainly directly reuse or adapt `jagua-rs`, we would also encourage further experimentation and interpretation of these building blocks to arrive at something that works best for their particular context.

In the following chapter, we demonstrate the capabilities of `jagua-rs` in practice by building a state-of-the-art optimization algorithm on top of it.

Chapter 4

An Open-Source Heuristic to Reboot 2D Nesting Research

This chapter is a minimally edited version of the following submitted work:

J. Gardeyn, G. Vanden Berghe, and T. Wauters. "An open-source heuristic to reboot 2D nesting research". In: *arXiv [preprint]* (2025). DOI: 10.48550/arXiv.2509.13329

This paper is under review at *European Journal of Operational Research* at the time of writing this thesis.

This chapter is presented together with the following open-source software:

J. Gardeyn. "`sparrow`: state-of-the-art nesting for 2D irregular strip packing". 2025. URL: https://github.com/JeroenGar/sparrow

4.1 Introduction

2D irregular cutting and packing (C&P) – or *nesting* – problems involve fitting a set of smaller items into larger containers, where the shape of the items and/or containers can be irregular (non-rectangular). Such optimization problems arise in a wide range of industries such as garment manufacturing, furniture production, shipbuilding, printing, laser-cutting, metalworking, woodworking, foam-cutting, leather crafting, and microchip design. Designing effective algorithms to solve these NP-hard problems [36] has been an active area of research since the 1960s [5].

Given the real-world relevance of 2D irregular C&P problems and the ongoing shift in ethos toward open-source practices [88], one would expect to find a vibrant ecosystem of publicly available implementations that respond to advances in the academic state of the art (SotA). However, despite more than a decade of stagnation – where there has been little to no improvement in achievable solution quality – the open-source landscape remains disconnected from academic progress and trails significantly behind in performance. We hypothesize that there are two interconnected factors primarily responsible for this situation.

The first is a **high barrier to entry**. Beyond the fundamental *optimization* challenge of finding good solutions within a vast search space, irregular C&P problems also pose a significant *geometric* challenge of verifying the feasibility of such solutions in the first place. Accurately and efficiently determining whether a given placement of items does not involve any collisions is far from trivial. These compounding challenges are currently holding back academic advances, as all implementations must start from scratch.

The second factor, we believe, that is leading to this stagnation is a **lack of reproducibility**. At the time of writing, not a single academically competitive algorithm for 2D irregular C&P problems has publicly available source code. Moreover, many methods fail to provide enough explanation to facilitate complete reproducibility.

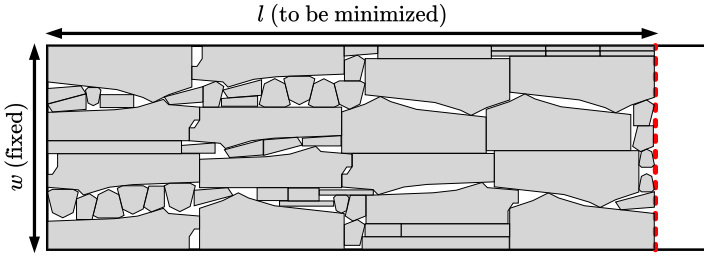


Figure 4.1: The 2D irregular strip packing problem.

This chapter has two interrelated goals: (i) bridge the gap between academic and open-source algorithms and (ii) lower the barrier to entry for future researchers. When addressing these goals, we will rely on the recently introduced *Collision Detection Engine* (CDE) for 2D irregular C&P problems by Gardeyn et al. [42], which can handle the geometric challenge. Our entire focus can therefore target the optimization challenge. On top of the CDE, we will design an open-source heuristic for the 2D irregular strip packing problem, arguably the most canonical 2D nesting problem. While we focus on strip packing, our broader ambition is to develop an algorithm that can generalize to other variants of 2D irregular C&P problems without much effort.

4.2 Problem Definition

The 2D irregular strip packing problem (2DISPP) involves placing a set of irregular items into a rectangular strip of fixed width w and variable length l. The objective is to minimize the length of the strip, while ensuring all items are placed entirely within it and none of them collide with each other. Figure 4.1 illustrates the problem through an example solution.

Let us begin by introducing some notation:

$i \in I$	: an item i from the set of items to be packed I
S	: the interior of a shape, represented by a set of points $(x, y) \in \mathbb{R}^2$
S_i	: item i's shape
t	: a rigid transformation: a translation $\in \mathbb{R}^2$ + rotation $\in [0, 2\pi[$ + reflection $\in \{0, 1\}$
$t(S)$	: the result of transforming shape S by t
$S_a \cap S_b \neq \emptyset$	: a collision between two shapes

We can now formulate the 2DISPP more formally:

$$\begin{aligned}
\min \quad & l \\
\text{s.t.} \quad & t_i(S_i) \subseteq [0, w] \times [0, l] && \forall i \in I && (4.1)\\
& t_a(S_a) \cap t_b(S_b) = \emptyset && \forall a, b \in I, a \neq b && (4.2)\\
& t_i \in \mathbb{R}^2 \times [0, 2\pi[\times \{0, 1\} && \forall i \in I && (4.3)\\
& l > 0 && && (4.4)
\end{aligned}$$

The objective of the 2DISPP is to determine a set of rigid transformations $\{t_i : i \in I\}$ such that, when each item i's shape is transformed by its corresponding transformation t_i, all items are positioned entirely within the strip (Equation 4.1) and none collide with each other (Equation 4.2) for the smallest possible strip length l.

In practice, the 2DISPP is more restricted than our definition. Item shapes are usually represented by the interior of simple polygons, with transformations restricted to continuous translations plus a set of discrete rotations (such as multiples of 90 or 180 degrees). Continuous rotation and reflection (flipping) are seldom considered in academic publications, despite them being relevant for some real-world applications. In the interest of generality, we aim for our algorithm to be compatible with any rigid transformation.

4.3 State of 2D Irregular Strip Packing

Solution methods for the 2DISPP fall into three main algorithmic categories. We will briefly discuss each category, its notable contributions and some key takeaways.

Exact Algorithms

Exact methods for the 2DISPP typically rely on mathematical programming formulations. Leao et al. [60] provide a comprehensive survey of the broad range of exact algorithms for irregular C&P problems. Such approaches can generally only handle a small set of items, support few angles of rotation and struggle with complex shapes. Although more recent work – such as the mixed-integer formulation by Lastra-Díaz and Ortuño [58] – demonstrates continued advances, exact approaches remain impractical for many academic instances, let alone complex real-world instances.

Construction-based Heuristics

Construction heuristics build solutions sequentially by placing one item at a time according to a predetermined ordering in addition to a placement rule. To improve solution quality, many methods embed a higher-level heuristic that explores different orderings and placement strategies. Representative work has been conducted by Amaro Junior et al. [3], Bennell and Song [9], Burke et al. [16], Gomes and Oliveira [46], and Oliveira et al. [74]. These algorithms are generally straightforward to implement and reproducible provided that sufficient detail is included in the corresponding publication.

However, the achievable solution quality of construction heuristics lags significantly behind the SotA. We can attribute this to the nature of the 2DISPP, which does not lend itself well to such approaches for two main reasons. First, even minor changes to the ordering of items or their placement early in the construction process leads to unpredictable changes concerning final solution quality. This chaotic nature undermines the effectiveness of the higher-level heuristics.

Second, if one jumps ahead to the densely packed solutions in Figure 4.11, try to imagine the orderings and placement rules needed to position the items one by one into similar configurations. Maybe one could devise an effective placement strategy that works well for one of these instances, but it is hard to imagine a single set of rules that would work well for all.

Iterative-improvement Heuristics

Most methods relying on iterative improvement begin with a complete solution and repeatedly apply local modifications – such as shifting or swapping items – in an attempt to find new solutions of higher quality. Early contributions include Egeblad et al. [28], Leung et al. [61], and Umetani et al. [87]. More recent advances by Elkeran [29], Sato et al. [82], and Wang et al. [93] represent the SotA for the 2DISPP. The common theme across these methods is that they tolerate temporary collisions between items. The strip gets converted into a container with fixed dimensions and, every time a feasible solution is found, its length is reduced further. This shrinking leads to new collisions, which the aforementioned methods attempt to gradually resolve in order to reach a feasible solution for the shorter strip.

While many of these iterative-improvements heuristics can achieve high-quality solutions, they tend to be more complicated than construction-based heuristics. Even when their algorithmic flow is presented clearly – as in Sato et al. [82] – the absence of source code or sufficient implementation details prevents true reproducibility.

Open Source

Solid, robust and publicly available implementations to solve the 2DISPP are scarce. A notable exception is a tree-search heuristic by Fontan [35]. This heuristic has a unique approach to handling the geometric challenge that involves decomposing shapes into trapezoids. Despite representing a valuable open-source contribution, Fontan [35]'s method is unrelated to the iterative-improvement methods and cannot consistently compete with the academic SotA.

4.4 A Sequence of Feasibility Problems

As indicated in Section 4.3, the best-performing algorithms for the 2DISPP temporarily relax the non-collision constraint (Equation 4.2) during their iterative search. These methods essentially convert the optimization problem into a **sequence of feasibility problems**. Each feasibility problem fixes strip length l and the goal becomes to find a single solution that is feasible.

Umetani et al. [87] referred to this subproblem as the 'Overlap Minimization Problem', but we find this term somewhat misleading. Feasibility, not minimal overlap, is the true goal. Any solution with overlap is ultimately invalid, regardless of how small it may be. Additionally, reaching a feasible solution does not entail strictly minimizing overlap, there may be additional factors that aid the search in its goal to eliminate overlap. Therefore, a more appropriate name for this subproblem would probably be the 'Overlap Elimination Problem'.

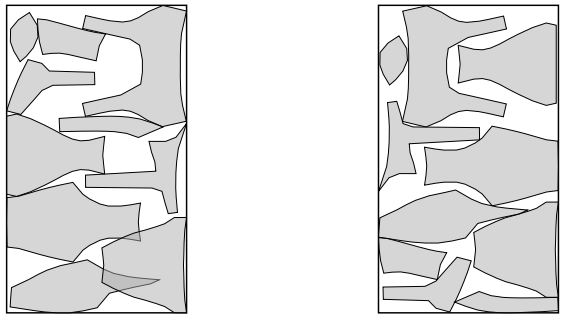
The idea of exploring infeasible regions has existed for a long time [45] and is a strategy commonly used in the irregular C&P literature. However, the reasons for its effectiveness in nesting problems are rarely elaborated upon. We will therefore take this opportunity to offer some intuition for why this conversion works so well.

The 2DISPP is a highly constrained problem, with the non-collision constraint (Equation 4.2) being the most significant. This constraint is so restrictive that virtually any local modification, such as shifting or swapping the positions of items, would immediately render a solution infeasible. The high-quality solutions we seek are effectively rare oases in a vast desert of infeasibility. Algorithms that cannot venture outside the feasible region are prevented from making meaningful progress. The best-performing algorithms therefore temporarily relax this non-collision constraint, enabling them to traverse otherwise inaccessible routes towards high-quality solutions.

In order to effectively solve the feasibility problem, an algorithm must be able to move from an infeasible solution, such as the one depicted in Figure 4.2(a), towards a feasible solution like the one in Figure 4.2(b). We identify three key components required to achieve this:

- **Assessing feasibility**: A binary collision check to determine which constraints are violated.
- **Search algorithm**: A strategy that explores the solution space in order to incrementally resolve collisions.

Figure 4.2: Example of an infeasible and feasible configuration of a set of items within containers of equal dimensions.



(a) Two colliding pairs of items (b) Feasible

- **Quantifying infeasibility**: As collisions are often not resolvable in a single step, a binary check is too coarse to effectively guide the search. A smooth, continuous metric is required to evaluate the severity of each collision and direct the search algorithm.

Sections 4.5-4.8 introduce these components one by one and eventually combine them into a complete algorithm to solve the feasibility problem. Finally, Section 4.9 will integrate everything into a heuristic to solve the 2DISPP.

4.5 Detecting Collisions

The most elementary challenge involved in solving any feasibility problem is assessing whether or not a constraint is violated. Given the geometric challenge that accompanies nesting problems, checking the feasibility of a solution or even of a single placement in a way that is efficient, precise and robust is not trivial. The fundamental ways of dealing with this geometric challenge have been described by Bennell and Oliveira [10]. Currently, all state-of-the-art nesting algorithms rely on either *no-fit polygons* (NFPs) or a *raster* to handle the geometric challenge. However, these two approaches have serious limitations regarding robustness and precision, respectively.

A third approach mentioned by Bennell and Oliveira [10] is trigonometry: a simple, precise, robust and general way of detecting collisions between two polygons. This approach was historically very sensitive to the total number of edges in the shapes involved, and therefore seldom used in practice.

Recently, Gardeyn et al. [42] introduced a *Collision Detection Engine* (CDE) to provide a fast and reliable way for 2D nesting algorithms to assess feasibility. This CDE builds on trigonometric principles and includes a comprehensive suite of algorithmic enhancements to overcome the traditional limitations of such trigonometric approaches. The CDE is implemented in an open-source library called `jagua-rs`[1]. `jagua-rs` supports irregular shapes for both items and containers, continuous rotation and translation, and easily extends to problem variants containing additional spatial constraints.

The optimization algorithm we will introduce in this chapter will be built on top of `jagua-rs`: assessing feasibility through collision **queries** and issuing **updates** to reflect any changes in the solution. All interactions with `jagua-rs` throughout this chapter can be abstracted into the following two functions:

`jagua-rs::collisions`$(S) \rightarrow C$
Returns the set of items C ($\subseteq I$) that collide with shape S.

`jagua-rs::move_item`(i, t)
Moves item i to a new position defined by transformation t.

4.6 Quantifying Collisions

To facilitate an effective search algorithm, the binary feasibility check provided by the CDE (Section 4.5) needs to be converted into a continuous one capturing the *severity of the collision*: an expression of the expected difficulty to resolve it.

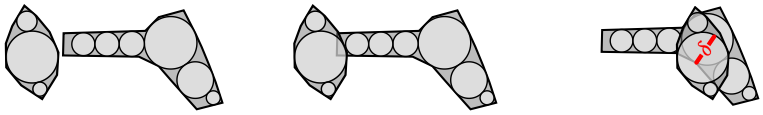
Algorithm 4 `evaluate_item_pair`(a, b)

```
1: if a ∈ jagua-rs::collisions(S_b) then
2:     return quantify_collision(S_a, S_b)        ▷ Alg. 7
3: else
4:     return 0
```

Let us begin by introducing Algorithm 4, the general function to evaluate any pair of items $a, b \in I$. The procedure begins by querying the CDE to determine whether the two items are colliding with each other. When no collision occurs, the function returns 0. When a collision is detected, the shapes of the items involved will be passed to the `quantify_collision` function.

[1] https://github.com/JeroenGar/jagua-rs

Figure 4.3: Two shapes in different configurations, with their respective poles drawn.



(a) Separated (b) Collision (minor) (c) Collision (major)

Note: In (c), the penetration depth (δ) between the largest pair of poles is marked in red.

Many existing approaches for the 2DISPP quantify the severity of a collision in some way. Imamichi et al. [52] proposed using *penetration depth*, which denotes the smallest translation needed to separate two colliding entities. Instead of the exact penetration depth, they measure the minimum translation required along a single axis, such as horizontally or vertically. Leung et al. [61], Elkeran [29], Wang et al. [93] and Sato et al. [82] each employ some variant of penetration depth to quantify the severity of a collision. Regardless of the concrete implementation, these penetration-depth derivatives are essentially all one-dimensional metrics.

When relying on a trigonometric approach to detect collisions, such as `jagua-rs`, computing penetration depth is impractical. We therefore need to find a different way to efficiently quantify the severity of a collision. It is important to stress that this function will only serve to guide the search towards a feasible solution as effectively as possible. It does not need to represent anything tangible, nor does it need to be precise. We aim for a metric that supports continuous rotation and is insensitive to the complexity of the shapes involved, for the sake of greater generality and real-world applicability.

The remainder of this section describes how we design our quantification function, eventually leading to the complete `quantify_collision` function that is called within Algorithm 4.

4.6.1 Area of Overlap

Figure 4.3 shows two shapes in three different placements, once separated and twice colliding. Intuitively, the collision in Figure 4.3(c) appears harder to resolve than the one in Figure 4.3(b). The intersection area – how much the shapes overlap – is an obvious candidate for measuring collision severity. However, calculating this metric precisely is far too expensive to be practical in an iterative improvement heuristic. Let us investigate whether we can use a shape's *pole of inaccessibility* to devise a cheap proxy.

A shape's pole of inaccessibility is the point within its interior that lies furthest from the boundary. This point, along with its distance to the boundary, defines the largest inscribable circle in the shape. Figure 4.3 shows a sequence of such inscribed circles – referred to simply as *poles* from now on – for each shape. Gardeyn et al. [42] describe the precise procedure to compute these sets of poles, extending the approach by Agafonkin [2]. The CDE leverages them to accelerate the collision detection process and lower its sensitivity to the complexity of the shapes. The set of poles of shape S is represented by $P(S)$. Poles are generated once (in preprocessing) for every base shape and can be efficiently transformed in unison with any rigid transformation applied to the shape.

Algorithm 5 `overlap_proxy`(S_a, S_b)

```
for (p_a, p_b) ∈ P(S_a) × P(S_b) do                ▷ all pairs of poles
    δ ← penetration depth(p_a, p_b)
    if δ > 0 then
        α ← α + δ · min{⌀(p_a), ⌀(p_b)}            ▷ ⌀: diameter
return α
```

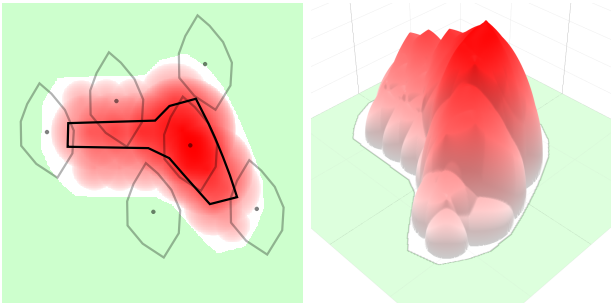
Algorithm 5 introduces a proxy for the overlap between two shapes that leverages these poles. For every combination of poles from the two shapes, the penetration depth (δ) is calculated as the sum of their radii minus the distance between their centers. All these penetration depths are then weighted by the diameter of the smaller pole and combined into a weighted sum α, which provides a suitable proxy for the area of overlap.

Figure 4.4(a) visualizes the output of this function for the pair of items from Figure 4.3, with the smaller item being positioned all around the larger one. The shapes and reference points of six example placements are visualized. The region where the two items do not collide is colored green. The region where they collide is shaded red, with the intensity of the red mapping to the output of the function described in Algorithm 5. To help visualize the intensity further, Figure 4.4(b) maps the value of this function to the height of a 3D surface.

In theory, this proxy function has a quadratic complexity $O(n^2)$ regarding the number of poles. In practice however, the computation time is manageable because (i) a relatively small number of poles (8-16) suffices to capture most shapes and (ii) the loop can be vectorized in modern CPUs either automatically by the compiler or by explicitly leveraging SIMD instructions.

This overlap proxy exhibits many desirable properties of a quantification function for the severity of a collision: fast computation, a smooth gradient, and a value that consistently increases with the amount of overlap. However, poles do not cover the entire item and therefore collisions may occur without any pairs of poles overlapping. For the collision depicted in Figure 4.3(b) or throughout the entire white region in Figure 4.4, Algorithm 5 will return 0 even though the CDE rightfully detects a collision. This is problematic because these (minor) collisions are not contributing anything to the quantification function and consequently provide no information to help guide the search.

Figure 4.4: Visualization of Algorithm 5



(a) 2D (b) 3D

Note: The same pair of items as in Figure 4.3 is shown, : feasible, : `overlap_proxy`.

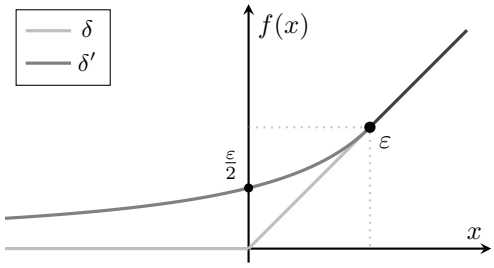
4.6.2 Decaying Penetration Depth

To address the aforementioned issue, we introduce a *decaying* version of the penetration depth between pairs of poles. When δ is smaller than a certain threshold ε, we switch to a hyperbolic decay function that asymptotically approaches 0 as the distance between the pair of poles increases. The decaying penetration depth δ' is defined as follows:

$$\delta' = \begin{cases} \delta & \text{if } \delta \geq \varepsilon \\ \frac{\varepsilon^2}{-\delta+2\varepsilon} & \text{otherwise} \end{cases} \tag{4.5}$$

A comparison of the decaying variant and standard penetration depth function profiles is visualized in Figure 4.5.

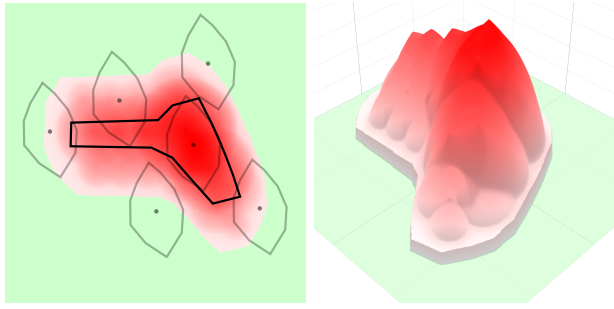
Figure 4.5: Comparing the standard (δ) and decaying (δ') penetration depth functions.



Algorithm 6 `overlap_proxy_decay`(S_a, S_b)

$\varepsilon \leftarrow \mathtt{R}_\varepsilon \cdot \max\{\varnothing(S_a), \varnothing(S_b)\}$ ▷ $\varnothing$: *diameter*
for $(p_a, p_b) \in P(S_a) \times P(S_b)$ **do**
  $\delta \leftarrow$ *penetration depth*(p_a, p_b)
  if $\delta > \varepsilon$ **then**
    $\delta' \leftarrow \delta$
  else
    $\delta' \leftarrow \varepsilon^2/(-\delta + 2\varepsilon)$
  $\alpha \leftarrow \alpha + \delta' \cdot \min\{\varnothing(p_a), \varnothing(p_b)\}$
return α

Figure 4.6: Visualization of Algorithm 6



(a) 2D (b) 3D

Note: The same pair of items as in Figure 4.3 is shown, : feasible, : `overlap_proxy_decay`.

Algorithm 6 describes the modified overlap proxy, which employs the decaying penetration depth. Threshold ε is computed as the larger diameter of the shapes – the furthest distance between any two of its points – multiplied by a small constant $\texttt{R}_\varepsilon$.

Figure 4.6 visualizes the new function in the same way as Figure 4.4. Every collision now contributes to the quantification function and there is a smooth gradient present everywhere. Additionally, a sharp drop-off is now present at the feasibility boundary, resulting in a significant distinction between a colliding and a non-colliding placement.

4.6.3 Shape-based Penalty

The proxy provided by Algorithm 6 is a decent starting point for quantifying the severity of a collision. However, the area of overlap is not necessarily the only factor that influences how difficult a collision is to resolve. When quantifying collisions purely on the basis of the amount of overlap, we noticed that our search algorithm had a strong tendency to converge to situations with minor amounts of overlap between pairs of items with large and/or concave shapes. It seems that resolving such collisions is disproportionately more disruptive to the structure of the overall solution and therefore much harder compared to resolving collisions between smaller and/or more convex items.

To counter this, we experimented with adding a fixed term λ, that penalizes collisions involving such troublesome shapes. One candidate to base the penalty on is the area of a shape's convex hull, given that it captures both the size and the concavity of a shape. However, employing this area directly penalized these troublesome shapes too excessively. The square root of this area instead provides a much better balance. Therefore, we define shape S_a's penalty λ_a as:

$$\lambda_a = \sqrt{area(convex\ hull(S_a))} \tag{4.6}$$

Since a collision always takes place between a pair of shapes, we need to merge individual penalties. From our experiments, we found that the geometric mean of the two penalties was the most effective way to combine them. For a collision between S_a and S_b, we define the combined penalty as:

$$\lambda_{ab} = \sqrt{\lambda_a \cdot \lambda_b} \tag{4.7}$$

Algorithm 7 `quantify_collision`(S_a, S_b)

1: $\alpha \leftarrow$ `overlap_proxy_decay`(S_a, S_b) ▷ *Alg. 6*
2: $\lambda_{ab} \leftarrow \sqrt{\lambda_a \cdot \lambda_b}$ ▷ *Eq. 4.7*
3: **return** $\sqrt{\alpha} \cdot \lambda_{ab}$

Algorithm 7 presents the full procedure for quantifying a collision between two shapes. We first compute overlap proxy α, and then scale its square root by the combined penalty λ_{ab}. Taking the square root of α converts the proxy from an area-like (2D) metric into a one-dimensional one, making it more comparable to the penetration depth derivatives commonly used in other approaches. This function can now be used in Algorithm 4 to complete the evaluation of pairs of items.

4.7 Navigating a Continuous Search Space

All that now remains is to design a search algorithm that is able to gradually resolve collisions. One of the most fundamental components of any local search algorithm is the *move* operation, enabling the algorithm to transition from one solution to a neighboring one. Algorithm 8 introduces the local search move our approach will employ. Its core idea – repositioning every colliding item, one by one, in a randomized order – is shared by Elkeran [29], Sato et al. [82], and Umetani et al. [87].

Algorithm 8 `move_items`

1: $I_c \leftarrow \{i \in I : |$`jagua-rs::collisions`$(S_i)| > 1\}$ ▷ *colliding items*
2: **for** $i \in I_c$ in **a random order do**
3: $\quad t \leftarrow$ `search_position`$(()i)$ ▷ *Alg. 9*
4: $\quad$ `jagua-rs::move_item`(i, t)

In 2D irregular C&P problems, items can be positioned in a continuous 2D space and sometimes also rotated in a continuous manner. This means it is far too time-consuming to evaluate a full neighborhood of all possible item positions. Previous approaches have handled this continuous search space in a variety of ways. Sato et al. [82] discretize the search space into a (multi-resolution) grid. Meanwhile, approaches that rely on NFPs [29, 87] narrow the search space down to a set of positions along the edges of the union of all NFPs. From this (still continuous) set of positions, only a few specific points such as the vertices of the combined NFP are typically considered. All of these approaches effectively *restrict* the search space to a finite set of positions.

Instead of restricting the search space, we will preserve its continuous nature by relying on a *sampling* strategy to search for new item positions. Such an approach only evaluates a limited number of sampled positions, but every position remains a possible candidate. The aim is to obtain high-quality placements while keeping the number of samples to be evaluated as low as possible.

Algorithm 9 `search_position`(i)

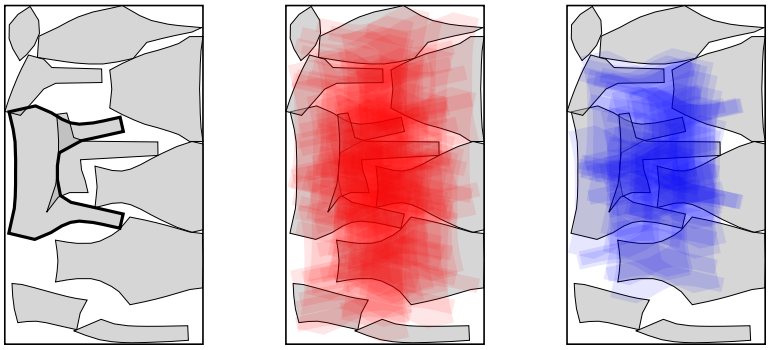
1: $f(i,t) \mapsto e \ \leftarrow$ `evaluate_sample`(i,t) ▷ *Alg. 10*
2: $t^* \leftarrow$ `perform_sampling`$(i, f(i,t) \mapsto e)$ ▷ *Fig. 4.7*
3: **return** t^*

Algorithm 9 indicates how we search for a new position for a single item i. A sample is defined as an item and an accompanying transformation: (i,t). We define a function to evaluate samples $f(i,t) \mapsto e$ which will, for now, remain abstract: it takes as input a sample and returns a single value e indicating the quality of that sample. This function, along with the item, is passed to the sampling procedure, which returns t^*, the best position it found to reposition item i to.

Thanks to the open-source nature of our implementation, we can visually illustrate how the `perform_sampling` function inside Algorithm 9 works. This allows us to provide an intuitive understanding of the process without relying on convoluted pseudocode or dense mathematical notation. Readers interested in the concrete implementation details can refer directly to the source code (Section 4.10).

Figure 4.7 illustrates the complete process of searching for a new position for the item highlighted with a thick border in Figure 4.7(a). First, we generate a number of samples uniformly at random within the container – the red overlays in Figure 4.7(b) – and collect them in a set T_{div}. Next, we perform a number of samples uniformly at random in the vicinity of the item's current position – the blue overlays in Figure 4.7(c) – resulting in set T_{foc}. This leads to a set of samples $T_{div} \cup T_{foc}$, which is both *diverse* within the container and also *focused* around the item's current position. While this set may contain promising samples, it is unlikely that any of them will yield a high-quality placement given they are all randomly generated.

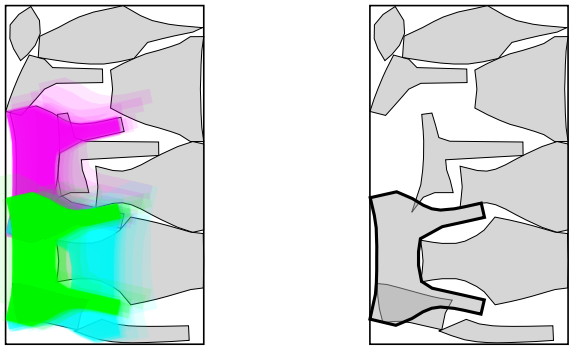
Figure 4.7: The `perform_sampling` procedure.



(a) Before

(b) Random samples in the strip: T_{div}

(c) Random samples near the current position: T_{foc}



(d) Refinement of 3 samples

(e) After

Note: A new position for a single item (in bold) is being searched.

We therefore select some of the most promising samples $\in T_{div} \cup T_{foc}$ to be *refined* into their respective local optima. A sample is considered promising if it is among the best evaluated and its transformation is sufficiently different from the other samples already selected. Figure 4.7(d) visualizes this refinement process for three samples that are highlighted in lime, cyan and purple. The refinement procedure is inspired by the *adaptive coordinate descent* approach described by Loshchilov et al. [66] and greedily shifts sampling in the direction of improving evaluation. Figure 4.7(e) shows the position eventually chosen to move the item to.

Provided that the cost of evaluating a sample is low enough, this approach is an effective way to find high-quality placements while remaining completely agnostic to the underlying implementation of $f(i, t) \mapsto e$.

4.8 Solving the Feasibility Problem

We have introduced how to detect collisions, how to quantify those collisions, and how to move from one solution to a neighboring one by repositioning colliding items. The final step in solving the feasibility problem is to combine all these components into a complete **search algorithm**, which gradually resolves collisions with the ultimate aim of reaching a feasible configuration of items.

4.8.1 Guided Local Search

How to search for a new position was explained in Algorithm 9, but the function to evaluate samples was left abstract. A straightforward way to evaluate a sample would be to aggregate the severity of all collisions that would occur if the item were placed at the sampled position, akin to Algorithm 4. However, such a *static* evaluation function would cause the search to converge to a local optimum within a couple of iterations. Approaches proposed by Umetani et al. [87] and Sato et al. [82] avoid this issue by relying on Guided Local Search (GLS): a metaheuristic introduced by Voudouris et al. [92] that *dynamically* modifies the evaluation function to escape local optima and move towards promising regions of the search space.

Algorithm 10 `evaluate_sample`(i, t)

1: $C \leftarrow$ `jagua-rs::collisions`$(t(S_i))$
2: **for** $c \in C \setminus \{i\}$ **do** ▷ *ignore item i itself*
3: $\quad w_{ic} \leftarrow$ weight of item-pair (i, c)
4: $\quad e \leftarrow e + w_{ic} \cdot$ `quantify_collision`$(t(S_i), S_c)$ ▷ *Alg. 7*
5: **return** e

Algorithm 10 introduces how we evaluate a sample (i, t). First, the CDE is queried to retrieve the set of items C that would be colliding with item i if it were moved to the position defined by transformation t. These prospective collisions are quantified, multiplied by the corresponding item-pair weight w_{ic}, and summed up to a total evaluation value e. These dynamic **item-pair weights** are the mechanism through which GLS modifies the evaluation function, thereby influencing the search process.

Algorithm 11 `update_weights`

$e_{\max} \leftarrow \max\{\texttt{evaluate_item_pair}(a, b) : a, b \in I, a \neq b\}$ ▷ *Alg. 4*
for $a, b \in I : a \neq b$ **do** ▷ *all pairs of items*
  $e \leftarrow \texttt{evaluate_item_pair}(a, b)$ ▷ *Alg. 4*
  if $e > 0$ **then**
    $m \leftarrow \texttt{M}_l + (\texttt{M}_u - \texttt{M}_l) \cdot (e / e_{\max})$
  else
    $m \leftarrow \texttt{M}_d$
  $w_{ab} \leftarrow \max\{1, w_{ab} \cdot m\}$

All item-pair weights are initialized to 1 and every time Algorithm 11 is called, each weight is updated by multiplying it by its corresponding factor m. If the pair of items is colliding, m represents a linear mapping $\in]\texttt{M}_l, \texttt{M}_u]$ of the severity of their collision e relative to the most severe collision $e_{\max}$. Item pairs with more severe collisions will thus have their weight rise faster than those with less severe collisions. If a pair is not in collision, their weight is multiplied by $\texttt{M}_d$ (< 1) and eventually decays back to 1 over time.

4.8.2 Separation Procedure

Simply looping `move_items` (Algorithm 8) and `update_weights` (Algorithm 11) would technically suffice to traverse the solution space, but this approach lacks any mechanism to backtrack to previous solutions. A more sophisticated approach that incorporates this ability should improve the effectiveness of the search algorithm.

Let us begin by defining the evaluation function for the entire solution:

$$\sum_{\substack{a,b \in I \\ a \neq b}} \texttt{evaluate_item_pair}(a, b) \tag{4.8}$$

This function is the sum of the evaluation of every item pair (Algorithm 4). Whenever Equation 4.8 returns 0, all collisions have been resolved and a feasible solution has been reached.

Algorithm 12 `separate`$(k_{\max}, n_{\max})$

```
1:  s* ← current solution ; e* ← Eq. 4.8 ; k ← 0
2:  while k < k_max and e* > 0 do
3:      restore to s* ; s_init ← s* ; n ← 0
4:      while n < n_max and e* > 0 do                ▷ an attempt
5:          move_items_multi                         ▷ Alg. 13
6:          update_weights                           ▷ Alg. 11
7:          e ← Eq. 4.8 ; n ← n + 1
8:          if e < e* then
9:              s* ← current solution ; e* ← e
10:             n ← 0
11:     k ← k + 1                                    ▷ add a strike
12:     if s* ≠ s_init then
13:         k ← 0                                    ▷ reset strikes
14: return s*
```

Algorithm 12 describes the complete procedure to solve the feasibility problem introduced in Section 4.4. The separation procedure starts from an infeasible situation with colliding items and attempts to resolve all of its collisions. At its core, the algorithm continually repositions items, updates the item-pair weights and replaces the incumbent solution s^* whenever improvements are found with respect to Equation 4.8. Whenever feasibility is reached, the procedure stops and immediately returns the solution. Once a certain number of iterations without improvements $n_{\max}$ is reached, we conclude one *attempt*. If the incumbent solution was never updated during this attempt, a *strike* is added. Conversely, if the attempt improved s^*, the strike counter k is reset to 0. As long as the number of strikes is below a certain threshold $k_{\max}$, the procedure restores the incumbent solution s^* and starts a new attempt, continuing its quest for a feasible solution. Once the number of strikes exceeds $k_{\max}$, the procedure halts the search and returns s^*, which will unfortunately still be infeasible.

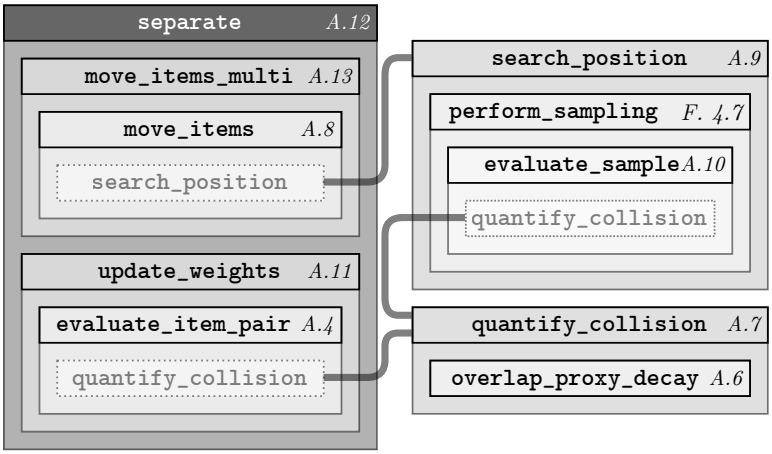


Figure 4.8: The `separate` procedure and all its underlying components presented in a hierarchical overview. Every component references its corresponding algorithm ($A.*$) or figure ($F.*$).

The behavior we intend to generate through this procedure can be described as follows. GLS ensures that item pairs colliding across consecutive iterations quickly reach prohibitively large weights, especially pairs that are involved in the most severe collisions. By contrast, when a pair of items manages to resolve its collision, the weight slowly decays back down to 1. When a collision occurs in a region with low GLS weights, the items involved will often 'transfer' the collision to neighboring pairs. Observing this process over multiple iterations reveals that items effectively *push* and *pull* each other, constantly 'fighting' for their own space in the container. When the search inevitably stalls, some items start to accumulate significant weights with all their neighbors. This will cause them to *jump* away to an entirely different position in the container, effectively escaping the local optimum. Readers who are curious to see this behavior in action are referred to the repository (Section 4.10), which contains a simple visualizer.

Figure 4.8 provides an overview of the `separate` procedure, where each of the previously introduced components are hierarchically related with one another.

4.8.3 Balancing Exploration and Exploitation

The random order in which Algorithm 8 repositions colliding items ensures that the search algorithm can explore a wide range of neighboring solutions. However, in many cases, this random order might be far from ideal. In order to strike a better balance between exploration and exploitation, Algorithm 13 introduces an enhanced approach to move from one solution to a neighboring one.

Algorithm 13 `move_items_multi`

```
1: s_i ← current solution ; e* ← ∞
2: for N_WORKERS times do                          ▷ parallelized
3:     move_items                                  ▷ Alg. 8
4:     e ← Eq. 4.8
5:     if e < e* then
6:         s* ← current solution ; e* ← e
7:     restore to s_i
8: restore to s*
```

Algorithm 13 essentially executes Algorithm 8 multiple times with different random orders of items. From these multiple neighboring solutions, the best one (Equation 4.8) is chosen. This significantly increases the chances of finding improvement while preserving the diversity of the search process. This enhanced version is employed in the separation procedure (Algorithm 12) to solve the feasibility problem. Although presented here sequentially, Algorithm 13 is parallelized across a pool of `N_WORKERS` threads in our implementation.

4.9 Solving the 2D Irregular Strip Packing Problem

Algorithm 12, which targets the feasibility problem introduced in Section 4.4, can now be integrated into a complete heuristic to solve the actual 2DISPP. The complete heuristic we propose is introduced in Algorithm 14 and is divided into two distinct phases. The **exploration** phase first provides a lot of *freedom* to enable investigation of a wide range of solutions. Next, the **compression** phase attempts to shrink the strip as much as possible while remaining *constrained* to the general configuration of items found at the end of the exploration phase.

Algorithm 14 `solve_ispp`

```
s* ← explore                                   ▷ Alg. 15
s* ← compress(s*)                              ▷ Alg. 16
return s*
```

4.9.1 Exploration Phase

Algorithm 15 `explore`

```
s* ← construct initial solution and shrink strip by R_x
while TL_x not reached do
    s ← separate(K_x, N_x)                     ▷ Alg. 12
    if s is feasible then
        s* ← s
        shrink strip by R_x
        S ← ∅
    else
        S ← S ∪ {s}                            ▷ pool of infeasible solutions
        ŝ ← select from S
        s' ← disrupt ŝ by swapping two large items
        restore to s'
return s*
```

Algorithm 15 describes the exploration phase, which starts from an initial solution created by a simple bottom-left-fill heuristic. This constructive heuristic will provide a feasible solution, after which the strip is shrunk by a factor $\mathtt{R}_x$. Each time the strip is shrunk, all items have to be contained within these new boundaries. This is achieved by selecting a certain vertical axis within the strip and shifting items positioned right of this axis to the left, most likely creating multiple minor collisions.

The separation procedure (Algorithm 12) attempts to convert this infeasible solution into a feasible one. If successful, the returned solution s replaces the incumbent solution s^*, after which the strip is once again shrunk. If unsuccessful, s represents an infeasible local optimum that is added to a pool of solutions $\mathcal{S}$. For the next separation attempt, a random solution $\hat{s}$ is chosen from $\mathcal{S}$, with those closer to feasibility (Equation 4.8) having a greater chance of being selected. A new starting solution s' is then created by swapping two large items from $\hat{s}$. This ensures s' inherits many high-quality placements from $\hat{s}$ while, at the same time, being sufficiently disrupted so it can escape from the local optimum. This procedure continues until time limit $\texttt{TL}_x$ is reached, at which point the exploration phase returns the best feasible solution found s^*.

4.9.2 Compression Phase

Algorithm 16 `compress`(s^*)

```
1: while TL_c not reached do
2:     τ ← elapsed time in phase
3:     r ← R_c^s + (R_c^e − R_c^s) · (τ/TL_c)
4:     restore to s* and shrink strip by r
5:     s' ← separate(K_c, N_c)                    ▷ Alg. 12
6:     if s' is feasible then
7:         s* ← s'
8: return s*
```

Algorithm 16 describes the compression phase, which starts from the best feasible solution s^* the exploration phase generated. The procedure iteratively (i) restores to the incumbent solution s^*, (ii) shrinks the bin by ratio r and (iii) attempts to separate. If successful, the incumbent solution is updated. If unsuccessful, the incumbent solution is restored and the process reattempted.

Shrink ratio r is much smaller than during the exploration phase and decays linearly from $\texttt{R}_c^s$ to $\texttt{R}_c^e$ based on the phase's remaining time limit $\texttt{TL}_c$. This means that the separate procedure only needs to resolve (progressively) less severe collisions, which it is often able to do without making major changes to the configuration of items. Another important difference with the exploration phase is that, before each new separation attempt, the best feasible solution s^* is always restored and the strip is shrunk again. This causes the search process' freedom to be significantly more restricted than during the exploration phase. The compression phase is squarely focused on finding out how much further the strip can be shrunk without deviating too far from the configuration of items provided by the exploration phase.

4.10 Implementation

The implementation of our 2D nesting algorithm is written in Rust and called `sparrow`. It is publicly available at: https://github.com/JeroenGar/sparrow. The codebase closely follows the structure of this chapter, making it straightforward to understand how `sparrow` relates to the algorithms introduced in this chapter. Throughout this project, we have strived for maximum computational performance, robustness and leanness of both the algorithm and its implementation.

The result of these priorities is an implementation that is relatively insensitive to the complexity and the number of shapes involved, making `sparrow` suitable as either a practical tool to tackle real-world problems or a basis for future research. The contribution of each individual algorithmic component to the overall performance of `sparrow` was carefully evaluated and those without any significant contribution were eliminated. As a result, the components presented in this chapter represent an irreducible set required to achieve the performance demonstrated in Section 4.11.

It is important to note that, in contrast to some other approaches, `sparrow` will never produce a feasible solution where items are 'touching' the container or each other. Instead, items will always be completely separated, if only by a minuscule distance. Exact fits – where moving the position of an item by even the smallest representable distance causes a collision – cannot be produced by `sparrow`.

There are two reasons for this. First, whenever two entities are extremely close to each other, `jagua-rs` will err on the side of caution and report a collision. The CDE was deliberately designed this way in order to avoid numerical instability due to floating point arithmetic. Second, when searching new positions for an item, `sparrow` relies on a sampling approach (Section 4.7). While the sampling is configured to be very precise, it cannot target one specific coordinate.

So far, this limitation concerning exact fits only surfaces in artificial academic instances featuring simple shapes and vertices with integer coordinates. Conditions that, generally, do not occur in real-world applications.

4.11 Computational Experiments

In this section we will evaluate the performance of `sparrow` by conducting a series of computational experiments on a set of academic benchmark instances.

4.11.1 Experimental Setup

All experiments were performed on a machine equipped with an AMD Ryzen™ 9 7950X CPU. System memory is irrelevant as the entire program easily fits within the 64 MB cache of the CPU. Unless specified otherwise, each experiment was run for 20 minutes, with 3 threads active, and configured with the parameters presented in Table 4.1.

Table 4.1: Configuration of `sparrow`'s parameters throughout the experiments.

$\texttt{R}_\varepsilon \leftarrow 1\%$	▷ *Alg. 6 – diameter ratio to compute ε*
$(\texttt{M}_u, \texttt{M}_l, \texttt{M}_d) \leftarrow (2.0, 1.2, 0.95)$	▷ *Alg. 11 – upper, lower and decay weight multipliers*
`N_WORKERS` $\leftarrow 3$	▷ *Alg. 13 – how many times `move_items` is executed*
$(\texttt{K}_x, \texttt{N}_x) \leftarrow (3, 200)$	▷ *Alg. 15 – separation parameters for exploration*
$(\texttt{K}_c, \texttt{N}_c) \leftarrow (5, 100)$	▷ *Alg. 16 – separation parameters for compression*
$\texttt{R}_x \leftarrow 0.1\%$	▷ *Alg. 15 – exploration shrink ratio*
$(\texttt{R}_c^s, \texttt{R}_c^e) \leftarrow (0.05\%, 0.001\%)$	▷ *Alg. 16 – compression shrink ratio range*
$(\texttt{TL}_x, \texttt{TL}_c) \leftarrow (0.8, 0.2) \cdot 20'$	▷ *Alg. 15,16 – time limits*

Note: The right column contains the relevant algorithm(s) and a short description for every entry.

We used a combination of manual tuning and hyperparameter optimization [12] to identify a suitable set of parameters for the algorithm. These parameters were optimized for the maximum expected performance across all instances from the academic benchmark set when subject to a 20-minute time limit. This single set of parameters was used for every instance in our experiments.

4.11.2 Performance Analysis

To evaluate the performance of `sparrow` we will use academic benchmark instances from the ESICUP website[2]. We refer to Table 3 in Sato et al. [82] for an overview of these instances and their properties.

To provide a sense of the efficiency of the implementation and scale of operations, when using the aforementioned experimental setup, `sparrow` is able to evaluate samples (Algorithm 10) at a rate of approximately 2.4 million per second with `SWIM`, which represents the most complex instance in the academic benchmark set.

[2] https://www.euro-online.org/websites/esicup/data-sets

Throughout the experiments, we report the quality of a solution in terms of its packing density ρ. This is the ratio of the area of the items in the strip with respect to the area of the strip itself expressed as a percentage:

$$\rho = \frac{100}{w \cdot l} \cdot \sum_{i \in I} \texttt{area}(S_i) \tag{4.9}$$

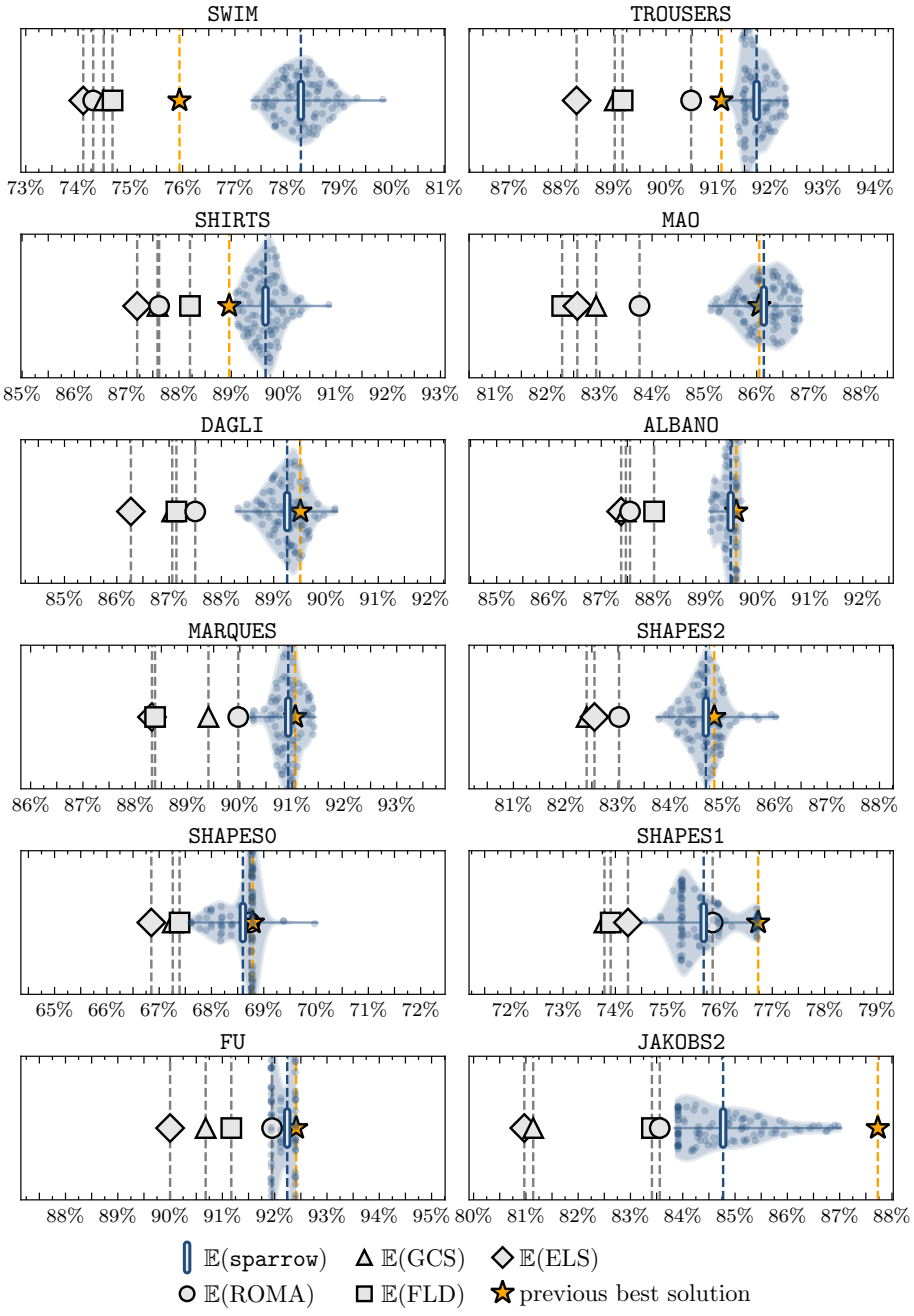
The expected quality $\mathbb{E}(\rho)$ and interquartile range of `sparrow`'s solutions in comparison to the academic SotA algorithms for the 2DISPP and one open-source implementation are presented in Table 4.2, the caption of which provides further detail on the setup of this comparison. The distributions of the solutions from Table 4.2 are visualized as violin plots in Figure 4.9. One important thing to note is that the width of the strip for some of the most simple instances – `SHAPES0, SHAPES1, SHAPES2, FU, JAKOBS1 and JAKOBS2` – was inflated by 0.01%. This minuscule inflation should not affect which configurations of items are possible, but suffices to avoid the issue concerning exact fits (described in Section 4.10), which would otherwise occur in these artificial instances. For academic purposes, `sparrow`'s best solution ever produced for each instance, alongside the previous best-known solution, is presented in Table 4.3 and shown in Figure 4.11. It is important to note that these 'best' solutions, although conceived under the same experimental conditions and time limit, are not necessarily elements of the sets of 100 independent runs from which the expected densities in Table 4.2 have been derived.

`sparrow` generally dominates in terms of expected solution quality, both on individual instances and across the entire benchmark set. Figure 4.9 reveals that for the three largest instances in terms of number of items and complexity of their shapes – `SWIM`, `SHIRTS` and `TROUSERS` – `sparrow`'s performance distribution lies entirely beyond the previous best-known singular solution. Indeed, we were able to match or improve upon every previous best-known solution, in some cases by a significant margin.

The algorithms composing the academic SotA report only two values per instance: the best and the expected density from a limited number of runs. This limits our ability to provide detailed statistical or visual performance comparisons against them. However, to support future in-depth comparisons against `sparrow`, we provide all solution data and a complete reproducibility guide in the repository.

In conclusion, `sparrow` establishes a new state of the art for the 2DISPP, consistently producing solutions whose quality, in most cases, significantly exceeds any previous academic or open-source results.

Figure 4.9: Distribution of `sparrow`'s solution quality ρ across the academic benchmark instances.



Note: Each dot represents an individual solution from the experiments composing Table 4.2. The expected solution qualities of ROMA - Sato et al. [82], GCS - Elkeran [29], FLD - Wang et al. [93] and ELS - Leung et al. [61] algorithms are shown. A star marks the previous singular best-known solution for each instance. `JAKOBS1` is omitted since it virtually always produced the same density.

Table 4.2: Comparison of the expected solution quality $\mathbb{E}(\rho)$ of `sparrow` on academic benchmarks.

	sparrow			ROMA	GCS	FLD	ELS	PS
instance	Q1(ρ)	Q3(ρ)	$\mathbb{E}(\rho)$	$\mathbb{E}(\rho)$	$\mathbb{E}(\rho)$	$\mathbb{E}(\rho)$	$\mathbb{E}(\rho)$	ρ
ALBANO	89.39	89.58	**89.47** ±0.03	87.55	87.47	88.01	87.38	84.95
DAGLI	89.01	89.50	**89.26** ±0.07	87.50	87.06	87.14	86.27	84.17
FU	91.93	92.40	**92.24** ±0.04	91.95	90.68	91.17	90.00	90.39
JAKOBS1	89.09	89.09	**89.09** ±0.00	**89.09**	88.90	88.96	88.35	81.67
JAKOBS2	83.91	85.25	**84.77** ±0.17	83.56	81.14	83.41	80.97	80.42
MAO	85.84	86.45	**86.14** ±0.08	83.76	82.93	82.28	82.57	75.94
MARQUES	90.78	91.02	**90.93** ±0.05	89.97	89.40	88.38	88.32	85.48
SHAPES0	68.38	68.79	68.60 ±0.07	**68.73**	67.26	67.39	66.85	66.50
SHAPES1	75.26	75.97	75.69 ±0.11	**75.86**	73.79	73.91	74.24	72.55
SHAPES2	84.44	84.88	84.68 ±0.07	83.02	82.40	-	82.55	**85.49**
SHIRTS	89.43	89.84	**89.66** ±0.06	87.62	87.59	88.21	87.20	85.99
SWIM	77.94	78.55	**78.26** ±0.09	74.29	74.49	74.66	74.10	71.44
TROUSERS	91.51	91.93	**91.73** ±0.05	90.48	89.02	89.17	88.29	89.30

Note: The expected solution quality $\mathbb{E}(\rho)$ on academic benchmarks obtained by `sparrow`, four algorithms from the literature: ROMA - Sato et al. [82], GCS - Elkeran [29], FLD - Wang et al. [93], ELS - Leung et al. [61] and one open-source implementation: PS - Fontan [35]. For `sparrow`, quartiles (Q1, Q3) and the margin of error (±) on $\mathbb{E}$, defined as the half-width of the 95% confidence interval computed via non-parametric bootstrapping, are reported. The reported values for `sparrow` result from 100 independent runs, each with a time limit of 20 minutes. Values for ROMA, GCS, FLD and ELS were taken from their respective publications. PS is a deterministic algorithm and was run once under the same conditions as `sparrow`. For each instance, the best $\mathbb{E}(\rho)$ is highlighted in bold.

4.11.3 Time-sensitivity Analysis

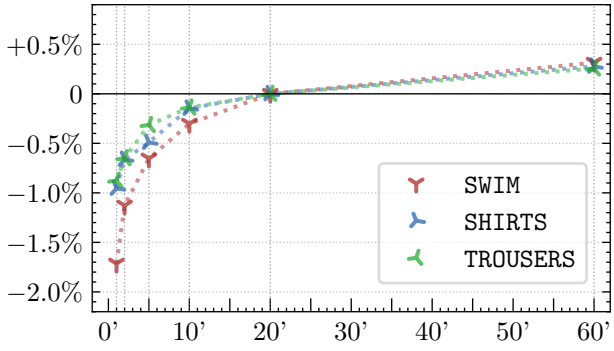
Different use cases call for varying computational budgets, and therefore it is important to understand how performance scales with time. Figure 4.10 presents a time-sensitivity analysis of `sparrow`'s expected performance for the three most complex instances from the academic benchmark set, normalized to the 20-minute time limit from Table 4.2. No specific tuning was performed to account for the different time limits. `sparrow` is able to quickly produce solutions within a couple of percentage points of the achievable quality after just 20 minutes. Beyond 20 minutes, the expected solution quality continues to improve, albeit at a diminishing rate.

Table 4.3: The best solution densities ever produced by `sparrow` for instances in the benchmark set

instance	`sparrow` ρ	previous best ρ	by
ALBANO	**89.82**	89.58	E
DAGLI	**90.17**	89.51	E
FU	**92.41**	**92.41**	E
JAKOBS1	**89.26**	89.09	L/E/W/S
JAKOBS2	**87.73**	**87.73**	E/S
MAO	**86.87**	86.05	S
MARQUES	**92.02**	91.02	S
SHAPES0	**69.98**	68.79	E/W/S
SHAPES1	**76.73**	**76.73**	E/S
SHAPES2	**86.23**	84.84	E
SHIRTS	**90.92**	88.96	E/W
SWIM	**79.83**	75.94	E
TROUSERS	**92.62**	91.06	S

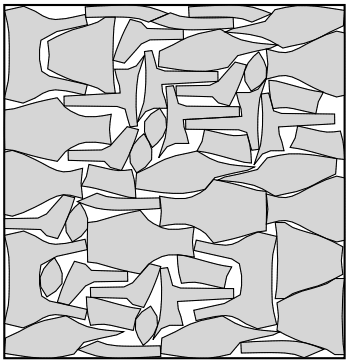
Note: The right column contains the previous best-known density and the algorithm that produced it: L - Leung et al. [61]. E - Elkeran [29], W - Wang et al. [93], S - Sato et al. [82]. Best densities are highlighted in bold. Time limit is 20 minutes.

Figure 4.10: Scaling of the expected solution quality $\mathbb{E}(\rho)$ of `sparrow` for the three most challenging instances across different time limits.

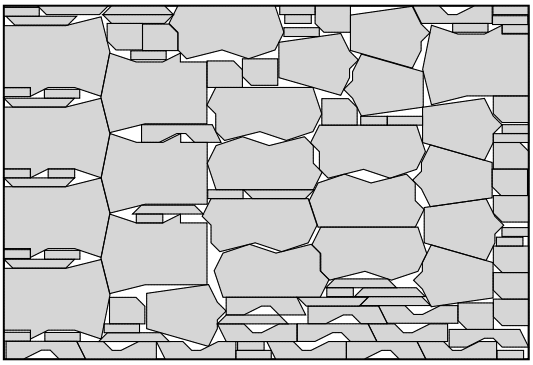


Note: The data points are normalized for 20 minutes, and each one is composed of 20 independent runs. No specific tuning was performed to account for the different time limits.

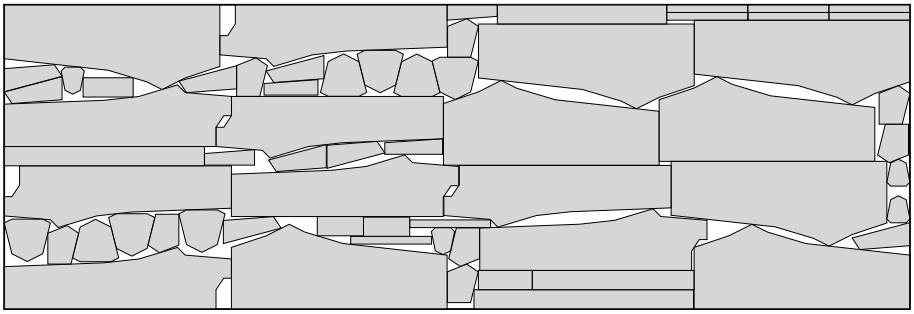
Figure 4.11: The best solutions obtained by `sparrow` (Table 4.3) and their respective densities.



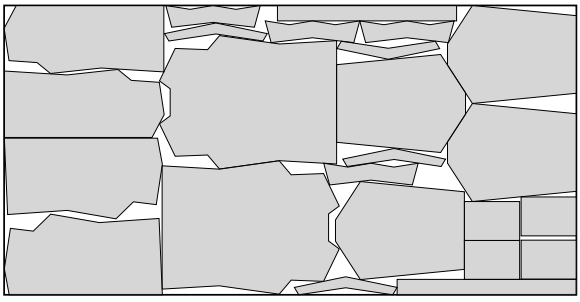
(a) SWIM - **79.83**%



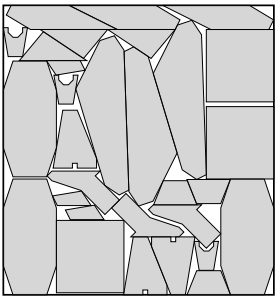
(b) SHIRTS - **90.92**%



(c) TROUSERS - **92.62**%



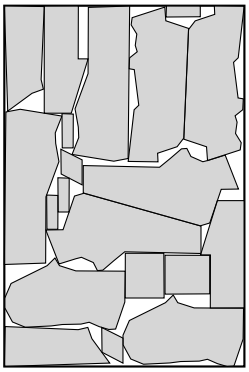
(d) ALBANO - **89.82**%



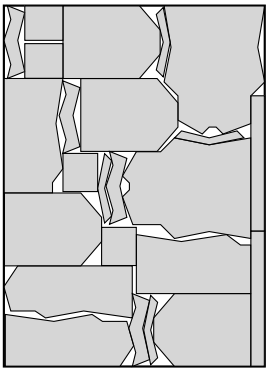
(e) DAGLI - **90.17**%

Note: Bold values indicate improved best-known solutions.

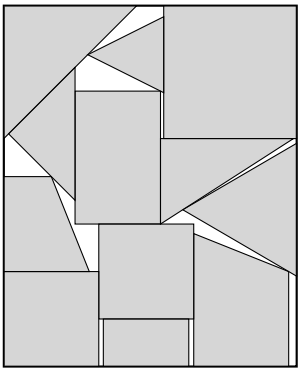
Figure 4.11: The best solutions obtained by `sparrow` (Table 4.3) and their respective densities. (continued)



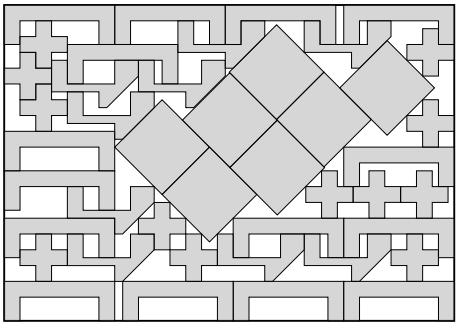
(f) MAO - **86.87**%



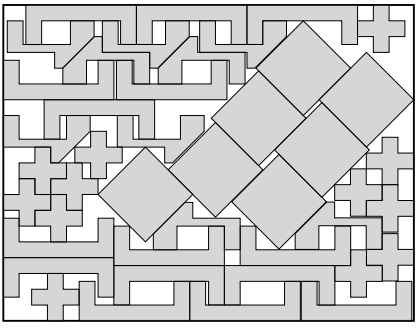
(g) MARQUES - **92.02**%



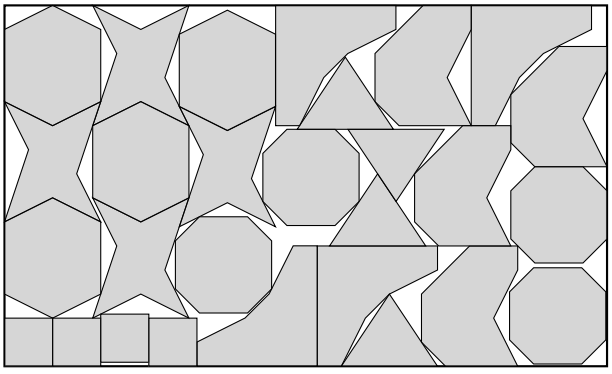
(h) FU - 92.41%



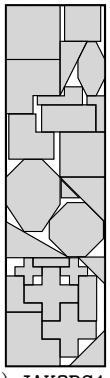
(i) SHAPES0 - **69.98**%



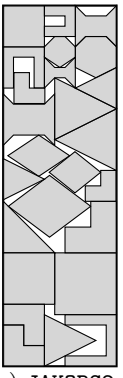
(j) SHAPES1 - 76.73%



(k) SHAPES2 - **86.23**%



(l) JAKOBS1 - **89.26**%



(m) JAKOBS2 - 87.73%

Note: Bold values indicate improved best-known solutions.

4.12 Real-world Benchmarks & Future Research Directions

In Section 4.11.2, we demonstrated that `sparrow` is capable of consistently producing high-quality solutions across the entire set of traditional academic benchmark instances. This set, however, is dated and contains many instances that were constructed artificially, featuring simple polygonal shapes with vertices located at integer coordinates. For `SWIM`, `TROUSERS` and `SHIRTS`, arguably the instances that most reflect real-world conditions, `sparrow` makes a significant leap forward. However, these instances are few in number and only represent a single application domain: the garment industry. We therefore take the opportunity to extend the set of academic benchmarks to better capture the breadth and intricacies of real-world applications.

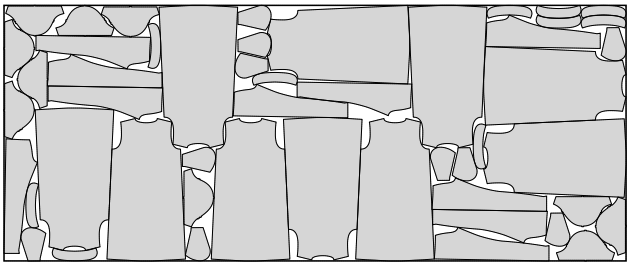
We introduce ten new instances that span a variety of application domains and contain both homogeneous and heterogeneous sets of items, as well as narrow and wider strips. Every item is represented as a simple polygon and two variants are provided for each instance: one which only allows discrete rotations in 90-degree increments and a second which allows continuous rotation. The full dataset and experimental results are available in `sparrow`'s repository and in the ESICUP datasets [3].

Figure 4.12 presents this new suite instances for the 2DISPP. Instances `GARDEYN0 - 3` have been derived from Lallier et al. [57], a large dataset of nesting jobs originating from fashion industry clients of Lectra[4]. Instances `GARDEYN4 - 8` are based on shapes from metalworking and shipbuilding applications, provided by AlmaCAM[5]. Finally, `GARDEYN9` represents a direct-to-film printing context, contributed by Tetrinest[6].

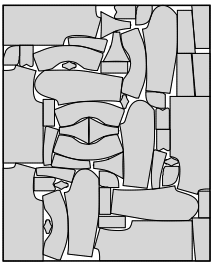
These new instances revealed a serious blind spot in the traditional academic benchmarks. Highly homogeneous instances – such as `GARDEYN4`, `5`, and `7` – often contain subsets of items that can be clustered together in a compact way, and repeated many times throughout the strip. While `sparrow` can generate these compact patterns at a local scale, it lacks a mechanism to repeat them and is therefore unable to exploit the inherent 'structure' or 'regularity' made possible by the homogeneity of the items. This characteristic is prevalent in real-world applications, but was not captured by the traditional academic benchmarks.

[3] https://github.com/ESICUP/datasets
[4] https://lectra.com/en
[5] https://almacam.com/
[6] https://tetrinest.com/

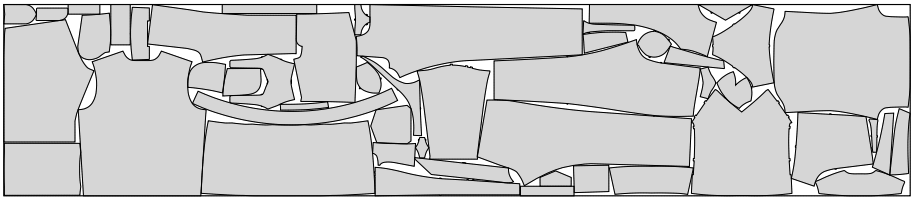
Figure 4.12: Ten new benchmark instances for the 2DISPP.



(a) GARDEYN0



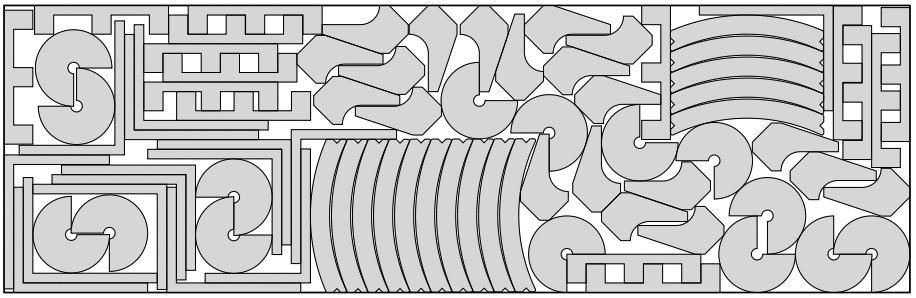
(b) GARDEYN1



(c) GARDEYN2



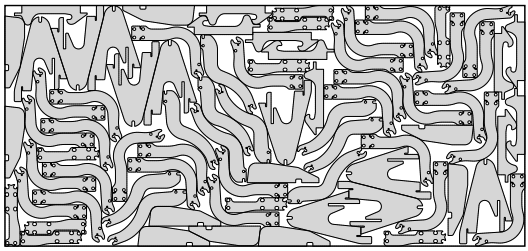
(d) GARDEYN3



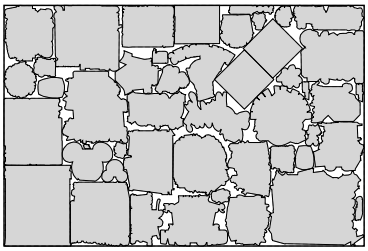
(e) GARDEYN4

Note: The visualizations are solutions produced by `sparrow`.

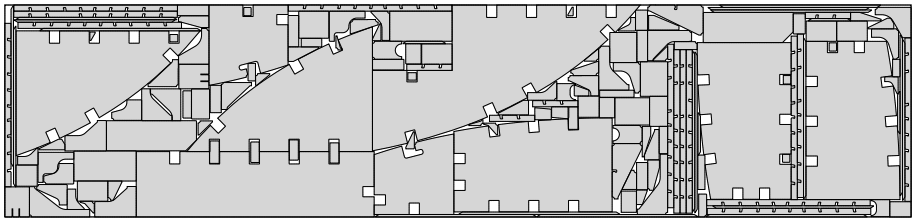
Figure 4.12: Ten new benchmark instances for the 2DISPP. (continued)



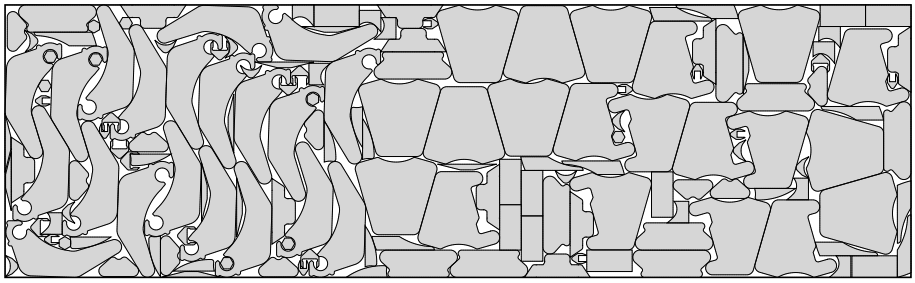
(f) GARDEYN5



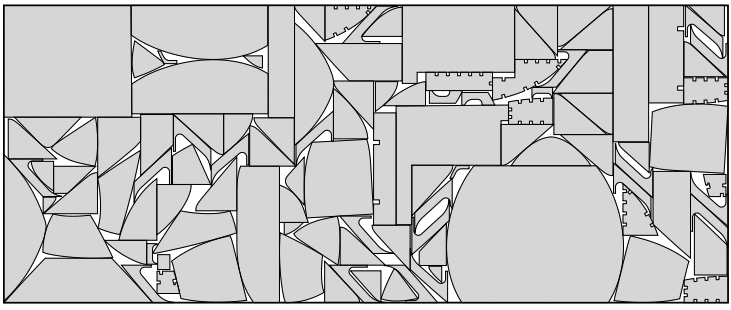
(g) GARDEYN9



(h) GARDEYN6



(i) GARDEYN7



(j) GARDEYN8

Note: The visualizations are solutions produced by `sparrow`.

Extending `sparrow` with the ability to exploit this inherent structure/regularity is a promising direction for future advances, as we are confident vastly superior solutions are attainable, especially for the aforementioned homogeneous instances.

4.13 Conclusions and Reflections

This chapter introduced a heuristic for the 2D irregular strip packing problem (2DISPP) that redefines both the academic and open-source state of the art. Our implementation, `sparrow`, is publicly available at: https://github.com/JeroenGar/sparrow. The core idea of the algorithm is to decompose the 2DISPP into a sequence of feasibility problems, where the strip is converted into a container with fixed dimensions and the items are temporarily allowed to collide with each other. A local-search algorithm then iteratively moves colliding items around in an attempt to gradually resolve them all and eventually reach feasibility. Although we focus on the 2DISPP, most of the heuristic's components operate on the feasibility problem and can easily be incorporated into solution strategies for other 2D nesting problems such as bin packing or knapsack problems.

The main contributions of this work are fourfold: (i) a reproducible and substantial improvement in solution quality across both academic and open-source contexts, (ii) a highly-optimized and robust implementation ready for deployment in practice or as a base for future research, (iii) a number of foundational algorithmic ideas that can be iteratively improved upon or adapted to other problem variants and (iv) a new set of academic benchmark instances designed to properly reflect real-world nesting applications and uncover weaknesses in our approach.

To be frank, the scale of improvements with respect to the previous SotA demonstrated in the computational experiments (Section 4.11) should not have been possible on such a well-studied problem using benchmark instances that are over 25 years old. On the most challenging instance, `SWIM`, `sparrow`'s expected and best solution quality are both ∼5% more compact than the previous best results. Such a margin of improvement confirms that stagnation with respect to 2DISPP research was not a result of it already being solved to near optimality.

Progress in this field has instead been held back for far too long by the issues we outlined in the introduction: the persistence of high barrier to entry and a severe lack of reproducibility. In an alternate scenario – where this work were published without its open-source implementation – the method we propose would have simply continued that trend and been nearly impossible to reproduce in practice. Even worse, by raising the performance bar while simultaneously leaving others unable to meaningfully verify or build upon our work, we could have inadvertently initiated a new decade of stagnation.

Reproducibility is a cornerstone of the scientific method. Without it, researchers are left in an unhealthy and inefficient environment: forced to reinvent the wheel without the means to increment on prior work or even validate its results. We strongly believe that an academic paper coupled with a high-quality and open-source implementation delivers value significantly exceeding the sum of its parts. Thankfully, the ethos we are advocating is not swimming against the current. The field of Operations Research is undergoing a gradual cultural shift toward open-source practices [77]. Vidal [90], for example, recently published an open-source implementation of their state-of-the-art algorithm for the capacitated vehicle routing problem. Many of the concerns we raise here are in fact echoing sentiments already expressed in Vidal [90].

We hope this chapter and `sparrow` will serve as catalysts for continued advances in 2D nesting research as well as deterrents to non-reproducible research practices that have historically hindered progress.

Chapter 5

Conclusions and Reflections

A common personal goal among PhD students is for their work not to fade into obscurity but to continue providing value long after the dissertation is completed. Over the course of my doctoral trajectory, the following research hypothesis gradually emerged:

> *The pursuit of simple, flexible methods – supported by polished open-source implementations – offers one of the most promising paths toward achieving lasting value and impact for the research community.*

The agenda of this thesis was neither preconceived nor motivated by altruism. Instead, it evolved naturally throughout the course of the research and aligns well with my own ambitions and values. Chapters 2, 3 and 4 are presented in the order they were written and, when read sequentially, reflect the gradual emergence of this agenda.

While the hypothesis we introduced at the beginning of the thesis cannot be proven here, we can offer anecdotal evidence and arguments in its favor.

Part I of the thesis focused on rectangular C&P problems, with Chapter 2 introducing `GDRR`: a heuristic for solving the well-researched 2D bin packing problem with guillotine constraints. On the classic benchmark instances with just a single bin type, `GDRR` proved competitive both with and without 90-degree rotation of items. On benchmarks featuring multiple bin types, it achieved a clear improvement over existing methods.

Driven by the project's requirements, the algorithm was designed with simplicity and flexibility in mind. These priorities are evidenced by a reimplementation and adaptation by Fernandes et al. [31], who, after reviewing the academic literature, chose `GDRR` as the basis for their own approach. They adapted the base algorithm slightly and extended it with support for an additional constraint that arises in a real-world industrial context.

The original publication lacked an open-source implementation and was therefore not properly reproducible. At the start of Part II, I revisited `GDRR`, reimplemented it from scratch in Rust and released it publicly. Due to multiple compounding factors, the rewrite is orders of magnitude faster than the version used for the original publication.

Although Fernandes et al. [31] appear to have faithfully captured the original methodology from our description, the results they report after 25 minutes can be produced within seconds using our open-source reimplementation. This should not be misconstrued as a critique of their work, but rather serve as a testament to the critical role the implementation plays. The effectiveness of methods, and in effect the contribution, often stands or falls with the quality of their implementation.

In Part II, reproducibility became a priority rather than an afterthought and we shifted our focus to irregular C&P (nesting) problems. A field which, at the time, was showing serious signs of stagnation. In hindsight, it became clear that this stagnation was not a consequence of our ability to solve these problems to near-optimality. Rather, the fact that virtually nobody was sharing implementations combined with the inherently high barrier to entry created the perfect conditions for the erosion of reproducibility and, in effect, stagnation to set in.

In Chapter 3, we introduced the first part of our attempt to change this with `jagua-rs`: a *Collision Detection Engine* (CDE) for 2D irregular C&P problems. When we entered the field four years ago, every new researcher had to start from a very similar platform as those who came years earlier, while having to clear ever-higher hurdles. Today, `jagua-rs` fundamentally lowers the barrier to entry by taking full responsibility over the geometric challenge, providing a robust, fast, and accurate way of checking for collisions. This frees researchers to focus on the already demanding optimization challenge of 2D nesting.

Simplicity within the CDE shows both in the low-level operations – understandable with only high-school geometry – and in the high-level interface, where optimization algorithms interact through simple *queries* and *updates*. While collision detection remains complex, for a user of `jagua-rs`, engaging with that complexity is a voluntary choice, as it is encapsulated between two layers of simplicity.

Flexibility of the CDE is evident in its support for continuous translation, continuous rotation, irregular containers, and its relative insensitivity to the complexity of shapes. The introduction of abstract *hazards* further generalizes the engine. Within the CDE there is no notion of containers, items or special zones. Only hazards exist, making it agnostic to the particular 2D nesting variant being addressed.

In the short time since release, `jagua-rs` has gained worldwide traction. Users are already building alternative algorithms on top of it and contributing improvements to its core geometry routines. Its reliability and robustness, aided by our choice to write it in Rust, are being thoroughly put through their paces and its performance should continue to improve in the years ahead as adoption grows.

With the CDE in place, Chapter 4 turned fully to the combinatorial optimization aspects of 2D nesting problems. This led to `sparrow`, an open-source heuristic for the canonical 2D irregular strip packing problem. Built on `jagua-rs`, it not only redefines the state of the art but also provides a general baseline for future research.

Four years ago, researchers in 2D nesting had to rely on unverifiable numbers in tables, with little chance of reproducing prior results. Today, `sparrow` comes with a detailed statistical analysis, complete solution data, and the exact commands to replicate our experiments. Its open-source implementation enables others to build upon our work.

Most of `sparrow`'s components operate at the level of a feasibility problem, which makes adapting the approach to other problem variants such as bin-packing or knapsack problems relatively straightforward. In fact, a 3D adaptation of `sparrow` has been developed by Tollenaere [86]. Provided with only a draft version of the methodology, he was able to grasp the general ideas and principles behind the heuristic and resolve any ambiguities or missing details by inspecting snippets of the source code. His implementation is achieving very promising results for the 3D irregular strip packing problem.

Throughout Part II, committing to open-source from the outset brought an unexpected benefit: it made the writing process easier. Source code provides a reliable reference, thereby allowing the methodological description to focus on key ideas and overall structure of the method. Details that are essential for reproduction but not central to the big picture can be delegated to the implementation. Any ambiguities which will inevitably arise can also be clarified directly by inspecting the source code. This helps both writer and reader as the gist of the method is easier to grasp without drowning in details. In this way, papers and code serve complementary roles – the former communicates the concepts, the latter preserves the details.

This thesis marks a turning point in 2D nesting. We are confident that the contributions within Part II will have a lasting impact on the field and are convinced that there remains much more work to do.

Bibliography

[1] "ACM CCS 2025 Call for Papers". 2025. URL: https://www.sigsac.org/ccs/CCS2025/call-for-papers/.

[2] V. Agafonkin. "Polylabel: a fast algorithm for finding the pole of inaccessibility of a polygon". 2016. URL: https://github.com/mapbox/polylabel.

[3] B. Amaro Junior, P. R. Pinheiro, and P. V. Coelho. "A Parallel Biased Random-Key Genetic Algorithm with Multiple Populations Applied to Irregular Strip Packing Problems". In: *Mathematical problems in engineering* 2017.1 (2017), p. 1670709. DOI: 10.1155/2017/1670709.

[4] F. Antonio. "IV.6 - FASTER LINE SEGMENT INTERSECTION". In: *Graphics Gems III (IBM Version)*. Ed. by D. KIRK. San Francisco: Morgan Kaufmann, 1992, pp. 199–202. ISBN: 978-0-12-409673-8. DOI: 10.1016/B978-0-08-050755-2.50045-2.

[5] R. C. Art Jr. "An approach to the two dimensional irregular cutting stock problem." PhD thesis. Massachusetts Institute of Technology, 1966.

[6] M. Baker. "1,500 scientists lift the lid on reproducibility". In: *Nature* 533.7604 (May 2016), pp. 452–454. ISSN: 1476-4687. DOI: 10.1038/533452a.

[7] R. Baldacci, M. A. Boschetti, M. Ganovelli, and V. Maniezzo. "Algorithms for nesting with defects". In: *Discrete Applied Mathematics* 163 (2014). Matheuristics 2010, pp. 17–33. ISSN: 0166-218X. DOI: 10.1016/j.dam.2012.03.026.

[8] T. Barzola, V.-D. Cung, N. Gast, and V. Jost. "Experiments in Operations Research are Hardly Reproducible: A Bike-Sharing Case-Study." In: *23ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*. INSA Lyon. Villeurbanne - Lyon, France, Feb. 2022. URL: https://hal.science/hal-03595289.

[9] J. A. Bennell and X. Song. "A beam search implementation for the irregular shape packing problem". In: *Journal of Heuristics* 16 (2010), pp. 167–188. DOI: 10.1007/s10732-008-9095-x.

[10] J. A. Bennell and J. F. Oliveira. "The geometry of nesting problems: A tutorial". In: *European Journal of Operational Research* 184.2 (2008), pp. 397–415. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2006.11.038.

[11] Bentley and Ottmann. "Algorithms for Reporting and Counting Geometric Intersections". In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 643–647. DOI: 10.1109/TC.1979.1675432.

[12] J. Bergstra, D. Yamins, and D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by S. Dasgupta and D. McAllester. Vol. 28. Proceedings of Machine Learning Research 1. Atlanta, Georgia, USA: PMLR, 2013, pp. 115–123. URL: https://proceedings.mlr.press/v28/bergstra13.html.

[13] J. O. Berkey and P. Y. Wang. "Two-dimensional finite bin-packing algorithms". In: *Journal of the operational research society* 38.5 (1987), pp. 423–429. DOI: 10.1057/jors.1987.70.

[14] A. Bortfeldt. "A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces". In: *European Journal of Operational Research* 172.3 (2006), pp. 814–837. DOI: 10.1016/j.ejor.2004.11.016.

[15] J. L. Bresina. "Heuristic-biased stochastic sampling". In: AAAI'96 (1996), pp. 271–278.

[16] E. Burke, R. Hellier, G. Kendall, and G. Whitwell. "A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem". In: *Operations Research* 54.3 (2006), pp. 587–601. DOI: 10.1287/opre.1060.0293.

[17] E. K. Burke and Y. Bykov. "The late acceptance hill-climbing heuristic". In: *European Journal of Operational Research* 258.1 (2017), pp. 70–78. DOI: 10.1016/j.ejor.2016.07.012.

[18] C. Charalambous and K. Fleszar. "A constructive bin-oriented heuristic for the two-dimensional bin packing problem with guillotine cuts". In: *Computers and Operations Research* 38.10 (2011), pp. 1443–1451. ISSN: 03050548. DOI: 10.1016/j.cor.2010.12.013.

[19] J. Christiaens and G. Vanden Berghe. "Slack induction by string removals for vehicle routing problems". In: *Transportation Science* 54.2 (2020), pp. 417–433. DOI: 10.1287/trsc.2019.0914.

[20] F. Clautiaux, A. Jouglet, and A. Moukrim. “A new graph-theoretical model for the guillotine-cutting problem”. In: *INFORMS Journal on Computing* 25.1 (2013), pp. 72–86. DOI: 10.1287/ijoc.1110.0478.

[21] E. G. Coffman Jr, M. R. Garey, D. S. Johnson, and R. E. Tarjan. “Performance bounds for level-oriented two-dimensional packing algorithms”. In: *SIAM Journal on Computing* 9.4 (1980), pp. 808–826. DOI: 10.1137/0209062.

[22] E. Coffman Jr and P. Shor. “Average-case analysis of cutting and packing in two dimensions”. In: *European Journal of Operational Research* 44.2 (1990), pp. 134–144. DOI: 10.1137/0209062.

[23] C. Collberg and T. A. Proebsting. “Repeatability in computer systems research”. In: *Commun. ACM* 59.3 (Feb. 2016), pp. 62–69. ISSN: 0001-0782. DOI: 10.1145/2812803. URL: https://doi.org/10.1145/2812803.

[24] Y.-P. Cui, Y. Cui, and T. Tang. “Sequential heuristic for the two-dimensional bin-packing problem”. In: *European Journal of Operational Research* 240.1 (2015), pp. 43–53. ISSN: 03772217. DOI: 10.1016/j.ejor.2014.06.032.

[25] Y.-P. Cui, Y. Yao, and D. Zhang. “Applying triple-block patterns in solving the two-dimensional bin packing problem”. In: *Journal of the operational research society* 69.3 (2018), pp. 402–415. DOI: 10.1057/s41274-016-0148-5.

[26] D. H. Douglas and T. K. Peucker. “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature”. In: *Cartographica* 10.2 (1973), pp. 112–122. DOI: 10.3138/FM57-6770-U75U-7727.

[27] H. Dyckhoff. “A typology of cutting and packing problems”. In: *European Journal of Operational Research* 44.2 (1990), pp. 145–159. DOI: 10.1016/0377-2217(90)90350-K.

[28] J. Egeblad, B. K. Nielsen, and A. Odgaard. “Fast neighborhood search for two-and three-dimensional nesting problems”. In: *European Journal of Operational Research* 183.3 (2007), pp. 1249–1266. DOI: 10.1016/j.ejor.2005.11.063.

[29] A. Elkeran. “A new approach for sheet nesting problem using guided cuckoo search and pairwise clustering”. In: *European Journal of Operational Research* 231.3 (2013), pp. 757–769. DOI: 10.1016/j.ejor.2013.06.020.

[30] C. Ericson. *Real-time collision detection*. Elsevier, 2005. DOI: 10.1201/b14581.

[31] E. A. M. Fernandes, T. F. de Noronha, and A. A. Coco. "Heuristic Solutions for the 2D Bin-Packing Problem with Varied Size". In: *Brazilian Conference on Intelligent Systems*. Springer. 2024, pp. 65–80. DOI: 10.1007/978-3-031-79032-4_5.

[32] R. A. Finkel and J. L. Bentley. "Quad trees a data structure for retrieval on composite keys". In: *Acta informatica* 4 (1974), pp. 1–9. DOI: 10.1007/BF00288933.

[33] P. F. Fisher, S. Zhou, and C. B. Jones. "Shape-aware line generalisation with weighted effective area". In: *Developments in Spatial Data Handling: 11 th International Symposium on Spatial Data Handling*. Springer. 2005, pp. 369–380. DOI: 10.1007/3-540-26772-7_28.

[34] K. Fleszar. "Three insertion heuristics and a justification improvement heuristic for two-dimensional bin packing with guillotine cuts". In: *Computers & Operations Research* 40.1 (2013), pp. 463–474. DOI: 10.1016/j.cor.2012.07.016.

[35] F. Fontan. "packingsolver: A solver for (geometrical) packing problems". 2023. URL: https://github.com/fontanf/packingsolver.

[36] R. J. Fowler, M. S. Paterson, and S. L. Tanimoto. "Optimal packing and covering in the plane are NP-complete". In: *Information processing letters* 12.3 (1981), pp. 133–137. DOI: 10.1016/0020-0190(81)90111-3.

[37] D. K. Friesen and M. A. Langston. "Variable sized bin packing". In: *SIAM journal on computing* 15.1 (1986), pp. 222–230. DOI: 10.1137/0215016.

[38] J. Gardeyn. "`gdrr-2bp`: a Rust implementation of the goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints." 2022. URL: https://github.com/JeroenGar/gdrr-2bp.

[39] J. Gardeyn. "`jagua-rs`: a fast and fearless Collision Detection Engine for 2D nesting problems". 2024. URL: https://github.com/JeroenGar/jagua-rs.

[40] J. Gardeyn. "`sparrow`: state-of-the-art nesting for 2D irregular strip packing". 2025. URL: https://github.com/JeroenGar/sparrow.

[41] J. Gardeyn, G. Vanden Berghe, and T. Wauters. "An open-source heuristic to reboot 2D nesting research". In: *arXiv [preprint]* (2025). DOI: 10.48550/arXiv.2509.13329.

[42] J. Gardeyn, G. Vanden Berghe, and T. Wauters. "Decoupling Geometry from Optimization in 2D Irregular Cutting and Packing Problems: an Open-Source Collision Detection Engine". In: *INFORMS Journal on Computing* (2025). article in advance. DOI: 10.1287/ijoc.2024.1025.

[43] J. Gardeyn and T. Wauters. “A goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints”. In: *European Journal of Operational Research* 301.2 (2022), pp. 432–444. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2021.11.031.

[44] P. Gilmore and R. E. Gomory. “Multistage cutting stock problems of two and more dimensions”. In: *Operations research* 13.1 (1965), pp. 94–120. DOI: 10.1287/opre.13.1.94.

[45] F. Glover. “Tabu search—part I”. In: *ORSA Journal on computing* 1.3 (1989), pp. 190–206. DOI: 10.1287/ijoc.1.3.190.

[46] A. M. Gomes and J. F. Oliveira. “A 2-exchange heuristic for nesting problems”. In: *European Journal of Operational Research* 141.2 (2002), pp. 359–370. DOI: 10.1016/S0377-2217(02)00130-3.

[47] R. M. Haralick and G. L. Elliott. “Increasing tree search efficiency for constraint satisfaction problems”. In: *Artificial intelligence* 14.3 (1980), pp. 263–313. DOI: 10.1016/0004-3702(80)90051-X.

[48] B. Heisler and J. Aparicio. “Criterion.rs: Statistics-driven benchmarking library for Rust”. 2014. URL: https://github.com/bheisler/criterion.rs.

[49] J. Heistermann and T. Lengauer. “The nesting problem in the leather manufacturing industry”. In: *Annals of Operations Research* 57.1 (1995), pp. 147–173. DOI: 10.1007/BF02099695.

[50] S. Hong, D. Zhang, H. C. Lau, X. Zeng, and Y. W. Si. “A hybrid heuristic algorithm for the 2D variable-sized bin packing problem”. In: *European Journal of Operational Research* 238.1 (Oct. 2014), pp. 95–103. ISSN: 03772217. DOI: 10.1016/j.ejor.2014.03.049.

[51] E. Hopper and B. Turton. “Problem generators for rectangular packing problems.” In: *Stud. Inform. Univ.* 2.1 (2002), pp. 123–136.

[52] T. Imamichi, M. Yagiura, and H. Nagamochi. “An iterated local search algorithm based on nonlinear programming for the irregular strip packing problem”. In: *Discrete Optimization* 6.4 (2009), pp. 345–361. DOI: 10.1016/j.disopt.2009.04.002.

[53] M. Iori, V. L. de Lima, S. Martello, F. K. Miyazawa, and M. Monaci. “Exact solution techniques for two-dimensional cutting and packing”. In: *European Journal of Operational Research* 289.2 (2021), pp. 399–415. DOI: 10.1016/j.ejor.2020.06.050.

[54] L. V. Kantorovich. “Mathematical methods of organizing and planning production”. In: *Management science* 6.4 (1960), pp. 366–422. DOI: 10.1287/mnsc.6.4.366.

[55] G. Kendall, R. Bai, J. Błazewicz, P. D. Causmaecker, M. Gendreau, R. John, J. Li, B. McCollum, E. Pesch, R. Qu, N. Sabar, G. V. Berghe, and A. Yee. "Good Laboratory Practice for optimization research". In: *Journal of the Operational Research Society* 67.4 (2016), pp. 676–689. DOI: 10.1057/jors.2015.77.

[56] B. Kröger. "Guillotineable bin packing: A genetic approach". In: *European Journal of Operational Research* 84.3 (1995), pp. 645–661. DOI: 10.1016/0377-2217(95)00029-P.

[57] C. Lallier, L. Vézard, B. Pinaud, and G. Blin. *Nesting Tasks Dataset for 2D-Nesting Efficiency Estimation*. Version 1.1. Zenodo, May 2022. DOI: 10.5281/zenodo.7030786.

[58] J. J. Lastra-Díaz and M. T. Ortuño. "Mixed-integer programming models for irregular strip packing based on vertical slices and feasibility cuts". In: *European Journal of Operational Research* 313.1 (2024), pp. 69–91. DOI: 10.1016/j.ejor.2023.08.009.

[59] T. V. Le, L. A. Albert, and T. Vidal. "Making Operations Research More Accessible: Insights from the Rise of Machine Learning". In: *INFORMS Journal on Data Science* (2025). DOI: 10.1287/ijds.2025.0076.

[60] A. A. Leao, F. M. Toledo, J. F. Oliveira, M. A. Carravilla, and R. Alvarez-Valdés. "Irregular packing problems: A review of mathematical models". In: *European Journal of Operational Research* 282.3 (2020), pp. 803–822. DOI: 10.1016/j.ejor.2019.04.045.

[61] S. C. Leung, Y. Lin, and D. Zhang. "Extended local search algorithm based on nonlinear programming for two-dimensional irregular strip packing problem". In: *Computers & Operations Research* 39.3 (2012), pp. 678–686. DOI: 10.1016/j.cor.2011.05.025.

[62] A. Lodi, S. Martello, and M. Monaci. "Two-dimensional packing problems: A survey". In: *European journal of operational research* 141.2 (2002), pp. 241–252. DOI: 10.1016/S0377-2217(02)00123-6.

[63] A. Lodi, S. Martello, and D. Vigo. "Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems". In: *INFORMS Journal on Computing* 11.4 (1999), pp. 345–357. DOI: 10.1287/ijoc.11.4.345.

[64] A. Lodi, S. Martello, and D. Vigo. "Recent advances on two-dimensional bin packing problems". In: *Discrete Applied Mathematics* 123.1-3 (2002), pp. 379–396. DOI: 10.1016/S0166-218X(01)00347-X.

[65] A. Lodi, M. Monaci, and E. Pietrobuoni. "Partial enumeration algorithms for Two-Dimensional Bin Packing Problem with guillotine constraints". In: *Discrete Applied Mathematics* 217 (2017), pp. 40–47. ISSN: 0166218X. DOI: 10.1016/j.dam.2015.09.012.

[66] I. Loshchilov, M. Schoenauer, and M. Sebag. "Adaptive coordinate descent". In: *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 2011, pp. 885–892. DOI: 10.1145/2001576.2001697.

[67] H. Ma and C.-C. Liu. "Fast nesting of 2-D sheet parts with arbitrary shapes using a greedy method and semi-discrete representations". In: *IEEE Transactions on Automation Science and Engineering* 4.2 (2007), pp. 273–282. DOI: 10.1109/TASE.2006.874973.

[68] S. Martello, M. Monaci, and D. Vigo. "An exact approach to the strip-packing problem". In: *INFORMS journal on Computing* 15.3 (2003), pp. 310–319. DOI: 10.1287/ijoc.15.3.310.16082.

[69] T. Mészáros. "libnest2d". 2018. URL: https://github.com/tamasmeszaros/libnest2d.

[70] "NDSS Symposium 2026 Call for Papers". 2025. URL: https://www.ndss-symposium.org/ndss2026/submissions/call-for-papers/.

[71] S. Nestler. "Reproducible (operations) research". In: (2011). URL: https://hdl.handle.net/10945/47788.

[72] N. Ntene and J. H. van Vuuren. "A survey and comparison of guillotine heuristics for the 2D oriented offline strip packing problem". In: *Discrete Optimization* 6.2 (2009), pp. 174–188. DOI: 10.1016/j.disopt.2008.11.002.

[73] R. Oberdieck. "Reproducibility in Operations Research". 2018. URL: https://www.linkedin.com/pulse/reproducibility-operations-research-richard-oberdieck/.

[74] J. F. Oliveira, A. M. Gomes, and J. S. Ferreira. "TOPOS–A new constructive algorithm for nesting problems". In: *OR-Spektrum* 22 (2000), pp. 263–284. DOI: 10.1007/s002910050105.

[75] O. Oliveira, D. Gamboa, and E. Silva. "Resources for Two-Dimensional (and Three-Dimensional) Cutting and Packing Solution Methods Research". In: *In Proceedings of the 16th International Conference on Applied Computing* 53 (2019). https://github.com/Oscar-Oliveira/OR-Datasets, pp. 131–138. DOI: 10.33965/ac2019_201912L016.

[76] F. G. Ortmann, N. Ntene, and J. H. Van Vuuren. "New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems". In: *European Journal of Operational Research* 203.2 (2010), pp. 306–315. DOI: 10.1016/j.ejor.2009.07.024.

[77] F. Petropoulos et al. "Operational Research: methods and applications". In: *Journal of the Operational Research Society* 75.3 (2024), pp. 423–617. DOI: 10.1080/01605682.2023.2253852.

[78] D. Pisinger and M. Sigurd. “The two-dimensional bin packing problem with variable bin sizes and costs”. In: *Discrete Optimization* 2.2 (2005), pp. 154–167. DOI: 10.1016/j.disopt.2005.01.002.

[79] S. Polyakovsky and R. M’Hallah. “An agent-based approach to the two-dimensional guillotine bin packing problem”. In: *European Journal of Operational Research* 192.3 (2009), pp. 767–781. DOI: 10.1016/j.ejor.2007.10.020.

[80] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Texts and monographs in computer science. Springer-Verlag, 1985. DOI: 10.1007/978-1-4612-1098-6.

[81] P. Rocha, R. Rodrigues, F. M. Toledo, and A. M. Gomes. “Circle covering using medial axis”. In: *IFAC Proceedings Volumes* 46.7 (2013), pp. 402–407. DOI: 10.3182/20130522-3-BR-4036.00081.

[82] A. K. Sato, T. C. Martins, A. M. Gomes, and M. S. G. Tsuzuki. “Raster penetration map applied to the irregular packing problem”. In: *European Journal of Operational Research* 279.2 (2019), pp. 657–671. DOI: 10.1016/j.ejor.2019.06.008.

[83] S. A. Segenreich and L. M. P. F. Braga. “Optimal nesting of general plane figures: A Monte Carlo heuristical approach”. In: *Computers & Graphics* 10.3 (1986), pp. 229–237. DOI: 10.1016/0097-8493(86)90007-5.

[84] M. Shimrat. “Algorithm 112: position of point relative to polygon”. In: *Communications of the ACM* 5.8 (1962), p. 434. DOI: 10.1145/368637.368653.

[85] E. Shusterman and M. Feder. “Image compression via improved quadtree decomposition algorithms”. In: *IEEE Transactions on Image Processing* 3.2 (1994), pp. 207–215. DOI: 10.1109/83.277901.

[86] J. Tollenaere. “`sparrow-3d`”. 2025. URL: https://github.com/JonasTollenaere/sparrow-3d.

[87] S. Umetani, M. Yagiura, S. Imahori, T. Imamichi, K. Nonobe, and T. Ibaraki. “Solving the irregular strip packing problem via guided local search for overlap minimization”. In: *International Transactions in Operational Research* 16.6 (2009), pp. 661–683. DOI: 10.1111/j.1475-3995.2009.00707.x.

[88] UNESCO. “UNESCO Recommendation on Open Science”. In: (2021). DOI: 10.54677/MNMH8546.

[89] “USENIX Security ’26 Call for Papers”. 2025. URL: https://www.usenix.org/conference/usenixsecurity26/call-for-papers.

[90] T. Vidal. “Hybrid genetic search for the CVRP: Open-source implementation and SWAP* neighborhood”. In: *Computers & Operations Research* 140 (2022), p. 105643. DOI: 10.1016/j.cor.2021.105643.

[91] M. Visvalingam and J. D. Whyatt. “Line generalisation by repeated elimination of points”. In: *The cartographic journal* 30.1 (1993), pp. 46–51. DOI: 10.1179/000870493786962263.

[92] C. Voudouris, E. P. Tsang, and A. Alsheddy. “Guided local search”. In: *Handbook of metaheuristics*. Springer, 2010, pp. 321–361. DOI: 10.1007/978-1-4419-1665-5_11.

[93] Y. Wang, R. Xiao, and H. Wang. “A flexible labour division approach to the polygon packing problem based on space allocation”. In: *International journal of production research* 55.11 (2017), pp. 3025–3045. DOI: 10.1080/00207543.2016.1229070.

[94] Z. Wang and J.-C. Müller. “Line generalization based on analysis of shape characteristics”. In: *Cartography and Geographic Information Systems* 25.1 (1998), pp. 3–15. DOI: 10.1559/152304098782441750.

[95] G. Wäscher, H. Haußner, and H. Schumann. “An improved typology of cutting and packing problems”. In: *European Journal of Operational Research* 183.3 (2007), pp. 1109–1130. ISSN: 03772217. DOI: 10.1016/j.ejor.2005.12.047.

[96] L. Wei, W.-C. Oon, W. Zhu, and A. Lim. “A goal-driven approach to the 2D bin packing and variable-sized bin packing problems”. In: *European Journal of Operational Research* 224.1 (2013), pp. 110–121. DOI: 10.1016/j.ejor.2012.08.005.

Valorization

The Faculty Board of Engineering Technology requires the inclusion of a valorization plan that explores how the research conducted in this PhD could be converted into value, either in societal or economic terms.

Part I of this thesis stems from a two-year collaboration with ARDIS[1] within a VLAIO[2] project (code: `HBC.2018.2297`). ARDIS is a global market leader in optimization & manufacturing software for sheet cutting. They serve a large set of customers with diverse profiles – from small kitchen manufacturers to large steel processing companies – requiring robust and flexible optimization solutions. The role of KU Leuven within this project was to design an algorithm to serve as the basis for a new commercial solver for 2D rectangular C&P problems. The algorithms had to be flexible enough to accommodate a broad range of industrial constraints while delivering consistent performance across these diverse contexts. At the end of the two-year project, knowledge about the methods and codebase was fully transferred to ARDIS' software engineering team. In the years since, development has continued within the company, and we remain in contact through sporadic meetings to discuss new ideas and improvements.

[1] https://www.ardis.eu/en-US/home
[2] Flanders Innovation & Entrepreneurship

Part II of this thesis was carried out under a four-year *strategic basic research* grant from FWO[3] (codes: `1S71222N`, `1S71224N`). The objective was to tackle 2D irregular C&P problems, which have industrial applications ranging from apparel manufacturing to shipbuilding. The irregular nature of the items adds significant geometric challenges on top of an already difficult combinatorial optimization problem. Before our work, there was virtually no reusable, publicly available code to tackle such problems or to build upon prior work. Within this grant, we developed both the `jagua-rs` library introduced in Chapter 3 and the `sparrow` algorithm introduced in Chapter 4. We invested significant effort to ensure that `jagua-rs` is a reusable, flexible library, which is well-documented and tested. In fact, a 3D adaptation of `sparrow` has been developed by Tollenaere [86]. Open-sourcing this software significantly increases the "reach" of the work, and both contributions have already gained notable traction around the globe. The code repositories associated with this thesis have accumulated about 200 stars on GitHub, and external developers have already contributed. We aim for these contributions to create value both by serving as a solid foundation for continued research in 2D nesting algorithms and by being directly adopted and used in industrial contexts.

Throughout the entire research process of this thesis, creating long-lasting value has been a priority instead of an afterthought to be addressed at the end. The algorithms and software developed in each of the three main chapters are open source, well-documented, and designed with a strong emphasis on flexibility and simplicity. Valorization was a priority from the start of this PhD and is already in motion.

[3]Research Foundation – Flanders

Statement on the use of Generative AI

In preparing this work, I used generative AI tools to suggest refinement to the flow and clarity of my own writing, as well as to proofread grammar and spelling. These tools were also leveraged to brainstorm about different phrasing options, synonyms that convey a particular nuance, and to find the most suitable names for abstractions and concepts. Additionally, I made extensive use of AI-assisted code completion tools when writing code.
The text, code, and images in this thesis are my own (unless otherwise specified). Generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

List of Publications

Journal Articles

J. Gardeyn and T. Wauters. "A goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints". In: *European Journal of Operational Research* 301.2 (2022), pp. 432–444. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2021.11.031

J. Gardeyn, G. Vanden Berghe, and T. Wauters. "Decoupling Geometry from Optimization in 2D Irregular Cutting and Packing Problems: an Open-Source Collision Detection Engine". In: *INFORMS Journal on Computing* (2025). article in advance. DOI: 10.1287/ijoc.2024.1025

Preprints Submitted for Publication

J. Gardeyn, G. Vanden Berghe, and T. Wauters. "An open-source heuristic to reboot 2D nesting research". In: *arXiv [preprint]* (2025). DOI: 10.48550/arXiv.2509.13329

Open Source Software

J. Gardeyn. "`gdrr-2bp`: a Rust implementation of the goal-driven ruin and recreate heuristic for the 2D variable-sized bin packing problem with guillotine constraints." 2022. URL: https://github.com/JeroenGar/gdrr-2bp

J. Gardeyn. "`jagua-rs`: a fast and fearless Collision Detection Engine for 2D nesting problems". 2024. URL: https://github.com/JeroenGar/jagua-rs

J. Gardeyn. "`sparrow`: state-of-the-art nesting for 2D irregular strip packing". 2025. URL: https://github.com/JeroenGar/sparrow

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
Celestijnenlaan 200A box 2402
B-3001 Leuven

