UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Squaring the Square: Parallelising and Refining the Exhaustive Algorithm

Anastasia Courtney

King's College

June 2023

Submitted in partial fulfillment of the requirements for the
Computer Science Tripos, Part III

Total page count: 46

Main chapters (excluding front-matter, references and appendix): 42 pages (pp 1–42)

Main chapters word count: 11166

Methodology used to generate that word count:

```
$ texcount -inc -total -sum=1,1,1,1,1,1,1 main.tex
```

Declaration

Abstract

Squaring the Square describes the problem of decomposing a square into smaller, unique, integer squares [1]. **Alan Turing** discussed the problem in his cornerstone paper *Automatic Computing Engine* [2] as an example of what electronic computers could achieve.

In this project, we build upon **Ian Gambini**'s work to enumerate squared squares and rectangles by their width, seeking new results or new decompositions [3, 4].

We present a fundamentally restructured algorithm, with **exponentially reduced complexity**, that we used to enumerate all squared squares and rectangles up to and including width 150. In this search, We placed 138,550,916,945,954 squares in 263 hours, to produce many new results. These include most notably the ninth ever triple isomer, which is the smallest possible triple isomer, and the first isomer of its order. This set of decompositions could be considered one of the most significant discoveries in the field since Gambini's work in 1999.

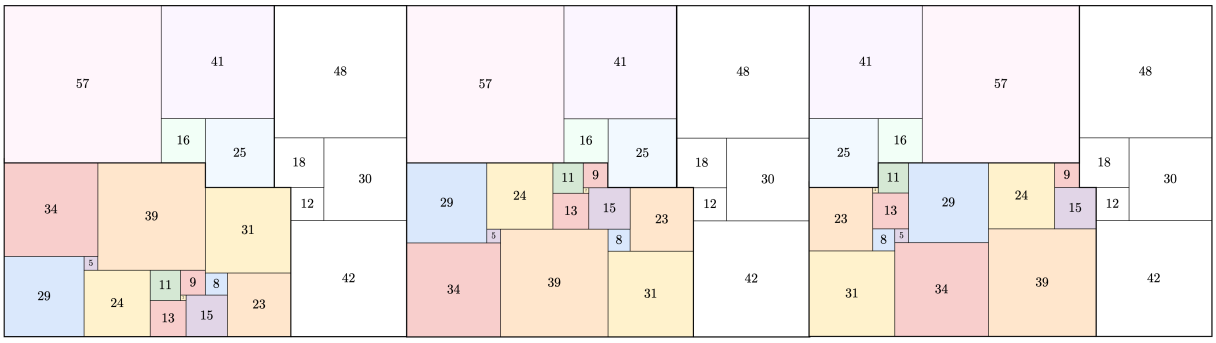


Figure 1: *The triple isomer discovered in this report, which is also the smallest possible triple, and the first identified of order 22, and the ninth to ever be discovered.*

The performance of the algorithm, compared on the same machine to account for technological advancements, was improved by a factor of 475x in the space enumerated, with increasing returns as the algorithm progresses. Additionally, we present novel proofs for new bounds which can be exploited in square decomposition, and methods of constructing decompositions with only five squares on their boundary. Finally, we suggest multiple avenues of future research.

Acknowledgements

This project, and the discoveries in it, are not solely my own. I am immensely fortunate to be surrounded by an extraordinary set of friends and colleagues whose encouragement and belief in me has made them possible.

To my friends, who patiently and enthusiastically endured my thoughts about squares, and provided much-needed support throughout the development and writing process.

To Professor Larry Paulson FRS, whose faith in my academic curiosity allowed me to pursue projects I was truly excited about. I hope the outcomes of this project validate your trust in that excitement, and that you find delight in reading this paper.

To Professor Tim Griffin, whose unwavering support throughout my time in Cambridge empowered me to produce work I can be proud of.

To my godchildren, Verity and David, who inspire me daily to pursue curiosity and joy.

Special thanks to Margaret, Martin, and Mikel. You have been a pillar of strength and supoort, and my gratitude for you is indescribable.

Contents

Chapter 1

Introduction

Squaring the Square describes the problem of decomposing a square into smaller, unique, integer squares. A decomposition made of unique constituent squares is called a *perfect squared square*, or *perfect squared rectangle*. A decomposition that contains a smaller squared rectangle is considered *compound*, and one that does not is called *simple*. The *order* of a decomposition is how many squares it is constructed from [1].

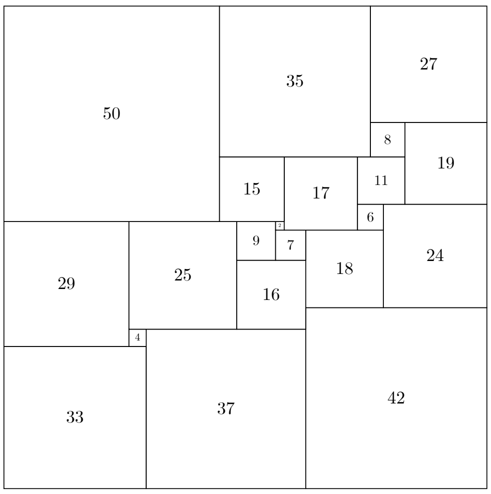


Figure 1.1: *21 : 112A AJD 1978, the smallest order squared square.*

Many squared squares and rectangles have been discovered, and most methods of decomposition are either constructive or exhaust solutions by order. **Ian Gambini's** work exhausted squared squares and rectangles by *size* [3], referring to the width of the solutions, and was considered pivotal in the field for proving that the smallest possible squared squares are those of width 110.

Figure 1.1 shows the squared square the Trinity Mathematical Society's logo is based on, discovered in 1978 by **A.J.W. Duijvestijn**, which is of the smallest order: 21 constituent squares.

In this project, we build on Gambini's work to enumerate perfect rectangles and squares up to a much larger size, seeking decompositions of interest; these include the smallest decompositions of various orders, any with particularly notable features, and isomer sets.

To do so, we leverage parallelism and mathematical bounds, as well as fundamentally change Gambini's exhaustive algorithm.

1.1 Motivation

1.1.1 History

Since its conceptualisation, *Squaring the Square* has captured the interest of many mathematicians and has primarily served as a purely academic pursuit. It has a colourful history as a seemingly straightforward puzzle, but even so, has possible real-world applications.

Henry Dudeney, a writer of recreational mathematics, was the first to present the problem of square decomposition, in 1902. He described dissecting a square into distinct squares *and a single rectangle*:

When young men sued for the hand of Lady Isabel, Sir Hugh promised his consent to the one who would tell him the dimensions of the top of the box from these facts alone:

There was a rectangular strip of gold, ten inches by $\frac{1}{4}$-inch; and the rest of the surface was exactly inlaid with pieces of wood, each piece being a perfect square, and no two pieces of the same size.

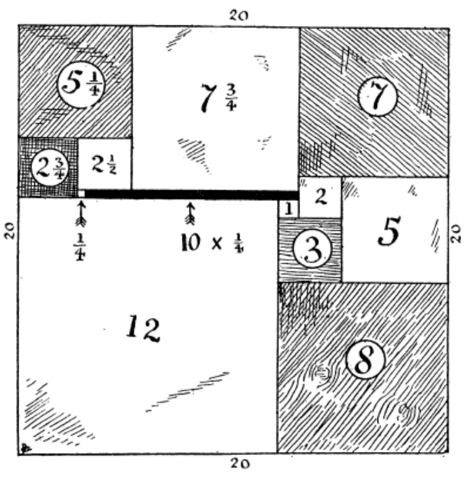


Figure 1.2: *The Lady Isabel's Casket puzzle, and its solution [5].*

Only a year later, in 1903, **Max Dehn** published a proof that any *rectangle* could be perfectly squared if and only if the ratio of its side lengths was a rational number. It was later included as one of 100 proofs in *Proofs from the Book* [6], the 1988 book dedicated to Paul Erdős' famous idea of The Book, in which God keeps the most elegant proof of each mathematical theorem.

By the 1930s, square decomposition was being researched globally. **Michio Abe** published *Covering the square by squares without overlapping* [7] in the Journal of Japan Mathematical Physics in 1930. Though the title describes *Squaring the Square* prior to it being named such, Abe did not produce a squared square, instead detailing a finite constructive procedure for squaring rectangles where the ratio of the sides was arbitrarily close to 1.

The phrase "Square the Square" originates from an article published by **Tutte et al** in the late 1930s: *The Dissection of Rectangles into Squares* [8]. Tutte et al represented Squared Rectangles as electrical circuits, and by demonstrating that the circuits obeyed Kirchoff's circuit current laws, built upon research in this area to find what is widely thought to be the first Squared Square – despite **Roland Sprague** beating them to publication, presenting the first Squared Square without detailing how it was found [9].

The problem of Squaring the Square is even referenced in **Alan Turing**'s revolutionary 1945 paper *Proposed Electronic Calculator* (later known as *Automatic Computing Engine*) [2]. It is presented as a non-numerical problem the 'electronic computers' could achieve. We now know that Turing and Tutte spoke about the problem during their shared time at Bletchley Park [10].

Plenty of mathematicians have continued to research *Squaring the Square* since the 1930s: the enumeration of Squared Squares is an ongoing pursuit, the latest contribution having been made in 2021 [11]. The site `squaring.net`, built and maintained by **Stuart Anderson** contains extensive catalogues of known decompositions, history, and details regarding interesting discoveries and open areas of interest. Anderson has also made many contributions directly to the field, enumerating all squared rectangles up to order 24, publishing those up to 21 along with notable discoveries within these orders [1].

1.1.2 Real-world Applications

Michio Abe's 1930 paper *Covering the square by squares without overlapping* [7] has been cited by several publications on packing and floorplanning in VLSI placement design [12, 13].

Despite being easily explainable to children, Squaring the Square is a deceptively difficult puzzle that only the most dedicated mathematicians have been able to crack. Outside of its contributions to silicon floorplanning, the problem has contributed to mathematics as an academic curiosity: it was shared Scientific American magazine in in 1958 in **Martin Gardner**'s column, who is cited as the reason many mathematicians chose to pursue mathematics [14]. Gardner returned to square tilings and squared squares many times in his columns and books before retiring, reporting on important discoverings made by both adademic and hobbyist mathematiciansn.

1.2 Literature Review

1.2.1 Gambini

In 1999, Ian Gambini published *A Method for Cutting Squares into Distinct Squares* [3]. In this paper, Gambini presents two methods for Squaring the Square, diverging from the classical models of circuits and graphs. Gambini modelled the intermediate stages of Squaring the Square using plates, placing squares on top of one another in a search. The first method fixed the number of plates and could be used to produce a great number of previously unseen squares, **the second was exhaustive** and proved definitively that Duijvestijn & Willcocks' 1978 discovery of a squared square with side 110 was the **smallest**

squared square possible.

Using his second algorithm, Gambini not only proved that the minimum side length of a Squared Square was 110, he proved none existed for side length 111, and that the smallest side length for a squared square with the fewest possible constituent squares (21 squares) is 112. However, **continuing the search was computationally prohibitive**, and, as a result, no further side-lengths have been provably exhausted. In this project, I fundamentally restructure Gambini's algorithm, and exploit parallelism as well as various proof-based optimisations to definitively enumerate a much larger set of squared squares and rectangles by size.

Gambini's thesis *Quant aux carrés carrelés* [4], published in French, contains various proofs regarding limitations particularly on the boundary squares of a squared square or rectangle. These are presumed to have been used to improve the running time of Gambini's algorithms.

1.2.2 Anderson

Stuart Anderson has enumerated all simple perfect squared rectangles between orders 9 and 24 [1] – due to the exponential increase in the volume of decompositions of larger orders, Anderson has not published the decompositions above order 21.

Anderson's work enumerating these orders also extended to identifying *isomers* – sets of decompositions that contain the same squares rearranged – up to order 21 [15]. From the work published, it is unclear whether anyone has *inspected* the orders above 21 due to the large number of decompositions. Through reviewing the available literature, we could find no evidence of any results that have been drawn from these orders beyond the number of decompositions found [1, 15, 16].

Additionally, Anderson built and maintains `squaring.net`, which contains extensive catalogues of known decompositions, history and research.

1.2.3 Moew

David Moew has published codes of all simple perfect squared rectangles of orders 9 to 16, all perfect squared squares of orders 21 to 24, and *some* of orders 25 to 27, available at `djm.cc/dmoews.html` [16].

1.3 Starting Point

No previous work undertaken in tiling by myself aided this project.

Many **orders** of squares and rectangles have been completely enumerated. Higher orders, which have exponentially more squared squares and rectangles, have yet to be exhausted, leaving gaps in our knowledge – beyond Gambini's work up to 112, existing research has only enumerated sets of specific orders, so unless properties (such as number of squares on the boundary) are exhausted, these methods cannot prove that a certain rectangle or square is the smallest with that given property.

Gambini used exhaustive search by size to show that the smallest squared square was ADJ's 110 of order 22, and that the smallest square of the smallest order were the isomer pair size 112 order 21 [3].

Isomer sets of rectangles such as pairs and triples have been enumerated up to order 21 [15]. While hundreds of millions of decompositions have been enumerated, only a handful of isomer sets have been discovered: hundreds of pairs, and only eight triples [1, 15].

The algorithm presented in this paper seeks to extend this exhaustive search by size, through which we sought new squared squares and rectangles, and to prove which decompositions are the smallest with certain properties such as order, number of boundary squares, and isometry. The results exceeded these expectations, and are discussed in section 6.2.

1.4 Report Structure

In this report, we have first explored the background of the problem, the starting point, and the motivation for pursuing it. In **Theory** (chapter 2), we explain the theory and definitions necessary for understanding the report.

Core Implementation (chapter 3) describes the implementation of the algorithm: this includes the three-step system we use to produce solutions (set, solve, process), and work done to adapt Gambini's work into a new algorithm. Parallelisation (chapter 4) explains the parallelisation of the core algorithm.

Optimisations and Analysis (chapter 5) details each optimisation I implemented and evaluates the extent to which these improve the runtime of the algorithm. **Results** (chapter 6) summarises the improvements made to the algorithm, and presents the results of the work done, including new decompositions, and the details of particularly notable discoveries.

Conclusions (chapter 7) summarises the report and results, and presents future work that could build on the work done.

Chapter 2

Theory

Here, we look at the key definitions, including those that are standard in the field, those presented by Gambini, and those introduced by this report. Additionally, we detail some bounds on squared squares and rectangles that justify the correctness of our optimisations.

2.1 Definitions

2.1.1 Tiling Definitions

Squared Rectangle A *squared rectangle* is a decomposition of a rectangle into a finite number of at least two squares.

Squared Square A *squared square* is a squared rectangle that is itself a square.

Perfect A decomposition is considered *perfect* when all of its constituent squares have distinct sizes.

Order The *order* of a decomposition is the number of squares.

Compound a *compound* decomposition is one in which a subset of the elements constitutes a rectangle.

Simple a *simple* decomposition which does not contain a subrectangle (ie. is not compound).

SPSR, CPSR, SPSS, CPSS Shown in Figure 2.1 is an example CPSR (compound perfect squared rectangles), SPSR (simple perfect squared rectangles), CPSS (compound perfect squared squares), and SPSS (simple perfect squared squares). In this project, we only consider decompositions that are perfect.

Classes Any decomposition can be trivially transformed into at least four isomorphisms of itself (via rotation and reflection), creating an isomorphism class of equivalent decompositions [1], called the **SPSS class** or **SPSR class**. Compound decompositions have a further **CPSS isomer class** composed of the isomorphisms produced by rotating and reflecting the smaller squared rectangles within them.

A single decomposition stands for the entirety of its class or isomer class. The canonical isomer that represents the class is the one which has the largest corner square in the upper left corner – for squares, the additional constraint is added that the larger of the two squares adjacent to that corner square is on the top. This selection is nested for compound decompositions. The four decompositions in Figure 2.1 are the representative arrangements for their class or isomer class respectively.

Isomer Pairs, Triples, Quadruples... Where a decomposition can be rearranged non-trivially into another distinct decomposition, they form [pairs, triples, quadruples...] of isomers. Noted in section 1.3, these are **exceedingly rare**: while hundreds of pairs have been identified among millions of decompositions, only seven triple isomer SPSRs have been found, and one triple isomer SPSS has been found.

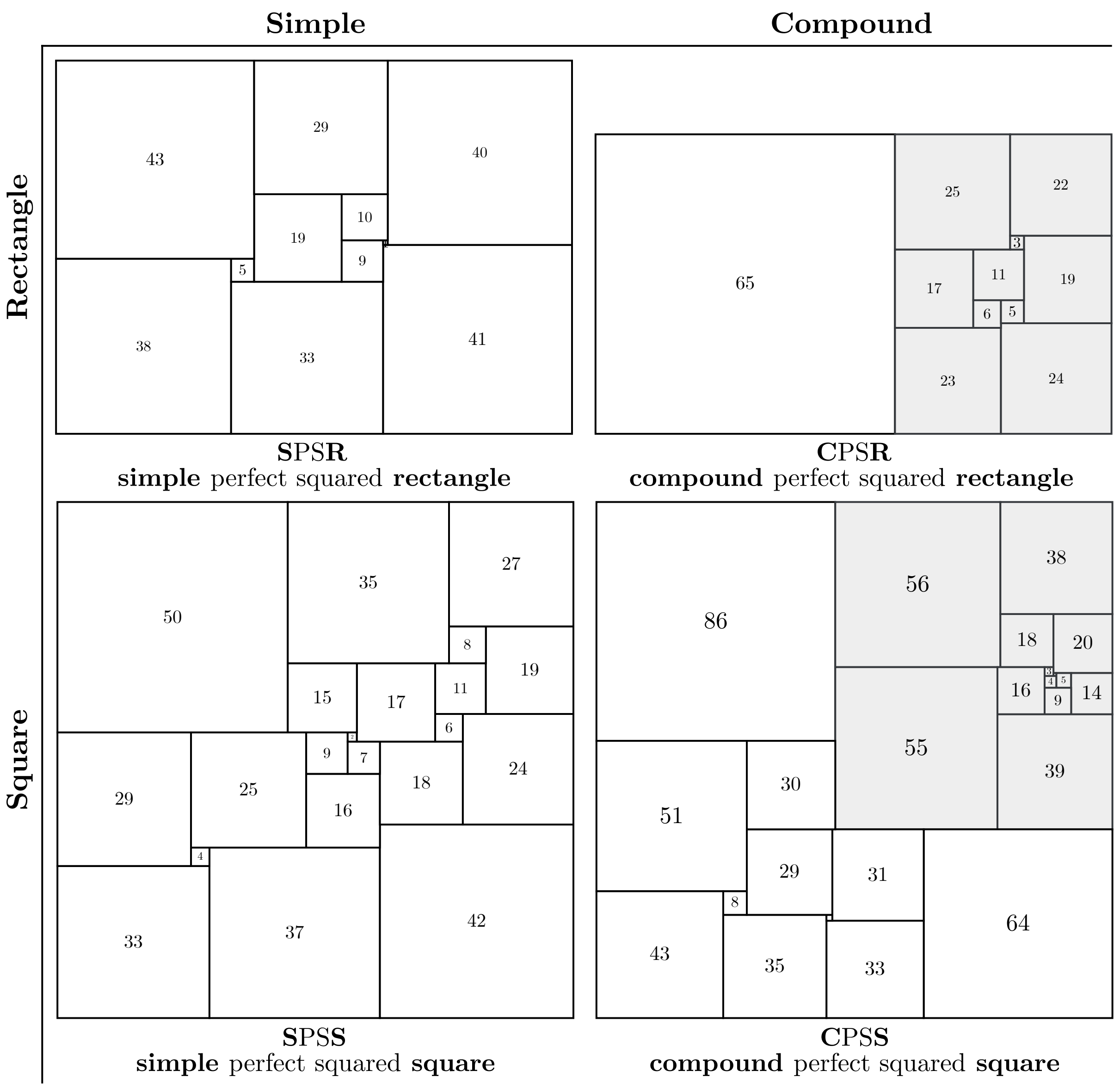

Simple
Compound
Rectangle
Square
SPSR
simple perfect squared rectangle
CPSR
compound perfect squared rectangle
SPSS
simple perfect squared square
CPSS
compound perfect squared square

Figure 2.1: *Table demonstrating the naming convention of* **[simple/compound]** perfect squared **[squares/rectangles]**.

2.1.2 Gambini's Definitions

Ian Gambini's algorithm introduces an intermediate representation for decomposition, and relies on placing squares from the bottom up [3]. The top of the filled space is represented as a series of *plates*, such as in Figure 2.2. The first and last plate delimit the total size of the target square.

Plate Each plate, which represents the layer on top of so-far placed squares, has an index, width, and height.

Delimited [plate] A delimited plate is one whose neighbours are taller than itself. In 2.2, {2: 28, 33} is a delimited plate.

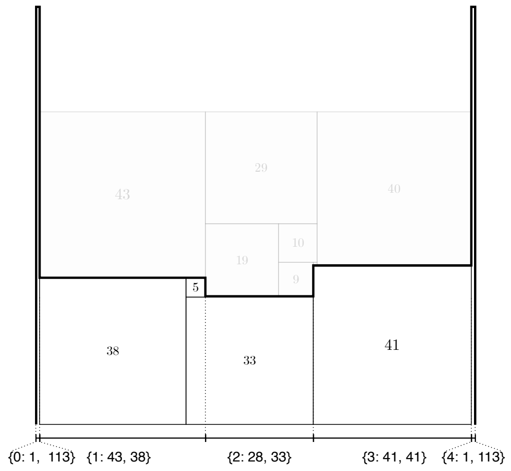


Figure 2.2: *An example of the plates in the incomplete construction of an SPSR, after the placement of 4 plates.*

2.1.3 New Definition: Gambinicode+

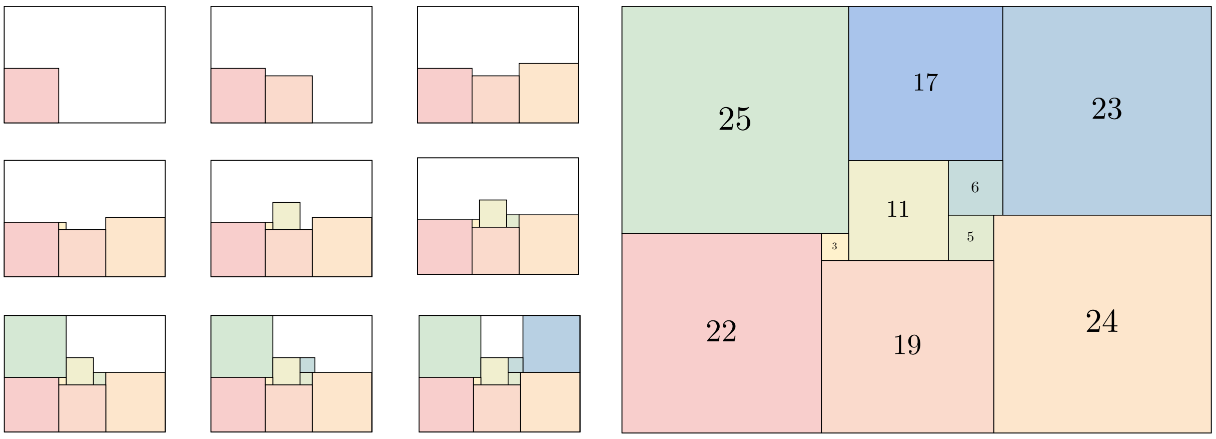
In the field of Squaring, the canonical encoding is called **Bouwkampcode** or **Tablecode**. Here, I define **Gambinicode+**, an alternative encoding for squared rectangles and squares, which is a natural extension of the algorithm presented by Gambini. Given the intuitive and deterministic nature of Gambini's algorithm, one can quickly recover a PSR based only on the order in which the squares were placed, and the width of the solution.

Notably, the algorithm we develop from Gambini's algorithm, detailed in section 3.3, modifies the order of square placement. In this report, we define Gambinicode+ to **include** this modification. Although this modification is a significant change to the algorithm, I have chosen to name the encoding Gambinicode+ in recognition that the core ideas and structure of the algorithm are still Gambini's, with a '+' to indicate the departure from the original algorithm, to avoid confusion outside of the context of this report.

Gambinicode+ consists of **order**, **width**, **height**, and **squares** ordered by placement.

While the order and height of a solution are redundant given the set of squares and the width, it is standard in both Bouwkampcode and Tablecode to provide the order, width, and height as well as the square placement.

Throughout this report, all results will be presented in Gambinicode+, which allows the reader to directly interpret the way solutions are produced, but I intend to provide a tool that converts Gambinicode+ to Bouwkampcode, so the results can be integrated into existing catalogues.



10 65 47 {22,19,24, 3 ,11, 5 , 6 ,25,23,17}

Figure 2.3: *The Gambinicode+ of an order 10 65x47 SPSR. The ordering of the squares unambiguously encodes the arrangement. To recover the rectangle, one places squares in the order given, according to our modified Gambini algorithm (detailed later), shown here to the left of the encoded solution.*

2.2 Important Bounds and Results

There are some bounds on Perfect Squared Rectangles that we can use to reduce the search space of Gambini's algorithm, some of which were presented by Gambini in his thesis [4], published after his original algorithm, but are presumed to have been used in his implementation. Additionally, I have proven a previously un-noted bound regarding boundary squares on each edge of PSRs and PSSs, which was also used to reduce the amount of work performed by the algorithm.

The bounds are not all implemented literally, but instead used to justify the correctness of various optimisations designed to exploit them, which are detailed in 5.2. Discussion of how the union of these optimisations do not violate exhaustion is in 5.4.

2.2.1 Gambini's Bounds

Minimum size for edge squares The minimum size a **boundary square** can be in a perfect squared rectangle is 5 [4].

> Both PSRs and PSSs exist with boundary squares of size 5 [4, 1].

Minimum size for corner squares The minimum size a **corner square** can be in a perfect squared rectangle is 9 [4].

> PSRs with corner squares of 9 do exist, found by Jasper Skinner [17], but it is thought this boundary may be higher for PSSs. The smallest known corner square in a CPSS is 9, but the smallest in a SPSS is 18. This is considered an active area of exploration in the field [15].

2.2.2 2-Square Edges in Squares and Rectangles

2-Square edges in Perfect Squared Squares

We assert for the first time that a perfect squared square (PSS) cannot have more than one edge with exactly two constituent squares. This bound does not extend to perfect squared rectangles (PSR).

In our literature review, despite the apparently simple nature of this bound, we have not found any sources that prove it, nor any that even explicitly state this. Proof of the bound itself is trivial, however, proof that exploiting it (combined with the other optimisations) does not compromise the exhaustive nature of the algorithm follows in section 5.4.

There are two cases for a Square to have two sides with only two constituent squares: adjacent sides, and opposite sides, shown in Figure 2.4.

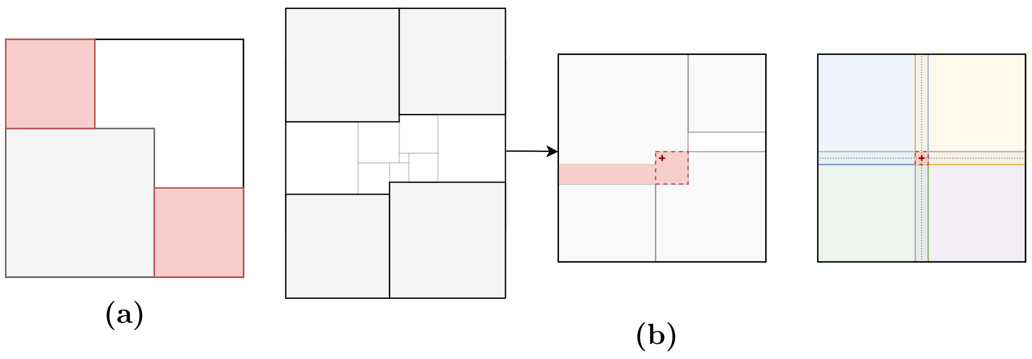


Figure 2.4: Case **(a)**: *Adjacent sides. As each pair of squares has identical combined length, so the squares in opposite corners must be the same size.*
Case **(b)**: *Opposite sides. Shown first as a rectangle for visual purposes, this shape is impossible in a square, unless all four squares are the same size. Each of the greater squares on each side must cover the centre, and so must overlap.*

For two **adjacent sides** of a PSS to each have only two constituent squares, the two squares that do not lie on the shared corner must be the same size, violating the *perfectness* of the decomposition.

For two **opposite sides** of a PSS to have only two constituent squares, each must have a square with sides greater than half the total width of the PSS. Each of these must cover the centre of the PSS, and therefore must overlap with one another.

This result directly implies a known result: that there must be at least seven squares on the boundary of a Perfect Squared Square [15].

2-square Edges in Perfect Squared Rectangles

In researching the bounds applicable to this project, I came across the fact that a Perfect Squared Rectangle **must have at least 5 squares along its boundary** – because, trivially, to have 4 would require that they all be the same size [15]. This provided a lower bound for the minimum, but the author omitted examples. For all other bounds, examples were provided in detail if they existed. Naturally, this implied that 5-boundary rectangles were either relatively rare (to the extent they were undiscovered or non-existent), or that they were so common it was not worth providing an example.

Before we understood it to be the latter, we sought to prove either that an SPSR with 5 boundary squares could not exist or to construct one. In Figure 2.5, I detail the ***exact* shape we discovered is necessary for a rectangle to have 5 boundary squares**, and the sub-shape we can seek to produce them.

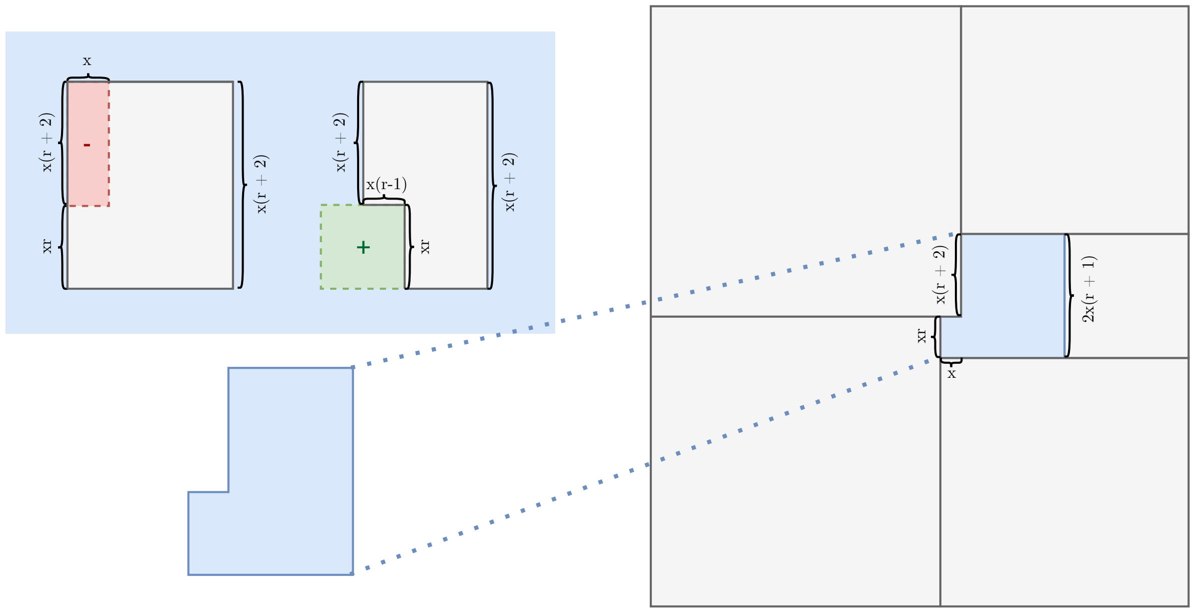


Figure 2.5: *The exact shape (under trivial transformations) of any rectangle with only 5 boundary squares. Shown on the left are two possible methods for constructing the essential sub-shape from certain CPSRs.*

Given this sub-shape, shown in blue in Figure 2.5, one can construct around it the **only possible** boundary shape of 5 squares, shown in grey. Additionally, I offer two means of constructing the sub-shape, by finding a Compound Perfect Squared Rectangle with a subrectangle that could be removed to create a notch. Two ratios are shown of CPSRs that could create this, one of which requires the addition of a Square for the desired ratio.

Another method for construction of the 5-perimeter SPSRs requires an SPSR which already has three edges with only two constituent squares. Shown in Figure 2.6, the same sub-shape is necessarily present. For a given sub-shape, it is guaranteed that there exists an SPSR that contains it: it is not guaranteed that there exists a CPSR with the properties shown in Figure 2.5, and so this method is more likely to lead to a high yield of 5-perimeter SPSRs.

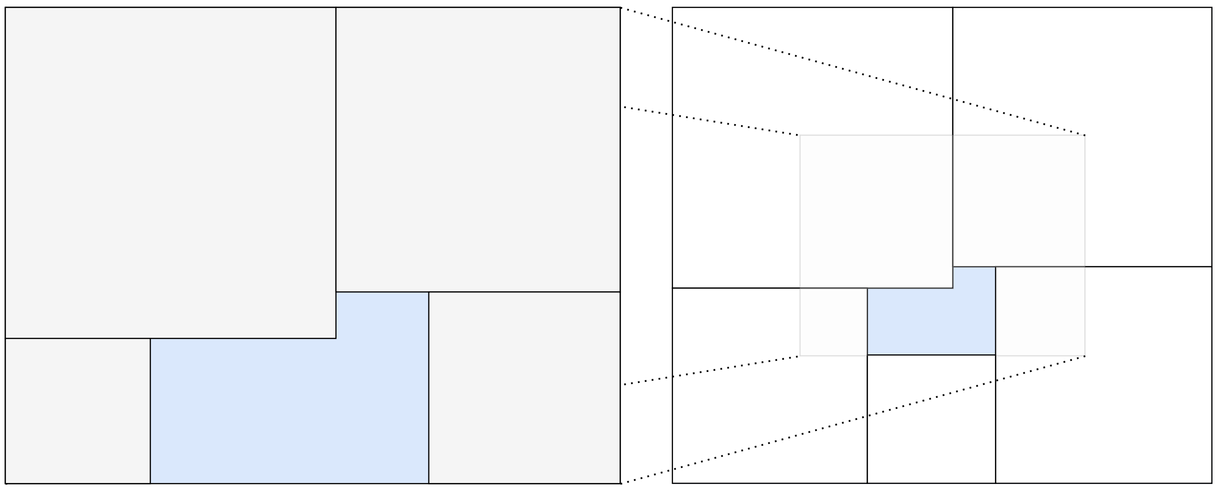


Figure 2.6: *Another method of construction, given an SPSR with three 2-square edges. The sub-shape is conserved, and other squares are added or replaced as shown.*

In this project, we not only found the **smallest possible rectangle with only 5 boundary squares** – not a new rectangle, but a new result – we also demonstrated that the construction from SPSRs in Figure 2.6 was fruitful in producing relatively small 5-square perimeter rectangles.

Additionally, exploring this bound was key to understanding to what extent we could implement an optimisation based on boundary squares: because we can only be certain that at least *one* edge of a rectangle has more than two squares, we implement this for a single edge: the bottom edge, as detailed in section 5.2.

Chapter 3

Core Implementation

3.1 Project Overview: The 3 Key Pillars

By splitting solving into multiple stages, we can speed up the algorithm, and still guarantee recovering *all* possible arrangements. To do this, we first determine sets that constitute decompositions, and then recover the arrangement of those sets, before processing them into results.

3.1.1 The Core algorithm: Finding *Sets* of Squares

If, instead of requiring the complete solutions, we instead seek only the *set* of squares that constitute a decomposition, we gain on multiple fronts:

Data structures: we don't need to conserve the order the squares are placed, and so can consider data structures such as HashSets and boolean arrays, which will have significantly faster performance on membership checks.

Additionally, we only need to find any set **once**, among many transformations of a given arrangement, which allows us to implement various pruning measures which will reduce the number of decompositions we find for a given set. It is **essential** that none of these optimisations clash and remove all versions of a given decomposition. Discussion of how this was avoided is in section 5.4.

3.1.2 The Solver: Recovering All Arrangements

Given a set of squares, we can recover every possible arrangement of those squares in a certain width in trivial time.

This includes all of the arrangements which were discarded through pruning, including any non-trivial isomers, and all trivial transformations (rotation and reflection), so we always find the canonically correct arrangement (defined in subsection 2.1.1).

Some rearrangements are *not* trivial transformations, but isomers with distinct arrangements: these are also recovered, but, essentially, none of the pruning optimisations entirely eliminate any isomer, pruning only trivial transformations of each arrangement.

3.1.3 Processing The Output

A tool was produced to produce SVG diagrams to visualise decompositions, following the logic of the algorithm itself.

Given these generated diagrams, one can easily sort decompositions by observation. While some of the sorting could be automated (rotational symmetry), one would need to inspect the solutions anyway to look for interesting properties, so it was decided this was not worth implementing.

In particular, we are looking for **isomers**, for the number of **boundary squares**, and any other artifacts of interest: many of which were found through this inspection, detailed in chapter 6.

3.2 The Original Algorithm

Various representations have been developed to represent *Squared Squares*, most famously electrical circuits [8]. Gambini proposed an intermediate representation for the construction of squares: plates. By placing squares on top of one another, and considering only the shape top-most layer, and the set of placed square sizes, Gambini developed two algorithms to square the square:

The first limited the number of plates at any point to a fixed integer, and works efficiently to produce many squared squares. The second exhausts the search space of a given size squared square, and is the one upon which we build our algorithm.

Shown in Figure 3.1 is an arbitrary subsection of the search space of Gambini's algorithm. In the algorithm, one identifies the smallest delimited plate, and places squares recursively to decomposed. **Only when the plate is entirely filled is a new one selected.**

Fundamentally, Gambini's algorithm decomposes one plate at a time.

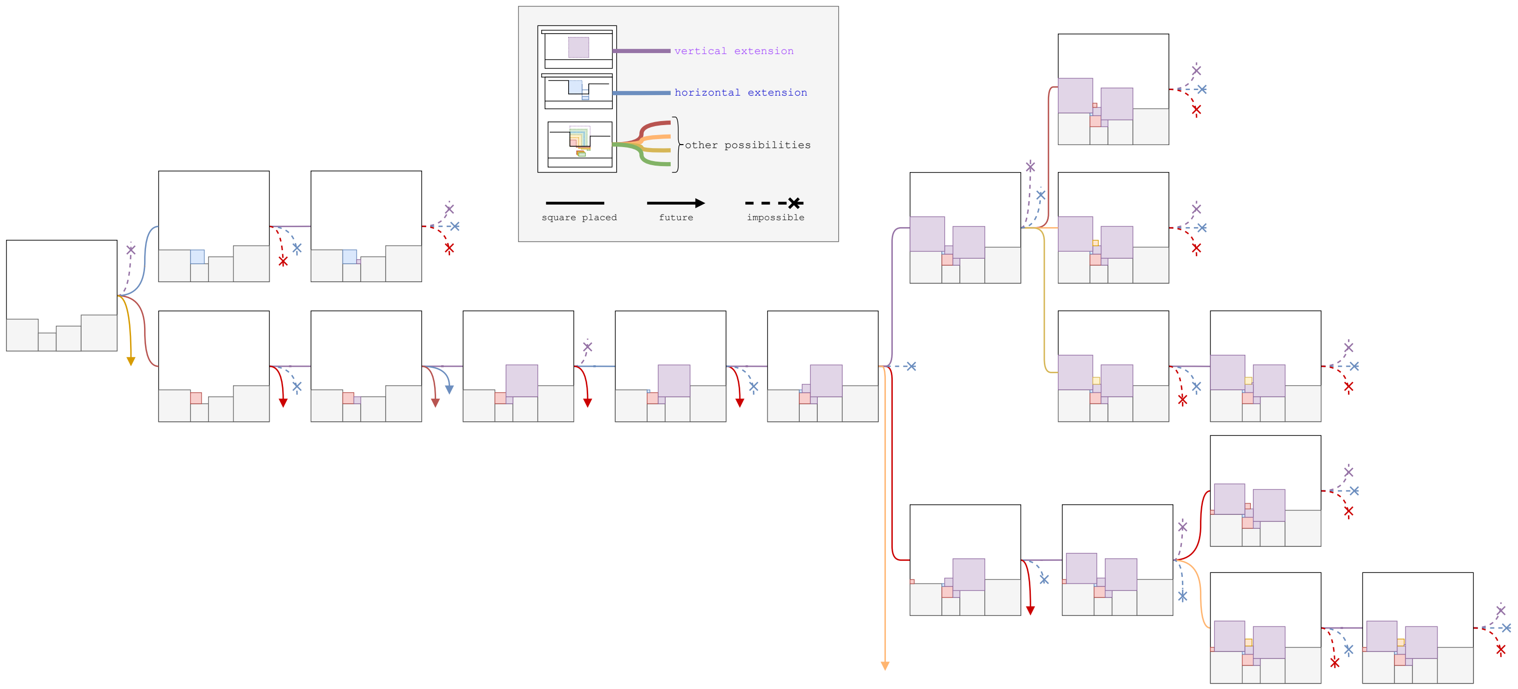


Figure 3.1: *An arbitrary subsection of Gambini's search space. Squares are always placed on delimited plates, decomposing an entire plate at a time.*

Shown in Figure 3.2 is a visual representation of Gambini's exhaustive algorithm. It is composed of two key functions which call themselves and each other recursively:

`decompose` Decomposing a plate consists of adding squares from left to right until the plate is filled, and then `decomposing` the next smallest plate. Before all of the following moves, a membership check prevents the placement of duplicate squares:

- **vertical extension** adding a square that fits into the entire delimited plate, extending it vertically.
- **horizontal extension** adding a square equal to the height difference between p_i and p_{i-1}, so that the left plate extends horizontally, shortening p_i.
- **remaining squares** starting at size 2 (because 1 can only be a part of the solution when it is a horizontal or vertical solution), we place all the possible remaining squares, the greatest being the minimum of either one less than the width of the plate, or the difference between the size of the total square and the height of the plate. This range is shown in Figure 3.3.

`next plate` When we vertically extend a plate, the smallest delimited plate that follows depends on the entire set of plates (unlike when we horizontally extend, or place other squares, in which case we know it is always p_{i+1}.). `next plate` selects the smallest delimited plate p_{min} and calls `decompose(`p_{min}`)`.

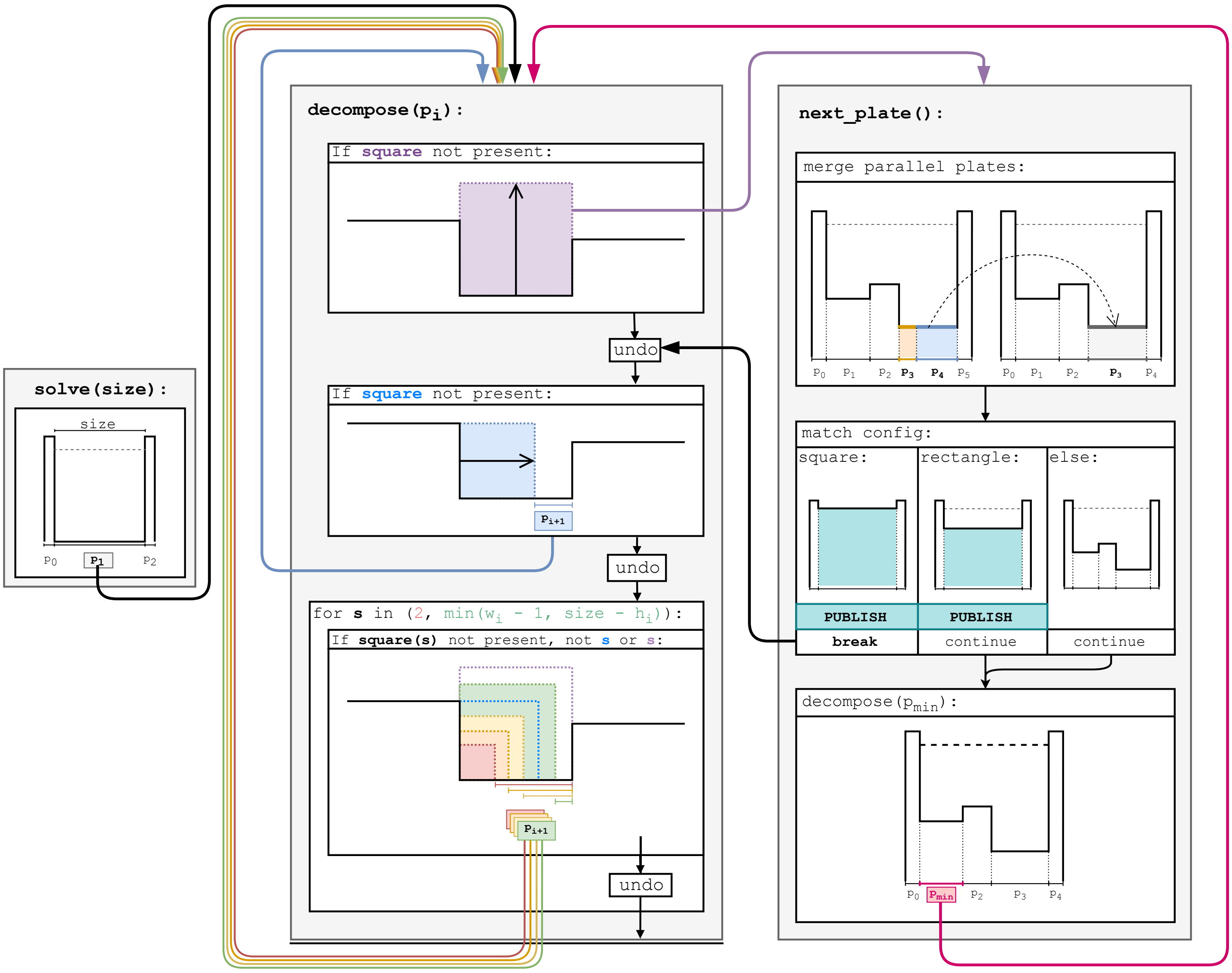


Figure 3.2: *Gambini's exhaustive algorithm for Squaring the Square.*

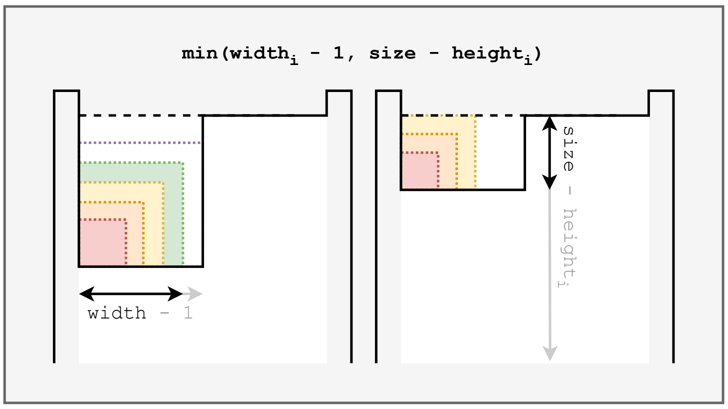


Figure 3.3: *In the iterative section of decomposition, we are limited by both the width of the plate, and the total size of the square.*

3.3 The Transformed Algorithm

One of the key developments in this project was a subtle but significant modification to the core of Gambini's algorithm. Later, we explore various optimisations which *prune* the search tree: cutting branches short that are already impossibilities. However, the modification we present here actually entirely transforms the algorithm and resulting search tree: significantly reducing its size, while conserving all successful solutions. The ordering of nodes is completely changed, and the distribution results in a significantly less asymmetric search space.

Intuitively, **Gambini's algorithm decomposes plates** recursively until solution or failure is reached. While this is intuitive, it is vulnerable to spending magnitudes of time on impossibilities. In development of this project, I understood the algorithm to be **placing squares, rather than decomposing plates** – at this granularity, it is clear that considering which plate to place on *at each step* is optimal.

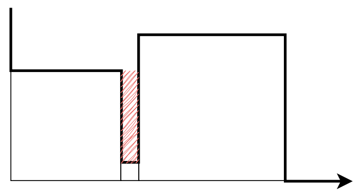


Figure 3.4: *An example of a forbidden shape. Filling the highlighted area is impossible.*

A regular occurrence in the runtime of the original algorithm is the placement of a square delimiting a thin plate to its left, resulting in an impossibility. One then proceeds with the futile effort of decomposing the much larger plate already being considered. Each successful decomposition of the larger plate eventually leads to an attempt to fill the smaller impossible plate, delaying inevitable failure to each node at the end of a very wide tree of exploration.

This modification relies on two facts: a smaller plate is significantly more likely to be an impossibility (by nature of being more likely to be taller than it is wide, and the limited squares that can be placed), and that it is always faster to decompose anyway. By prioritising quick decompositions that are more likely to fail, we bring failure forward and preserve correctness, transforming the algorithm and resulting search space.

Shown in Figure 3.4 is the basic shape of one kind of impossibility: often a plate that is taller than it is wide, where the width of the plate is already present as a square. This is a frequent occurrence in the runtime of the original algorithm: as all combinations of squares that fit a plate are attempted, there are always many configurations which contain squares of alternating sizes. By diverting to the smallest plate, we terminate quickly when an impossibility has been constructed.

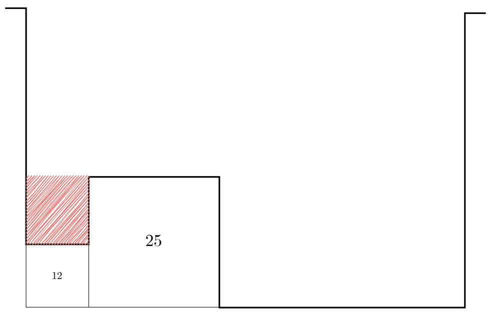


Figure 3.5: *The configuration for Figure 3.6. Gambini's original algorithm would continue to decompose the right plate before moving on to the impossibility it had already created.*

Shown in Figure 3.6 is the *very short* search performed by the algorithm to rule out the plate of width 12 shown in Figure 3.5 – this grows exponentially with width, and is especially quick on very small cases where one has already placed all squares up to that size, so it is straightforward to understand how much more efficient this is than if we were to decompose the entire right plate and append this futile search to the end of each successful decomposition.

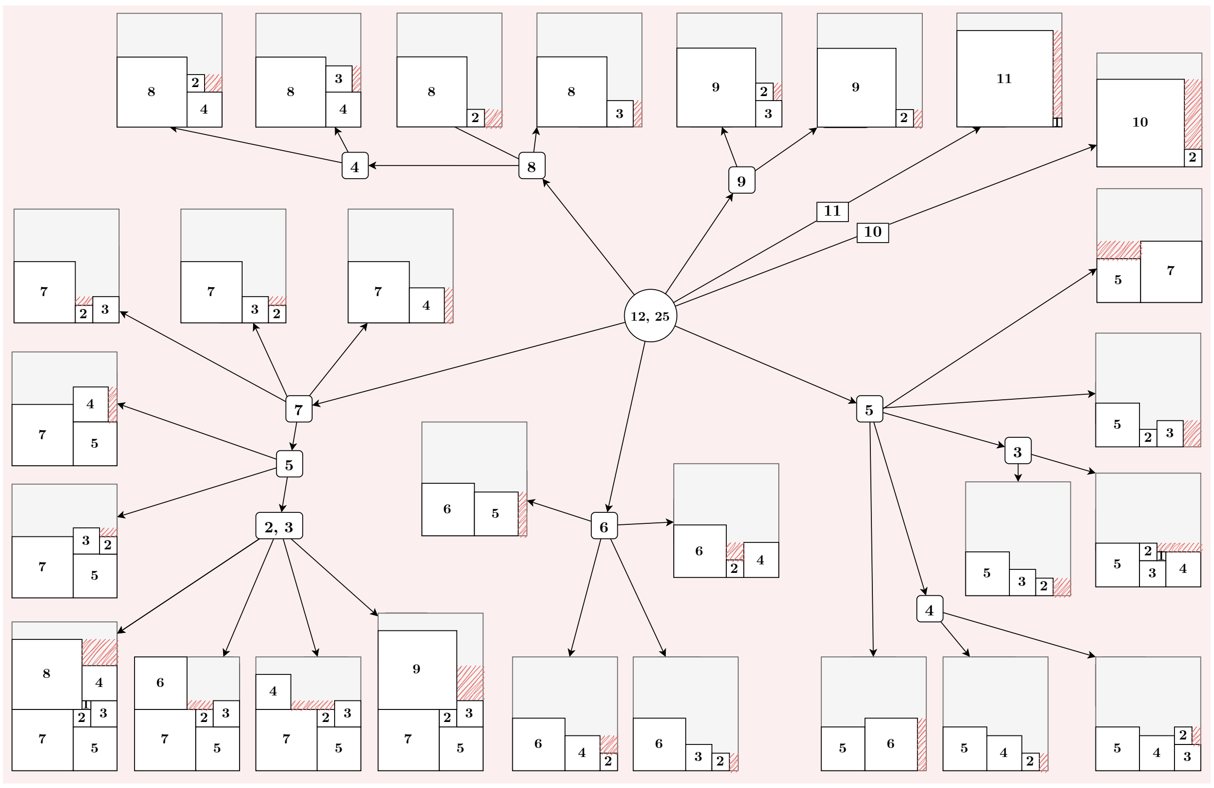


Figure 3.6: *The search space needed to exhaust an impossible plate of width 12, delimited by edges greater than 12. A human can enumerate this by hand in a few minutes.*

This modification does not tamper with the exhaustive nature of the algorithm, but does **change the order of placement in successful solutions**: hence the naming of Gambinicode+, to avoid directly implying that the encoding was exactly based upon Gambini's algorithm, but instead that it was referring to a placement that built on the foundations laid by Gambini.

Shown in Algorithm 1 is the modification made to the algorithm: while this appears to be a simple change, this significantly transforms the search, and changes the nature of the algorithm:

The pseudocode shown contains none of the secondary optimisations, as they do not impact the structure of the algorithm. Shown in Figure 3.7 is the modification to the algorithm shown in Algorithm 1.

Algorithm 1 ~~DECOMPOSE(i)~~ PLACE_SQUARE(i) ▷ *symbolic renaming*

```
1:  if $size\_not\_used(w_i) \wedge h_i + w_i \leq square\_size$ then
2:      ⟨save configuration⟩
3:      ⟨place square of size $w_i$⟩
4:      ~~MERGE_PLATES~~                          ▷ note: moved to within NEXT_PLATE
5:      NEXT_PLATE
6:      ⟨restore configuration⟩
7:  if $size\_not\_used(h_{i-1} - h_i) \wedge h_{i-1} - h_i < l_i$ then
8:      ⟨place square of size $h_{i-1} - h_i$⟩
9:      ~~DECOMPOSE($p_i$)~~ PLACE_SQUARE($p_i$)     ▷ symbolic: function renamed
10:     ⟨remove the square⟩
11: for s in 2..MIN($w_i - 1$, $square\_size - h_i$) do
12:     if $size\_not\_used(s) \wedge s \neq h_{i-1} - h_i$ then
13:         ⟨place square of size $s$⟩
14:         ~~DECOMPOSE($p_{i+1}$)~~ NEXT_PLATE   ▷ Major modification: always place
                                                    squares on smallest delimited plate.
15:         ⟨remove the square⟩
```

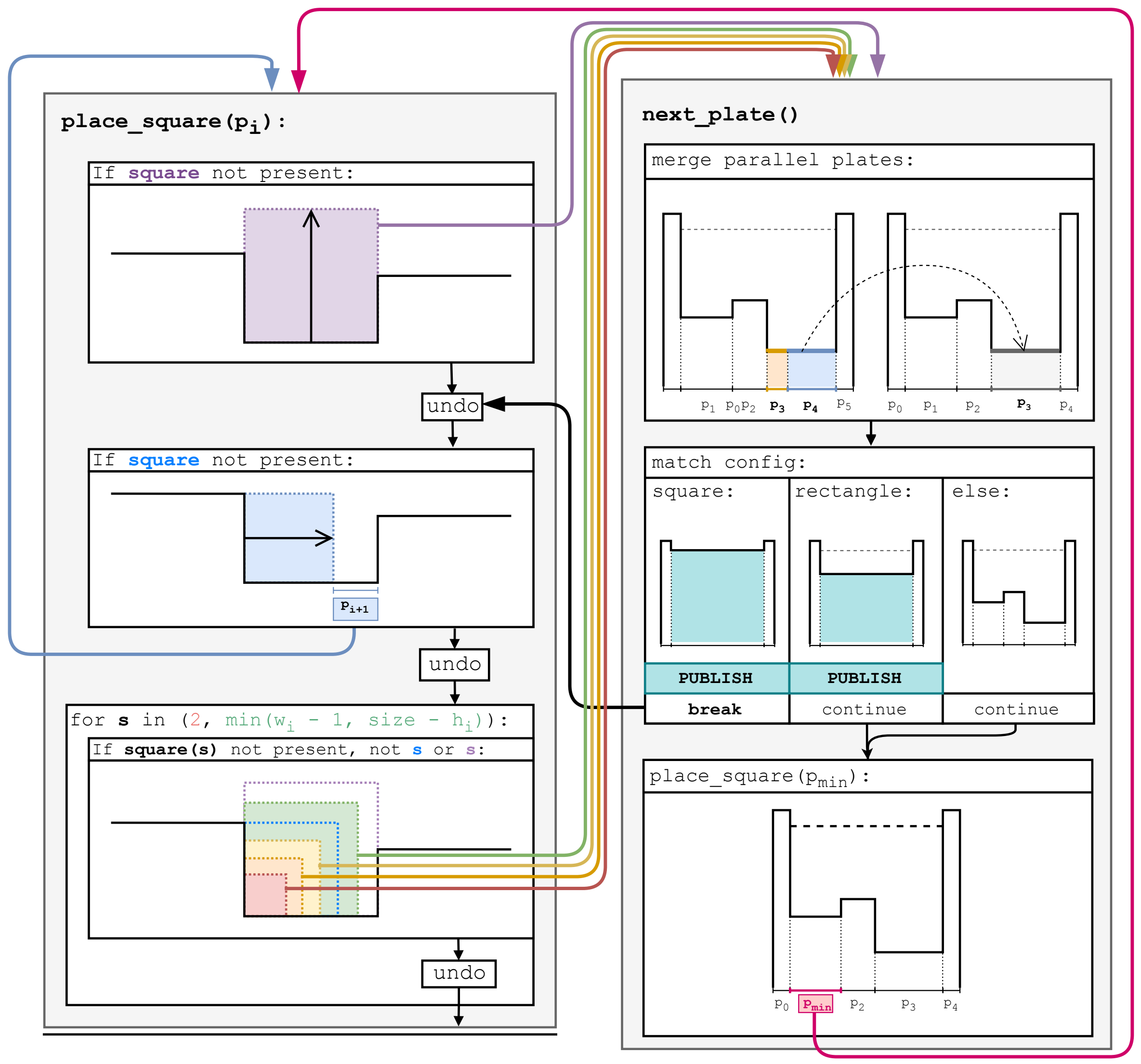


Figure 3.7: *The structure of the new algorithm, which always place squares on the smallest delimited plate.*

3.4 The Solver

The solver forms the second pillar of our project, converting the output of our core algorithm – an unordered sets of squares – into actual decompositions, and returning them in Gambinicode+ format. It does so by running the algorithm modified as follows:

Retain order of placement We store the placed squares in a vector, so that the output produces the correct Gambini Code for the solution produced.

Restricted set search the algorithm only attempts to place squares present in the solution it is searching for. As a result, the complexity is at worst exponential in the *order* of a solution rather than the size, with a relatively small coefficient growth coefficient.

No pruning optimisations In this version, no optimisations that reduce the search tree by reducing symmetry are included. The only optimisations are ones that prevent further exploration of dead-ends.

Algorithm 2 PLACE_SQUARE_SOLVE(p_i)

```
if remaining_squares.contains($w_i$) $\wedge$ $h_i + w_i \leq square\_size$ then
    ⟨save configuration⟩
    ⟨place square of size $w_i$⟩
    remaining_squares.remove($w_i$)
    NEXT_PLATE
    ⟨restore configuration⟩          ▷ side effect: restores $w_i$ to remaining_squares
if remaining_squares.contains($h_{i-1} - h_i$) $\wedge h_{i-1} - h_i < l_i$ then
    ⟨place square of size $h_{i-1} - h_i$⟩
    remaining_squares.remove($h_{i-1} - h_i$)
    PLACE_SQUARE_SOLVE($p_i$)
    remaining_squares.insert($h_{i-1} - h_i$)
    ⟨remove the square⟩
for s in 2..MIN($w_i - 1$, $square\_size - h_i$) do
    if remaining_squares.contains($s$) $\wedge$ $s \neq h_{i-1} - h_i$ then
        ⟨place square of size $s$⟩
        remaining_squares.remove($s$)
        NEXT_PLATE
        remaining_squares.insert($s$)
        ⟨remove the square⟩
```

The solver runs quickly (less than a few minutes for all solutions found in this project) even when single threaded. Its complexity is tied to the order of the solution it is seeking, not the size, and does not become prohibitive in the orders we are interested in: in any case, this solving algorithm is so significantly faster than any algorithm to *find* the set it is solving for, that it has a negligible effect on the time taken from start to finish.

Shown in Algorithm 2 is the addition of `remaining_squares`, a set of the squares in the solution that have not been placed, and the modification of conditionals to check instead if a square exists in `remaining_squares`, rather than if it exists in squares placed.

3.5 Processing

In processing the output of our solver, we first use the Gambinicode+ to recreate each decomposition visually, with a tool that converts Gambinicode+ to XML.

Given these generated diagrams, one can easily sort decompositions by observation. While some of the sorting could be automated (rotational symmetry), one would need to inspect the solutions anyway to look for interesting properties, so it was decided this was not worth implementing. Some of the SPSRs we are looking for have technically been enumerated, as Stuart Anderson has published a total *number* of SPSRs of orders 22 to 24, but has not processed these decompositions to find any of interest. It is through inspection that we expect to find new results.

In particular, we are looking for **isomers**, for the number of **boundary squares**, and any other artifacts of interest: many of which were found, detailed in chapter 6.

A tool to convert Gambinicode+ to Bouwkampcode is in production, all results will be converted so they can be integrated into existing catalogues. Additionally, the tools to produce the diagrams and Bouwkampcode will be open source, along with all code, so that further work can be done to enumerate SPSSs and SPSRs by width.

Chapter 4

Parallelisation

4.1 Trade-off Analysis

Because the algorithm could branch every single time it adds a square, the naïve method of spawning work units at *all* available opportunities comes with significant overhead: the majority of time spent would be on picking up work units, not doing work.

As a result of the asymmetry, great width, and limited depth of the search space, we are left to look for a balanced method for concurrency that combines dynamic and static parallelism to spread the work evenly amongst n cores. There are many systems by which we can divide the search space, and justification for the eventual result is presented in this chapter.

Independence

Each branch in the search space is completely independent of the rest of the search space: as work is done up the stack and back down, no side effects persist. Because of this, there is great scope for parallelism in this algorithm.

Depth

Additionally, because the depth of the search space (the maximum number of squares placed simultaneously in a given attempt to solve it) is bounded by a constant (the squares have minimum size, and a maximum combined area), we do not need to implement tracking beyond the stack, as there is no risk for overflow. This lends itself well to splitting the work space: we can define a *work unit* as the decomposition starting at a given plate and configuration, and face no issues keeping track of where the threads separate, or at what point the spawned thread needs to stop.

Asymmetry in Width

The subtrees of the search space are heavily asymmetric: many of the branches at a given point will terminate rapidly, and some will yield an enormous amount of work. Because of this, we pursued dynamically generating workunits when needed, in addition to statically splitting the search space into m workunits to be handled by n threads at the start of solving. The dynamic threading is designed to limit the effects of trailing work units.

High Frequency

Each thread has the opportunity to split *very frequently.* Thus, if we used global state to determine whether threads should split, we would create data races or spend significant work time spinning on the lock. If enforced synchronously, the threads would spend a great deal of time waiting to read from shared memory, and forcing unprotected reads in Rust would mean implementing unsafe code, which has been successfully avoided throughout this project.

Instead, we use asynchronous channels, which are composed of a sender and reciever. Messages are sent through channels in *one direction*, preventing data races or other unwanted behaviour.

4.2 Static Parallelism

4.2.1 Splitting The Search Space

The search tree we are exploring grows in width exponentially: to produce meaningful subtrees for exploration, we should start by **splitting the tree at its base**: creating individual work units for all possible combinations of the first n squares.

This allows us to split the workspace into k units, where $k > n$, and then have the n threads pick up additional units as they complete jobs.

This also allows us to prune the search tree using known bounds manually:

The first square :

- **Minimum size** the first square placed is guaranteed to be in a corner, and so, as shown by Gambini in his doctoral thesis [4], cannot be smaller than 9.
- **Symmetry** we do not bother to place any first squares greater than half of the width of the square: these are captured in symmetrical arrangements.

The second square :

- **Minimum size** the second square placed cannot be less than 5, as it lays on the boundary of the square [4]

The right corner It follows that we cannot place any squares that would leave any remaining space on the lower boundary less than 9, as we know the corner square must be at least that large [4].

2 square edge we do not produce any work units which contain 2 squares filling the lower edge. We know at most a PSR can have 3 edges with 2 squares [2.2.2]: preventing the lower edge from having two squares, we avoid 3 of these symmetries and a large number of deep work units.

The third square ... this nesting can be extended arbitrarily, but we found in practice that 2 produced work units of appropriate size.

4.3 Dynamic Parallelism

Splitting the search tree at the start by producing work units for each combination of the first n squares was effective, but we predicted it would result in some *trailing work units* – some combinations are exponentially more fruitful than others, and we expected to see that a small number of threads would handle a significant amount of remaining search space while the remainder idle.

To handle this, we also implemented *dynamic parallelism*. We implemented the necessary architecture for threads to split their remaining work space, at the behest of a single **coordinator** thread, with no data races, or unnecessary spinning.

4.3.1 The Coordinator

We constructed a `Coordinator` thread to control the entire process. The coordinator initiates solving, distributes work, and counts the squares placed.

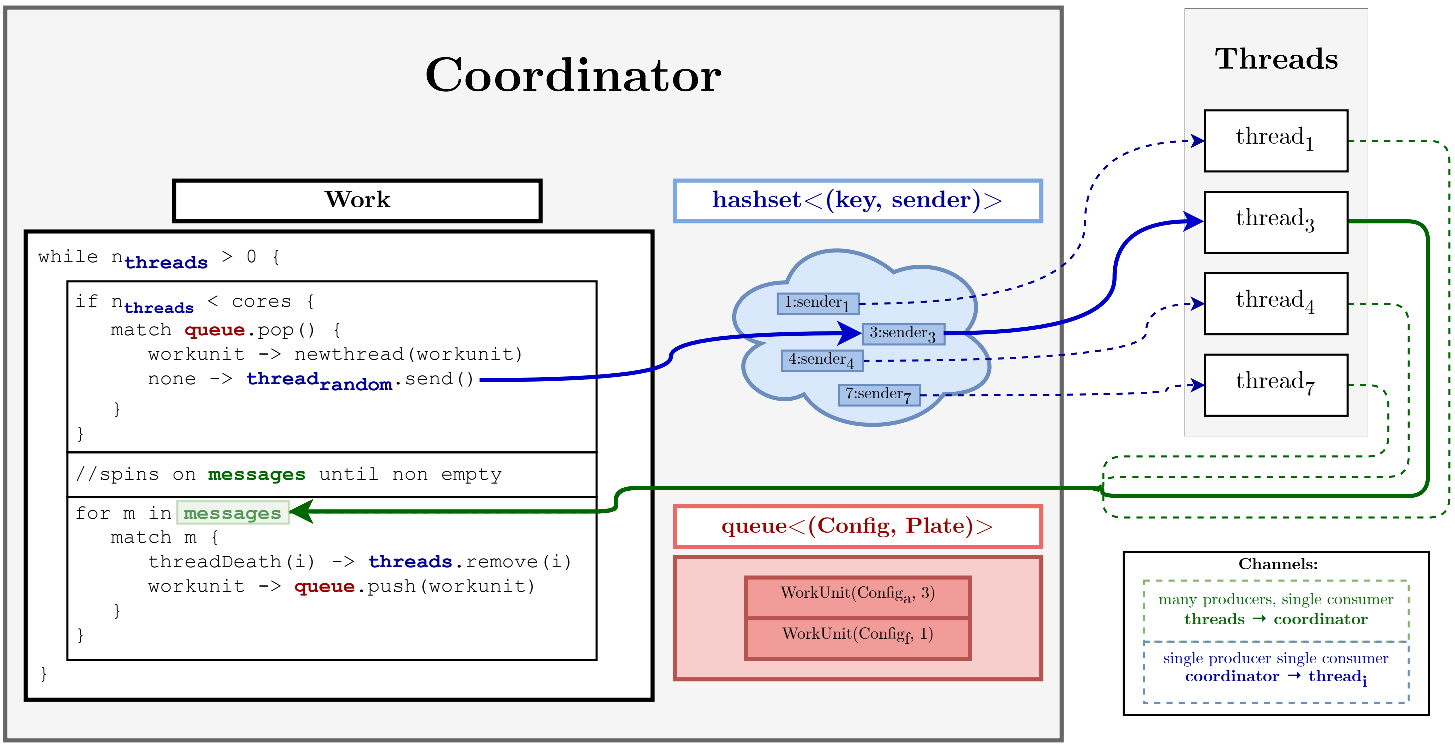
The `Coordinator` distributes work by observing the number of threads. When the number of threads is lower than the number of available cores, it either creates a thread to process a work unit from the queue or chooses an active thread at random and sends a message to it to split its remaining search space. Because the search space through which it will backtrack is held on the stack, and the search space ahead is held in the function call, the thread simply hands its entire current function call to the queue and then works back through its stack.

```
pub enum Message {
    ThreadDeath(index : usize, n_squares : u128),
    WorkUnit((config : Config, plate_id : usize)),
}
```

Figure 4.1: *The* `Message` *enum.*

The Coordinator receives two types of messages, shown in Figure 4.1. Upon receiving `ThreadDeath`, the coordinator removes the sender to the relevant thread (given by index.

Because the channel is *many producer, single receiver*, the receiver does not otherwise know which thread produced the message.), and adds the `squares_placed` to the total count of squares. A `WorkUnit` is used to produce new threads when needed.



```
while n_threads > 0 {
    if n_threads < cores {
        match queue.pop() {
            workunit -> newthread(workunit)
            none -> thread_random.send()
        }
    }
    //spins on messages until non empty
    for m in messages
        match m {
            threadDeath(i) -> threads.remove(i)
            workunit -> queue.push(workunit)
        }
    }
}
```

Figure 4.2: *The dynamic coordinator: using asynchronous channels, we can avoid any shared state, to prevent wasted time spinning. The logic shown is robust and passed all test cases.*

4.4 Hardware Choices: Apple Silicon and The HPC

It was planned that initial development would be on a Macbook Pro with M1 Pro cores, before processing was moved to a higher performance computing network available in the university. The combination of Rust, the standard `mpsr` libraries used, and the architecture on these servers resulted in 76 cores on the HPC being slower than 10 cores on the Macbook.

Brief analysis showed that the HPC cores were spending 95%+ time on kernel tasks, instead of work. Despite the code not logically containing *any* spin points, the cores were spinning on shared memory, which was unexpected behaviour for MPSR channels implemented in Rust. While this is something we could have pursued, it was clear we were within the realm of significant speedup on the Macbook alone and decided to pursue reducing the complexity and total runtime of the algorithm over debugging OS-specific bugs. Results exceeded expectations, due to algorithmic changes and optimisations.

Apple Silicon also demonstrated impressive performance when producing and scheduling miniscule threads, exhibiting almost no overhead.

Chapter 5

Optimisation and Analysis

We detail in this chapter the optimisations that target reducing dead-end sections in the search space, transforming the search space, and parallelising the program. We analyse how particular choices affected the search space, as observed through processing time. Each optimisation is compared to the algorithm immediately before it was implemented, and so the speedups are cumulative. Throughout the analysis, a speed-up is considered to imply a proportional reduction in redundant search space, unless otherwise specified.

Each speedup is shown relative to the optimisation made immediately before it, to demonstrate the stepwise improvements.

5.1 Speedups from Parallelisation

The initial algorithm grows exponentially, and provides a baseline against which we can compare our optimisations. After implementing the dynamic parallelism detailed in chapter 4, increasing the maximum number of concurrent threads increased performance, albeit non-linearly: 9 to 10 cores[1], the number available on the machine used in development, provided a 6.2x speedup.

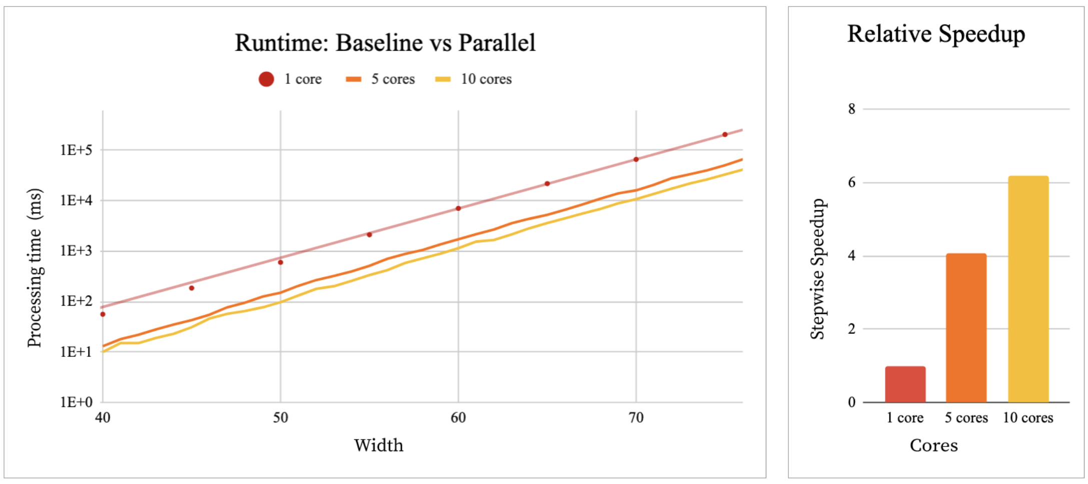


Figure 5.1: *Performance of the original algorithm as implemented, across 1, 5, and 10 concurrent threads.*

[1]As the machine was a personal laptop, also used for development and writing of the report, the algorithm was allowed to run between 9 and 10 cores at a time, run with high priority.

5.2 Proof-justified Optimisations

Corner Squares

The first domain-specific optimisation is Gambini's bound that the corner squares must be at least 9. Given this, by *only* implementing the restriction that **the *first* square not be smaller than** 9**,** I achieved a further 1.77x speedup.

It is possible to implement this bound elsewhere in the search space, which we do: when we double nested the production of work units (placing two initial squares for each work unit), we prevent the placement of squares that would not allow a square of at least 9 to fit in the rightmost corner.

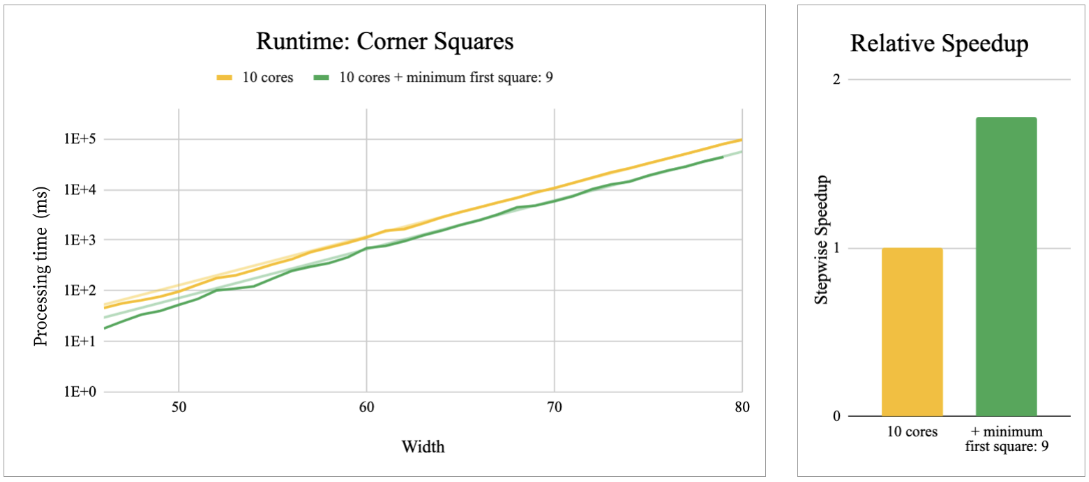


Figure 5.2: *Performance of the algorithm with the addition of the first square optimisation.*

Squares on Boundaries

Another bound proven by Gambini is that the minimum size square on the boundary of a Squared Square is 5. The first optimisation was to **prevent squares on the lower boundary of the rectangle from being smaller than 5,** before implementing the same bound for the sides and then the top.

Shown in Figure 5.3, this alone makes the program 2.66x faster – even running on 5 cores it is 1.89 times faster than 10 cores without it.

Notably, it dramatically changes the distribution of runtimes across the search space. Previously, growth was uniformly exponential, with little noise. Evidently, the pruned search space was proportionally larger for some sizes than others. Analysis of these intermittent peak values yielded, at this stage, no notable patterns. Analysis of why this occurs, and the algorithmic change that eliminated it is in subsection 5.3.1.

The distribution of processing time was validated across multiple runs, machines, and number of cores to have this same distribution. It is not noise.

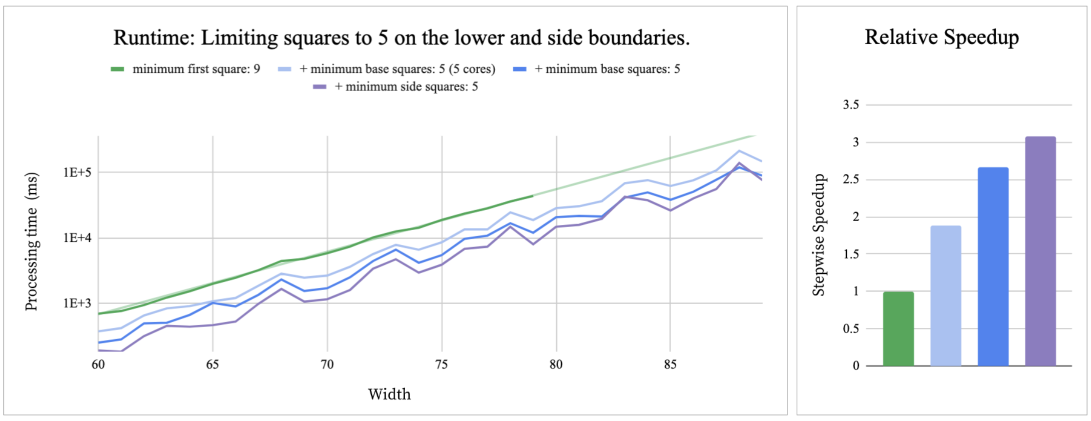


Figure 5.3: *Performance of the algorithm with the addition of the boundary squares optimisations, relative to the the version up to corner square optimisation.*

The second implementation of this bound was to prevent the placement of squares smaller than 5 on the left and right edges of the rectangle. This yielded an additional 1.16x speedup. While this applies to more *attempts* to place squares, these branches are naturally less deep than those placed on the bottom, and so terminate quickly even if left alone, resulting in the

It is unnecessary to implement any optimisation that would prevent the placement of < 5 squares along the top edge of the rectangle: the additional logic would not yield a worthy improvement.

Combining the implementations of this bound on the bottom and sides of the decomposition, we produced an 3.08x speedup.

At this point, we have a speedup, on the same number of cores, of 5.46x. Compared to the single-threaded algorithm: a 33.72x speedup.

A note on parallelisation

At this point, we also implemented *static* parallelisation, splitting the work units at the start of processing. It significantly reduces the number of total work units, but has a negligible effect on runtime compared to the dynamic parallelisation when run on Apple Silicon, presumably due to the architecture's reduced overhead in producing new threads. Either method runs well on Apple Silicon, and independently result in the same speedup.

2 Square Edges

We proved in subsection 2.2.2 that it was not possible for a squared square to have more than 1 edge with only 2 constituent squares. It *is* possible for a squared rectangle to have up to 3 edges with only 2 constituent squares. Because of this, our implementation can only exploit the latter without eliminating rectangular solutions:

To do this, we did not search for solutions that started with 2 squares filling the rectangle's bottom edge. This yielded a speedup of 2.36x due to reducing the reduced number of initial work units, of which many had immense search spaces. It is important to note this optimisation **eliminates some symmetries** of some solutions.

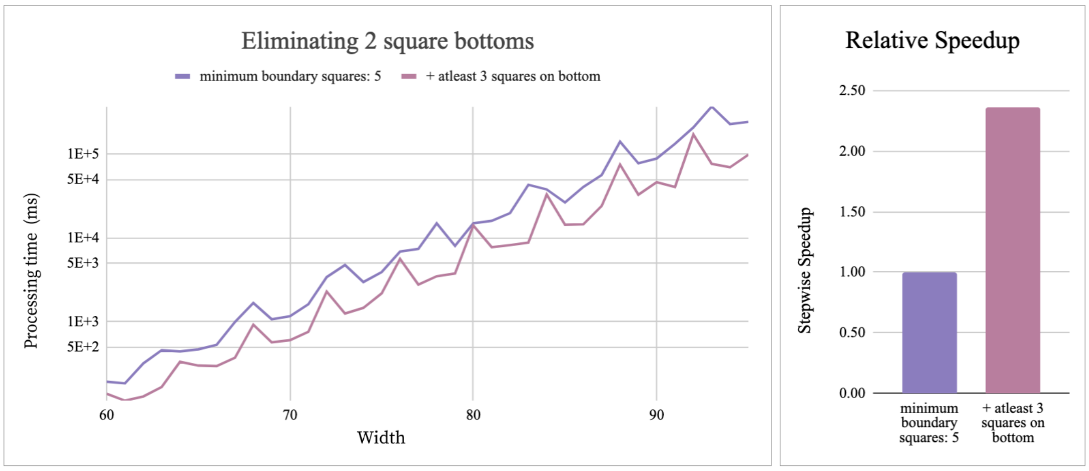


Figure 5.4: *Performance of the algorithm with the addition of the bottom edge always containing at least 3 squares.*

Large First Squares

We can remove some of the symmetry in the solution set by not placing squares greater than half the width of the edge as the first square. This reduces the first square placements by half, though it does not reduce the search space by so much as the search spaces removed are less fruitful. It is a no-cost speedup, as we implement this in the work units' production, not the decomposition logic. It reduces the work done by 11%, and, like the 2-square edge reduction, *does* remove some symmetries of some solutions. In section 5.4, we detail why the combination of these optimisations **does not entirely eliminate any solutions**.

5.3 The New Algorithm

5.3.1 Disproportionate Search Spaces

The optimisations based Gambini's bounds – that there are no squares smaller than 5 on the boundaries, and none smaller than 9 in the corners – results in a pronounced pattern.

The processing time distribution was validated by running the program across multiple machines, cores, and runs. The pattern is not noise.

To understand this pattern, we explored the program's behaviour as it traversed the search tree. To do this, we removed dynamic parallelism and allowed the program to run from the original queue of work units.

Importantly, at the point when the queue is exhausted, fewer than N (where N is the maximum number of available cores) threads will continue searching large work units while others idle. Using the dynamic coordinator, these would instead pick off small work units produced by the main threads. At their depth, they cannot provide meaningful work units.

On most widths, the amount of time spent after the queue is exhausted is not significant, but when plotted, we can see clearly that the overall pattern is an expression of the following underlying behaviour:

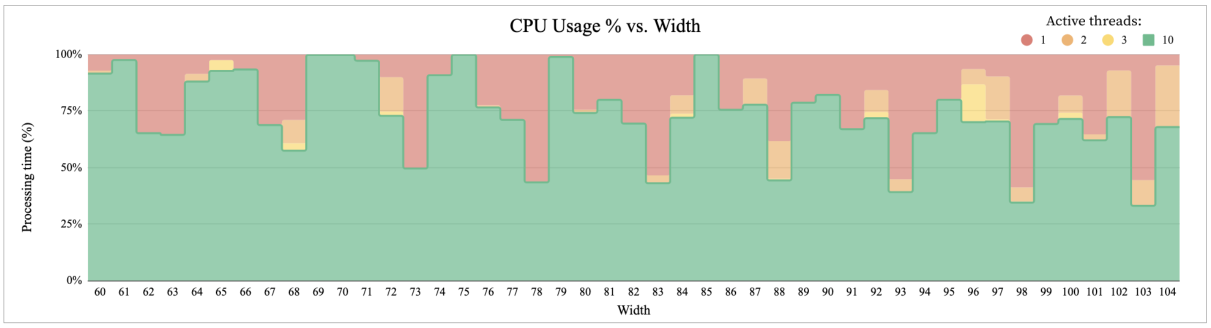


Figure 5.5: *The* ***proportion*** *of total processing time spent on X cores before meaningful work units are exhausted.*

At each $w = 3 \pmod 5$, the processing time jumps significantly, specifically because there is always one branch that is **significantly** wider than all of the others.

Prior to this analysis, the total processing time alone did not isolate that series. Normalising the distributed time as if m cores are exactly m times faster than 1 *does not* recover proportional growth. Evidently, the search space pruned is genuinely not exponentially proportional across widths, which is exacerbated by the attempts to parallelise it. If it were, we would see that the normalisation returned the distribution to a clear exponential.

This pattern is therefore a direct result of asymmetry in the search space: at least one work unit eclipsing the others in size, resulting in a trailing thread spending as long as all of the other threads combined on a single work unit. Intuition suggests the minimum size on the boundaries being exactly 5 is likely the reason the trailing units appear every 5 widths, but it is unclear what produces the offset, and the other outliers. While this may warrant further research, we implemented a solution to minimise the impact of the trailing threads, before the modification to the algorithm eliminated it entirely.

5.3.2 Overlapping Sizes

A straightforward solution to trailing threads is to start computation on the next size when a queue of work units has been exhausted. Because the length of the trailing units is proportional to the time taken to solve for the rest of the size, this does not create a cumulative issue – the longest searches complete within one or two further sizes being searched. This solution allows us to make use of all available cores throughout computation. As a result, we saw an 1.4x speedup. However, **the new algorithm rendered the overlap negligible**, making this speedup redundant, but a good solution for any further optimisations that could result in trailing units.

5.3.3 The New Algorithm

When we implemented the new algorithm – selecting the smallest delimited plate *every time* we placed a square, rather than after decomposing each plate – we produced an impressive speedup. Notably, the distribution of computing time by size was reduced exponentially, and the spikes we saw previously every 5 increments in width are mostly flattened.

This lines up with our expectations: the modification brings failure forward, prioritising small plates over much larger remaining plates, so that we perform much smaller searches first. The reduction of spikes implies that these eclipsing work units were vulnerable to this failure case: creating impossibilities outside the large plate with a significant remaining decomposition.

Not only did we see a speedup in the total time up to 112 of 18.64x, but the change also reduced the exponential complexity of the time/search space. Particularly, measured from 70 onward to reduce noise, we see a reduction from $O(e^{0.224})$ to $O(e^{0.148})$, on top of the constant factor of 6.02x speedup.

Projected to width 150, we expect it was at least a 200x speedup in the amount of computation time needed – it would have taken nearly a year to solve for 150 *alone*. Our algorithm enumerated width 150 in only 36 hours. Cumulatively, it took 11 days to produce all results presented in this report.

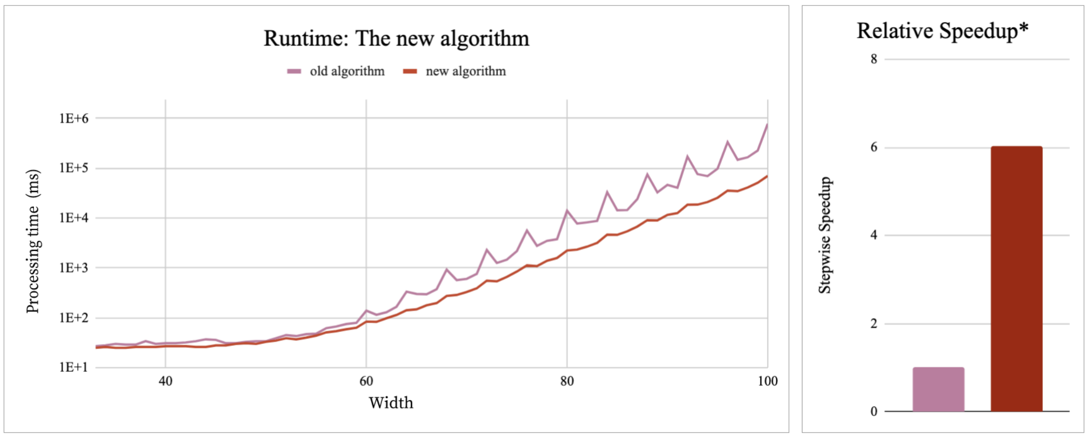


Figure 5.6: *The runtime of the old and new algorithm, up to width 100. The new algorithm is a significant exponential speedup.*
*Relative speedup in the range shown. The algorithm has reduced, so speedup increases with size.

5.4 Clash Prevention: Conserving Decompositions

Two of our optimisations did not prune impossible branches, but removed a subset of the symmetries of some solutions. Because of this, it is important to ensure that no entire solution could have all symmetries removed.

Consider the following example of a set of optimisations that are *each* legal, but combine to eliminate an entire known solution:

A : The minimum size for a square in a corner is 9, and each corner must be a different size. Therefore we could write an optimisation that places only squares at least as large as 12 in the lower left corner. We know every solution will have at least one corner of size greater than 12.

B : Every edge of the decomposition has two corners: a smaller and larger square. We can reduce symmetry by only considering placements along the bottom of a decomposition in which the leftmost square is smaller than the rightmost square.

C : Defined earlier and implemented in this project, we do not consider decompositions which have only two boundary squares at the bottom.

Shown in Figure 5.7 is an example decomposition that the given set of hypothetical proof-justified optimisations can eliminate. While individually justifiable, in combination they cover the set of transformations of this decomposition and would therefore violate correctness if combined.

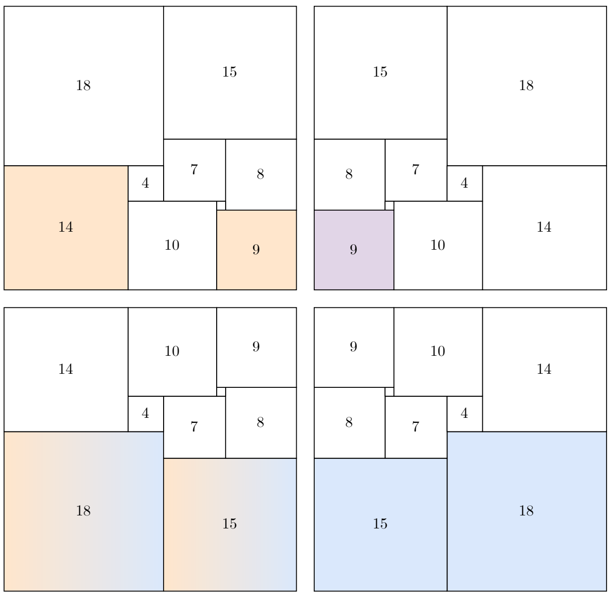


Figure 5.7: *A correct decomposition, eliminated by a combination of individually legal optimisations. Shown in color are the violating squares, colour coded by the hypothetical optimisation that remove them from the solution set.*

Careful analysis of our symmetry reducing optimisations was undertaken to ensure they could not entirely eliminate any solution: We do not place squares greater than half the width in the lower left corner, but no other optimisation reduces left-right symmetry (which is the vulnerability in the above example). Therefore no other optimisation can eliminate the removed solution's horizontal reflection without also eliminating it (implying there is at least one other solution, because both are independently legal).

As a sanity check, we confirmed the algorithm found all known decompositions up to width 150.

Chapter 6

Results

In this project, we enumerated all perfect squared squares and rectangles up to width 150– extending from previous work up to 112. To do so, we placed 190 trillion squares, in 263 hours, across 9 M1-Pro cores on a personal machine. This is at least a 475x speedup in the space enumerated.

In the development of the algorithm, various optimisations were applied , and we fundamentally changed the nature of Gambini's original algorithm, reducing the exponential complexity of the time taken and total squares placed to exhaust the equivalent space. The speedups of all improvements are summarised in this chapter, but detailed in chapter 5.

As a result, we enumerated a previously impossible search space: all squares and rectangles with edges up to width 150. In this chapter, we discuss previously unknown results, including the very exciting discovery of the **ninth** ever triple isomer.

The algorithm was exponentially improved, and the discoveries found exceeded expectations.

6.1 Performance

6.1.1 Establishing a Baseline

It is essential to acknowledge that some amount of our ability to explore more space than Gambini in 1999 is a matter of raw computing power due to technological development. To account for this, I first implemented the algorithm described by Gambini's pseudocode without parallelism or any optimisations (hardware or algorithmic). We could not run this code up to 112 due to time constraints, but it ran within an order of magnitude of what we expected given the machines each program ran on, implying it was a fair implementation of Gambini's pseudocode.

From the improvement in computer architecture and various design choices, we were able to predict that we could enumerate 112 in approximately 100 hours on a Macbook running on M1 Pro cores. This is in line with the approximate magnitude we were expecting prior to implementing Gambini's bounds.

For the baseline, we measured every fifth width between 30 and 80. Against this baseline, we parallelise the algorithm, and with that speedup, all further measurements are made for all widths until prohibited by time.

6.1.2 Improvement from Parallelisation

After implementing the dynamic parallelism described, prior to leveraging the algorithmic change, we saw a 6.17x speedup: the work was undertaken across 10 cores[1], so this constitutes 61% of the maximum speedup we would expect to see from perfectly splitting the search space of the algorithm across 9 cores, with perfect scheduling.

6.1.3 The New Algorithm: An Exponential Improvement

By comparing the exact number of squares placed, we can assess the pure improvements made to the algorithm, independent of architectures and concurrency:

Gambini placed 1,633,183,723,812 squares to enumerate up to 112.
Our algorithm placed 420,436,182,041 to enumerate the same space.

The change that reduced the square placement most significantly was that of the major algorithmic restructure (section 3.3).: in the space Gambini explored (up to 112), this resulted in an overall 3.88x reduction in squares placed, and a 6x speedup, but as it was an exponential reduction in both the order of squares placed and runtime, this improvement grows with width – we estimate that in the space we enumerated we would have placed at most 300x fewer squares than Gambini's original algorithm would have had to place.

Overall, we reduced complexity from $O(e^{0.224})$ to $O(e^{0.148})$ through the restructuring, which constitutes a *significant* reduction even at low widths.

6.1.4 Further Improvements from Design and Optimisations

Design Design decisions, such as exploiting the exact size of the cache line to reduce complexity in membership checks, saw an improvement of 2x prior to the baseline. As these do not directly improve the algorithm, and are dependent on architecture, language, and libraries used, the baseline was set after these decisions.

Mathematical reductions Detailed in section 5.2 is a variety of optimisations implemented to take advantage of known mathematical bounds on various properties of PSRs and PSSs.

[1]This is a conservative assumption: as the machine used was also being used for development and writing, significant time was spent at 9 cores.

Overall, across the optimisations, including that of implementing a novel bound presented in subsection 2.2.2, saw an overall speedup of 79.7x. Details of the exact improvement each optimisation offered are in section 5.2, including details of the effects of each on the distribution of the search space.

6.1.5 Total Speedup

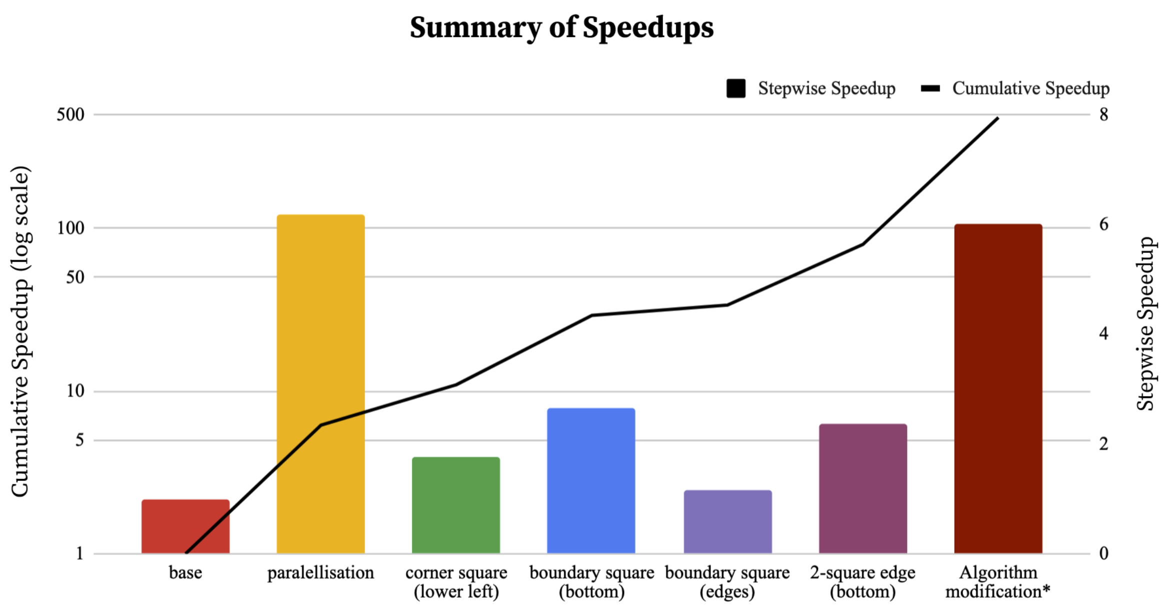


Figure 6.1: *A graph showing the stepwise and cumulative speedups of each optimisation.* *The algorithmic modification produced an *exponential* reduction in time, and so the shown value is for the finite space enumerated.

In total, we saw a speedup of at least 479x in the space enumerated, and a reduction of complexity from $O(e^{0.224x})$ to $O(e^{0.148x})$.

In the space of 0 to 80, which the new algorithm enumerates in 14 seconds, we saw a speedup of 280x relative to the baseline (43x relative to the parallelised baseline across the same number of cores).

Due to the reduction in complexity and prohibitive runtime of the baseline we cannot directly compare the runtimes – through the analysis given, we project that the baseline implementation would have taken nearly a year to solve for width 150 alone, and four and a half years to cumulatively reach 150.

Our computation enumerated width 150 in 36 hours, and enumerated all decompositions *up to and including* width 150 in only 11 days. By the analysis shown, it would have taken **4 and a half years to achieve the same results** without the complexity reduction given by our new algorithm.

6.2 Discoveries and Knowledge Gained

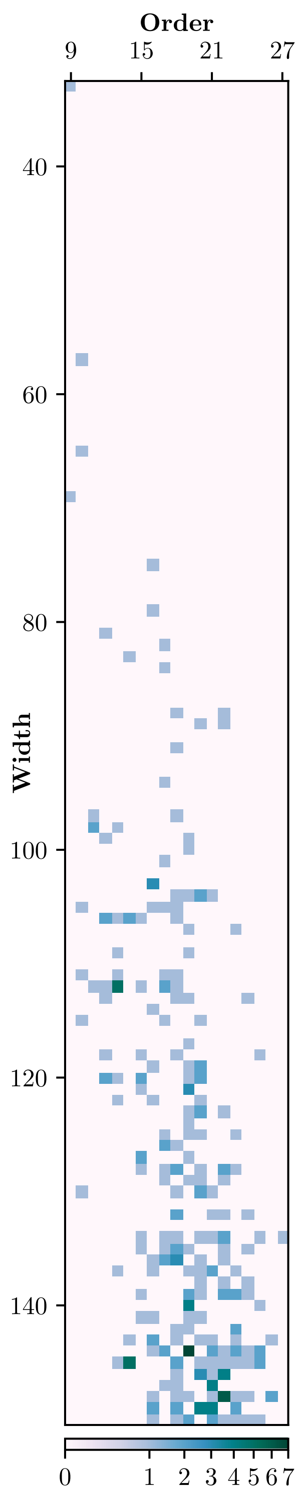


Figure 6.2: *A heatmap of all SPSRs up to width 150*

Through the exponential improvements detailed, we were able to definitively enumerate a significant search space: all squares and rectangles with edges up to size 150. Gambini was able to reach 112, of which only 110-112 included Squares. The search space grows exponentially with size – we placed 138,550,916,945,954 squares to find 294 SPSRs/SPSSs, **66 of which were previously undiscovered SPSRs**.

A heatmap of all SPSRs up to width 150 is shown in Figure 6.2, and the *complete* table listing these results is in the appendix in section A.2.

Our work proved that many of these SPSRs were the smallest of their order, or had other key properties, and that no further perfect squared rectangles or squares exist up to width 150 beyond those published in this report. Also of interest is the smallest SPSRs with each of 6 to 12 boundary squares.

Most excitingly, we discovered a **brand new set of triple isomers, the smallest in existence and the first of its order**, detailed in Figure 6.4 below. The isomer set has many remarkable properties.

6.2.1 Extending the Exhausted Space

Ian Gambini showed that there existed no Squares up to size 109, and that exactly 2 squares of size 110 existed, none of 111, and 1 of 112. This is considered one of the most important results in this field, which I hoped to extend. In exhausting the space up to 150, we found 294 simple perfect decompositions, 66 previously undiscovered. Additionally, we provably found the smallest SPSRs for orders 17 to 27, of which 22 to 27 were also undiscovered.

Decompositions

The project enumerated SPSRs, SPSSs, CPSSs, and CPSRs up to width 150.

SPSRs: Shown in Figure 6.2 is the heatmap showing the number of SPSRs found by width and order. Of the 282 SPSRs, 66 were previously undiscovered.

SPSSs: Table 6.1 is the total list of all SPSSs found. Of the 12 SPSSs, all were previously discovered – however, it is a new result that the SPSS **25 : 147 147** is the smallest SPSS of order 25.

CPSRs: We found 30 CPSRs, 16 of which were of orders that have not been previously enumerated for CPSRs.

CPSSs: *No* CPSSs were found, which is surprising.

	order					
width	**21**	**22**	**23**	**24**	**25**	**sum**
110	0	2	0	0	0	**2**
112	1	0	0	0	0	**1**
120	0	0	0	1	0	**1**
139	0	1	1	0	0	**2**
140	0	0	2	0	0	**2**
145	0	0	1	0	0	**1**
147	0	2	0	0	1	**3**

Table 6.1: *The complete enumeration of SPSSs up to width 150.*

6.2.2 Notable Results

Smallest PSRs by Order

We have proven that the PSRs in Table 6.2 are the smallest of their order among SPSSs, SPSRs, and CPSRs. Where a CPSR is the smallest, the smallest SPSR is also shown.

Five of the nineteen smallest SPSRs for their order have edges that only differ by one: orders **9, 11, 12, 13, 15,** and *nearly* **10** and **16**. Among the 324 total decompositions we found, only **fourteen** (4.3%) have this feature. The disproportionate prevalence (26%) among these special SPSRs is peculiar, and perhaps warrants further research.

Order	Smallest Perfect Decomposition (Gambinicode+)
9	9 33 32 {14, 10, 92, 1, 8, 4, 7, 18, 15}
10	10 57 55 {25, 17, 15, 2, 13, 8, 11, 30, 3, 27}
11	11 97 65 {65, 15, 8, 9, 7, 1, 10, 18, 4, 14, 32}
11	11 97 96 {40, 26, 31, 14, 12, 7, 24, 2, 17, 56, 41}
12	12 81 80 {36, 23, 22, 1, 21, 13, 11, 2, 9, 44, 7, 37}
13	13 98 97 {41, 28, 29, 13, 11, 4, 3, 26, 7, 2, 16, 56, 42}
14	14 83 77 {34, 25, 24, 5, 6, 13, 9, 12, 4, 8, 1, 7, 43, 40}
15	15 106 105 {46, 21, 16, 23, 5, 4, 7, 1, 3, 25, 2, 35, 59, 12, 47}
16	16 75 73 {32, 20, 23, 12, 8, 7, 6, 2, 16, 9, 41, 3, 1, 5, 4, 34}
17	17 82 75 {32, 18, 15, 17, 3, 10, 2, 19, 14, 7, 6, 11, 1, 5, 43, 4, 39}
18	18 88 65 {32, 31, 25, 12, 13, 1, 16, 14, 8, 4, 3, 10, 7, 2, 20, 17, 33, 18}
19	19 99 96 {42, 20, 14, 23, 5, 9, 1, 4, 10, 8, 3, 11, 2, 6, 12, 28, 17, 54, 45}
20	20 79 65 {18, 14, 25, 4, 10, 15, 7, 1, 9, 8, 22, 3, 19, 17, 11, 6, 5, 24, 32, 23}
20	20 89 71 {33, 15, 20, 10, 5, 21, 7, 8, 2, 17, 1, 9, 22, 38, 3, 12, 11, 6, 28, 23}
21	21 104 91 {23, 14, 19, 9, 5, 20, 11, 13, 28, 12, 8, 25, 7, 18, 4, 32, 29, 1, 31, 43, 30}
22	**22 88 74 {20, 17, 23, 3, 8, 6, 18, 5, 28, 15, 13, 14, 9, 19, 1, 12, 10, 2, 27, 36, 11, 25}**
23	**23 107 89 {43, 22, 23, 19, 4, 15, 21, 1, 17, 11, 8, 3, 12, 2, 18, 10, 58, 9, 49}**
24	**24 113 95 {17, 23, 9, 8, 34, 12, 1, 7, 10, 4, 3, 11, 39, 16, 15, 14, 24, 5, 19, 25, 45, 37, 6, 31}**
25	**25 118 106 {42, 23, 27, 19, 4, 26, 1, 25, 15, 17, 64, 12, 10, 7, 11, 8, 13, 3, 5, 22, 14, 18, 30, 6, 24}**
26	**26 143 120 {28, 22, 38, 6, 16, 34, 55, 24, 30, 23, 32, 58, 12, 11, 7, 4, 3, 17, 9, 1, 14, 13, 27, 8, 33, 25}**
27	**27 134 105 {31, 29, 36, 2, 27, 33, 38, 22, 14, 8, 19, 10, 28, 3, 7, 13, 1, 4, 11, 17, 5, 18, 39, 41, 30, 6, 24}**

Table 6.2: *The Gambinicode+s of the smallest SPSR for each order. In gray are the CPSRs which are smaller than the smallest SPSR in that order. In **bold** are the SPSRs that are also new discoveries themselves.*

Smallest SPSRs by Number of Boundary Squares

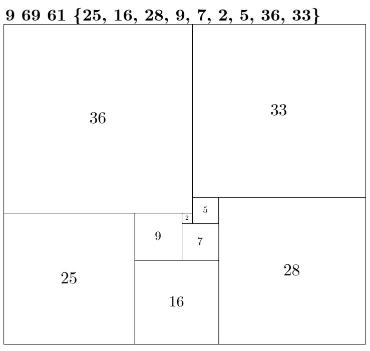


Figure 6.3: *The smallest SPSR with 5 boundary squares*

Throughout our literature review, we saw little research regarding the number of boundary squares, other than the proven minimums.

In subsection 2.2.2, we sought to construct an SPSR with only 5 boundary squares; the sixth smallest SPSR features this property, shown in Figure 6.3. In processing our results, we took note of the smallest possible decomposition for each number of boundary squares, shown in Table 6.3. Of the eight enumerated, four of them are also the smallest SPSR for their order.

Boundary Squares	Smallest Perfect Decomposition (Gambinicode+)
5	9 69 61 {25, 16, 28, 9, 7, 2, 5, 36, 33}
6	9 33 32 {14, 10, 92, 1, 8, 4, 7, 18, 15}
7	17 82 75 {32, 18, 15, 17, 3, 10, 2, 19, 14, 7, 6, 11, 1, 5, 43, 4, 39}
8	16 79 74 {45, 34, 11, 23, 29, 15, 12, 3, 7, 14, 4, 8, 10, 1, 17, 9}
9	17 84 60 {17, 19, 15, 2, 27, 13, 8, 21, 5, 16, 1, 4, 33, 3, 7, 23, 28}
10	21 104 91 {23, 14, 19, 9, 5, 20, 11, 13, 28, 12, 8, 25, 7, 18, 4, 32, 29, 1, 31, 43, 30}
11	**24 113 95 {17, 23, 9, 8, 34, 12, 1, 7, 10, 4, 3, 11, 39, 16, 15, 14, 24, 5, 19, 25, 45, 37, 6, 31}**
12	**23 139 100 {45, 35, 33, 26, 7, 19, 27, 13, 1, 18, 14, 10, 25, 55, 40, 23, 3, 21, 17, 6, 5, 16, 11}**

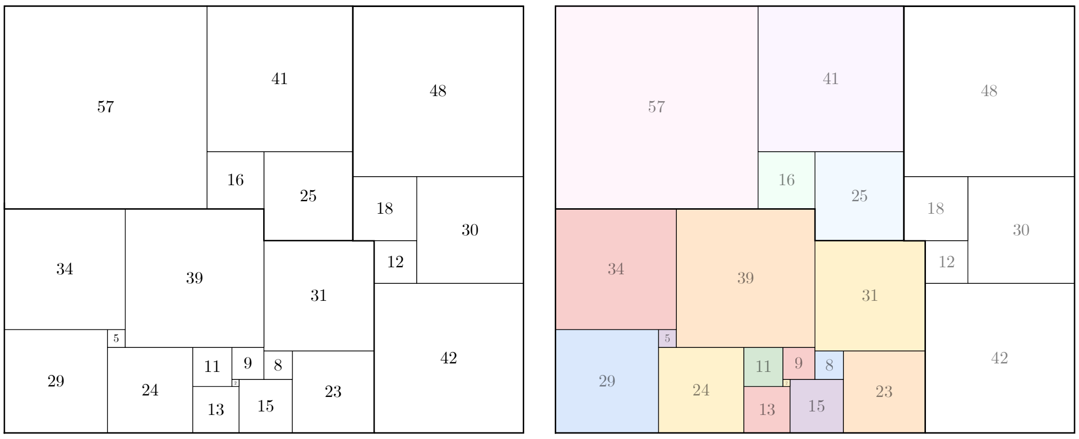
Table 6.3: *The Gambinicode+s of the smallest SPSR for each number of boundary squares. In* **bold** *are the SPSRs that are also new discoveries themselves.*

6.2.3 A Triply Special Triple Isomer

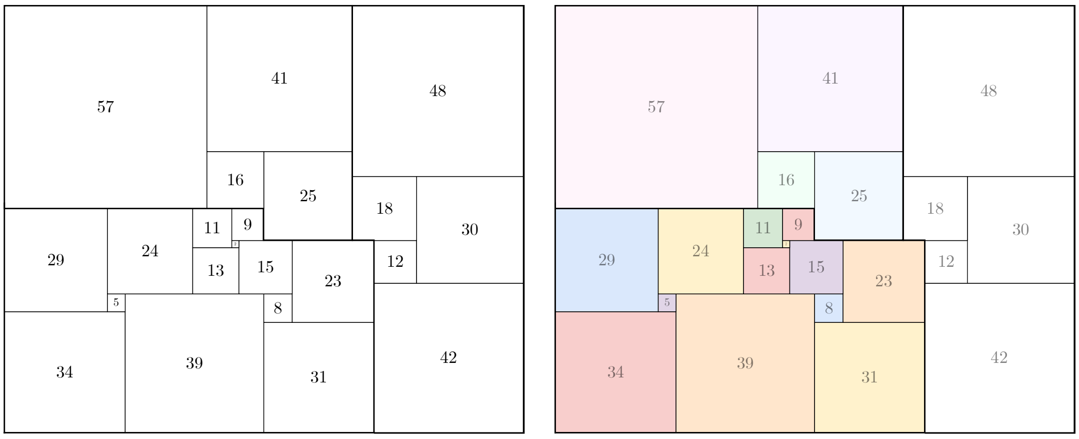
Perhaps the most exciting result of this project, shown in Figure 6.4, is a very special trio of SPSRs. These SPSRs are triple isomers: all three share the same set of constituent squares rearranged non-trivially. Among perfect squares *and* rectangles:

- This is a **new triple isomer**. It is the **ninth** to be discovered, among hundreds of millions of decompositions.
- We have proven it to be **smallest possible triple isomer** (by width).
- It is also the **first discovery of an isomer of order 22**.
  All pairs and triples of SPSRs up to order 21 have been exhaustively enumerated, this is the first discovery of an order 22 SPSR isomer (of *any* cardinality).

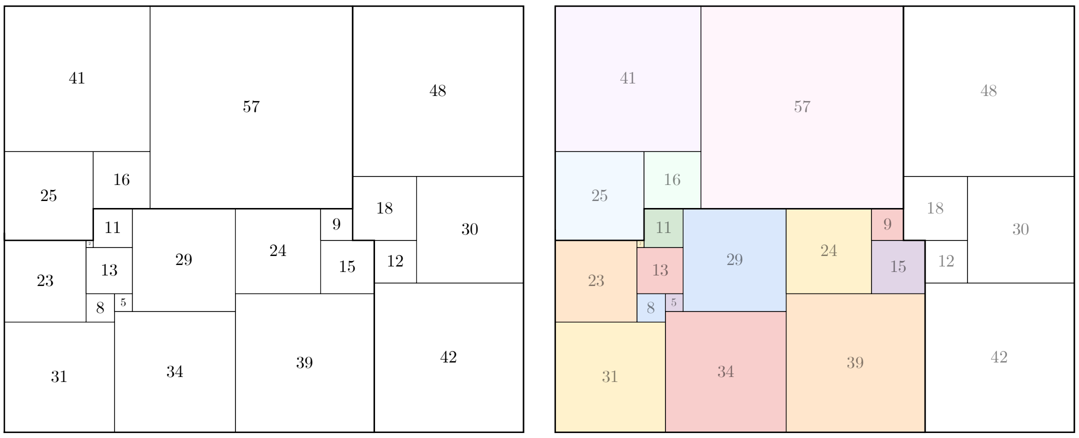
Individually, each of these properties would make any SPSR a notable discovery – the fact that each occur in a single triple isomer is extraordinary.



22: 146 120 **A** **22 146 120** {29, 24, 13, 15, 11, 2, 23, 9, 8, 42, 31, 12, 30, 5, 39, 34, 25, 18, 48, 57, 16, 41}



22: 146 120 **B** **22 146 120** {34, 39, 29, 5, 31, 42, 8, 23, 12, 30, 24, 13, 15, 11, 2, 9, 25, 18, 48, 57, 16, 41}



22: 146 120 **C** **22 146 120** {42, 39, 34, 31, 8, 23, 29, 5, 13, 11, 2, 25, 15, 24, 30, 12, 18, 9, 57, 16, 41, 48}*

Figure 6.4: *The* ***smallest triple isomer****, and the first order 22 isomer to be discovered. Each isomerism is shown plainly and in colour to visualise the rearrangement of the squares.*

In addition to the important discoveries about this new isomer, it has a quirky fourth property: In **A** and **B**, all of the squares have the **same horizontal position**; in **B** and **C**, all of the squares have the **same vertical position**.

This property comes from graph symmetry – certain symmetries in the underlying graph representation presented by [8]. This is not a property all Squared Rectangles and Squares have (or they would all have isomers) and not one that even many isomers have. Despite

*__22: 146 120 C__ is shown reflected horizontally to align squares visually with the other two, but the Gambinicode+ presented is the canonical arrangement (with the largest corner square in the top left) the fact we *know* there cannot be a fourth isomer (our search was exhaustive of size 146), it is a natural next question and an interesting experiment to attempt to construct one from the fourth combination of these coordinates: one with the vertical coordinates of the first isomer, and the horizontal of the third (the complement to the second one in this set). By nature of our algorithm, we know this is futile, but it is surprising to see how close it is to a valid decomposition:

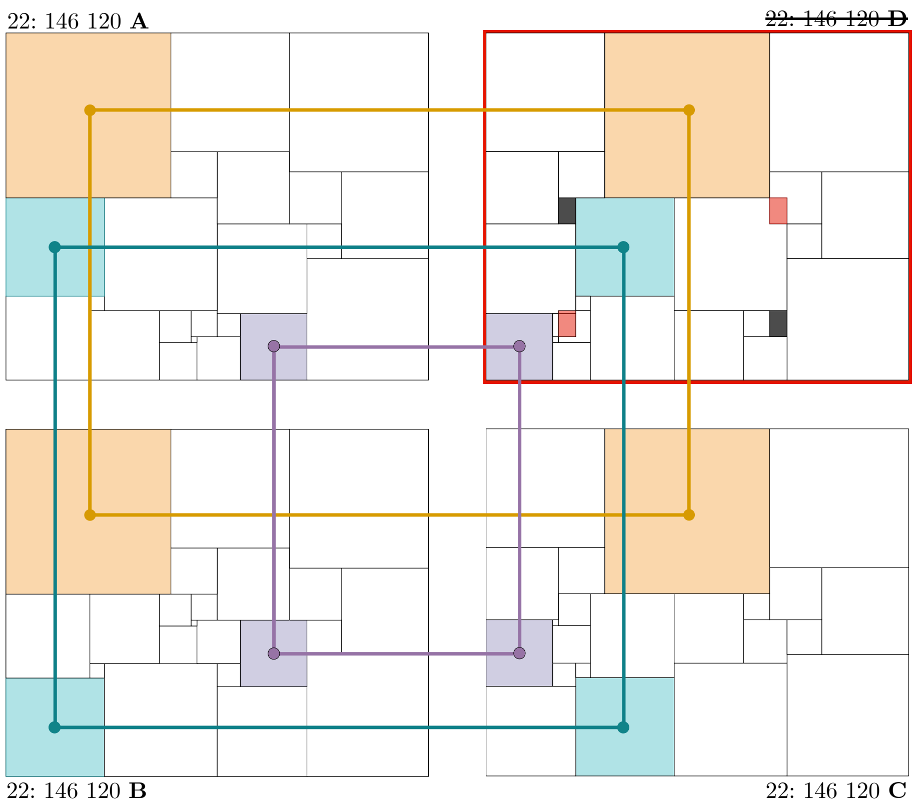


Figure 6.5: *A demonstration of the relationships between the triple isomers found, and a false fourth isomer. The left-right shows arrangements whose squares are vertically aligned, and top-bottom shows arrangements with squares horizontally. The fourth "isomer" (upper right) constructed from the last coordinate combination is invalid due to small areas of overlap and gap.* Shown only for 3 squares, **all** squares in these isomers exhibit this property.

A search was undertaken on a subset of other known pairs, triples, and quadruples: some pairs have these relationships (notably, the first ever isomer pair [18], which lead to the discovery of the underlying symmetry), and plenty of pairs within triples. While our search did not find any other known triples that are *all* related this way (or any quadruples), it is entirely possible others exist with this property – even if this were to be the case, it is striking to have seen another remarkable property in what may already be the most notable triple isomer.

Chapter 7

Conclusions

In this project, we built upon Ian Gambini's exhaustive algorithm for Squaring Squares and Rectangles by width. We implemented various mathematical and architectural optimisations, and leveraged concurrency to split the workload. We developed a **fundamentally restructured algorithm**, and saw a reduction from $O(e^{0.224})$ to $O(e^{0.148})$ in the exponential complexity of the algorithm.

The work exceeded expectations, reducing the total runtime by a factor of at least 475x. As a result, running on a M1 Pro over 9 cores for 263 hours, we enumerated all Perfect Squared Squares and Rectangles up to width 150, and made the following discoveries:

- An exhaustive enumeration of all SPSSs, SPSRs, CPSSs, and CPSRs up to width 150, presented with notable properties such as order and boundary squares.
- 66 undiscovered SPSRs.
- Proof of the 10 smallest SPSRs of orders 17-26.
- Proof of the 8 smallest SPSRs with 5-12 boundary squares.
- An undiscovered and remarkable isomer, shown in Figure 7.1:
  - The smallest possible, and ninth to be discovered, isomer triple.
  - The smallest possible, and first identified, isomer of order 22.
  - Demonstrates uncommon symmetry among isomers.
- A new bound on the number of 2-square edges a PSS can have.

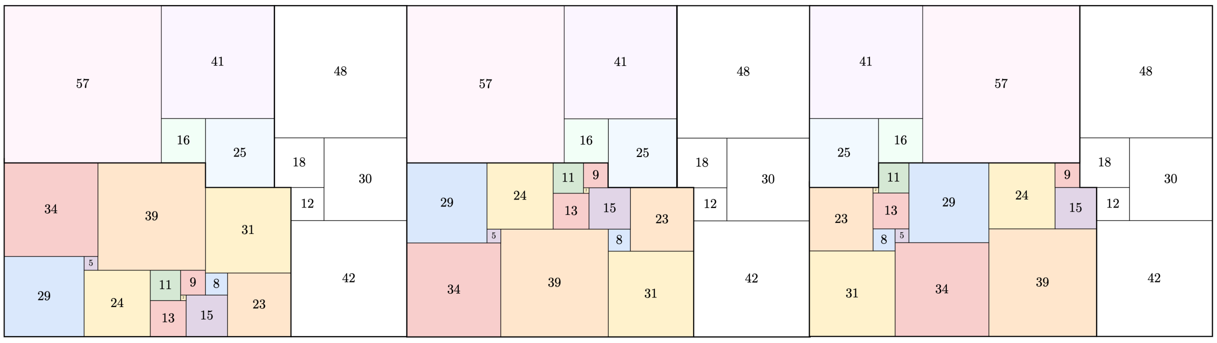


Figure 7.1: *The ninth ever discovery of a triple isomer, which is also the smallest possible triple, and the first identified of order 22.*

We also presented proof of a new bound, which was implemented in our search, but could also be used by others developing algorithms to search for or construct PSRs and PSSs. The new algorithm will be open source, along with the code to convert Gambinicode+ to diagrams and standard encodings so that results can be integrated into existing catalogues.

7.0.1 Future Work

Further efforts to reduce the algorithmic complexity of exhausting by size, particularly investigations into which sections of the work space are extravagantly wide, could aid the search for enumerating further smallest SPSRs by order. We also noted that there is disportionate respresentation of SPSRs whose edges differ only by 1 or 2 in these smallest SPSRs – further investigation into this specific class of SPSRs could yield at least new smallest known SPSRs by order.

Work could be done to efficiently parallelise the work outside of Apple Silicon to allow for greater parallelisation on stronger machines.

The new subshape proven to be the necessary subshape for decompositions with 5 boundary squares could be used to enumerate these, perhaps exhaustively by order or size, by a constructive or decompositional algorithm.

Much like Gambini's *first* algorithm fixed the number of plates to enumerate efficiently *many* but not all decompositions, we suggest that one could employ machine learning to prioritise promising branches and produce new decompositions. This could lead to many discoveries of smallest *known* rectangles by order or isometry.

Bibliography

[1] S. E. Anderson, "Compound perfect squared squares of the order twenties," 2013.

[2] A. M. Turing, "Proposed electronic calculator (1945)," Oxford University Press eBooks, p. 369–454, Apr 2005.

[3] I. Gambini, "A method for cutting squares into distinct squares," Discrete Applied Mathematics, vol. 98, p. 65–80, Oct 1999.

[4] I. Gambini, Quant aux carrés carrelés (in French). Doctoral thesis, L'UNIVERSITÉ DE LA MÉDITERRANÉE AIX-MARSEILLE II, Jan 2001.

[5] H. Dudeney, "Lady isabel's casket," The London Magazine 584, p. 7, 1902.

[6] M. Aigner and G. M. Ziegler, Proofs from THE BOOK. Springer Science Business Media, 1998.

[7] M. Abe, "Covering the square by squares without overlapping (in japanese)," Journal of Japan Mathematical Physics, vol. 4, no. 4, p. 359–366, 1930.

[8] R. L. Brooks, B. Smith, A. H. Stone, and W. T. Tutte, "The dissection of rectangles into squares," Duke Mathematical Journal, vol. 7, no. 11, p. 312–340, 1940.

[9] R. Sprague, "Beispiel einer zerlegung des quadrats in lauter verschiedene quadrate," Mathematische Zeitschrift, vol. 45, p. 607–608, Dec 1939.

[10] A. Hodges, Alan Turing : the enigma. Princeton, N.J.: Princeton University Press, Princeton, New Jersey, 2012.

[11] B. S. Bigham, M. Davoodi, S. Mazaheri, and J. Kheyrabadi, "Tiling rectangles and the plane using squares of integral sides," CoRR, vol. abs/2110.00839, 2021.

[12] K. Sakanushi, Y. Kajitani, and D. Mehta, "The quarter-state-sequence floorplan representation," IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, vol. 50, no. 3, pp. 376–386, 2003.

[13] X. Hong, S. Dong, G. Huang, Y. Cai, C.-K. Cheng, and J. Gu, "Corner block list representation and its application to floorplan optimization," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 51, no. 5, pp. 228–233, 2004.

[14] A. Notices, “Interview with martin gardner,” The College Mathematics Journal, vol. 40, p. 158–161, May 2009.

[15] S. Anderson, “Squaring.net.”

[16] D. Moew, “David moews’s home page,” Mar 2020.

[17] J. D. Skinner, “Uniquely squared squares of a common reduced side and order,” Journal of Combinatorial Theory, Series B, vol. 54, p. 155–156, Jan 1992.

[18] A. Duijvestijn, Electronic computation of squared rectangles. Jan 1962.

Appendix A

Complete Results

A.1 Complete runtimes for all milestones

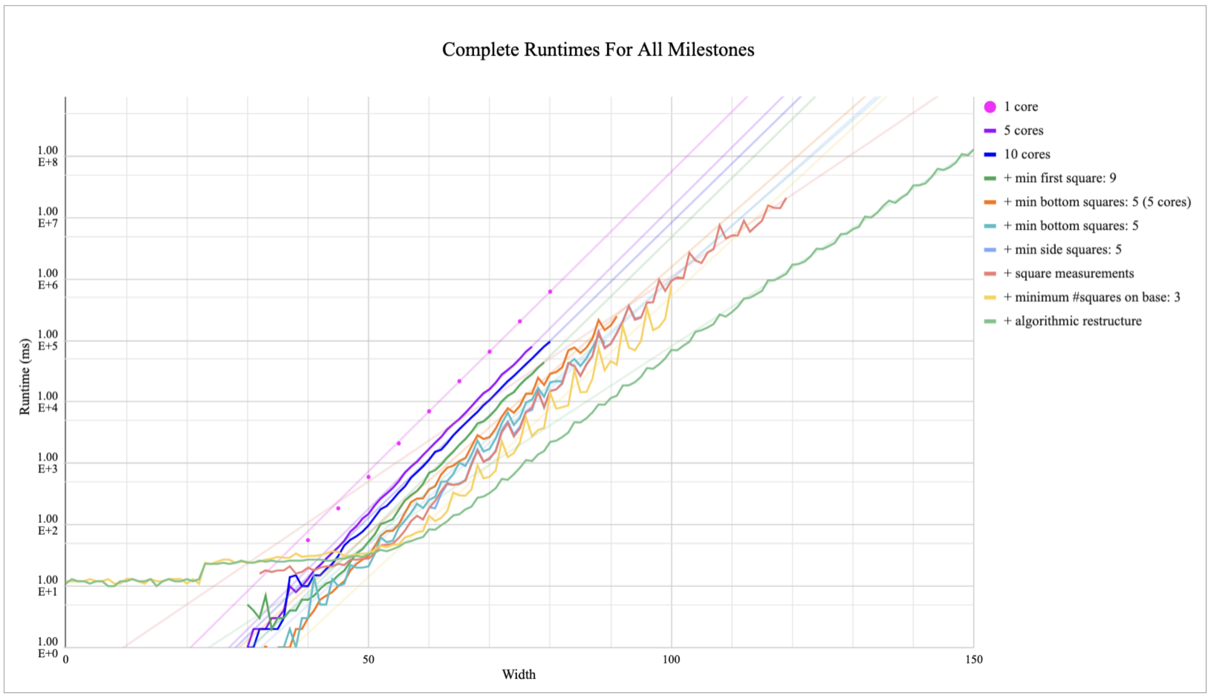


Figure A.1: *A graph showing all the runtime data for all milestones. During analysis, overlapping domains that were large enough to have negligible noise were used to compare each milestones.*

A.2 SPSRs by Width and Order

	order																			
size	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**sum**
33	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
57	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
65	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
69	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
75	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	**1**
79	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	**1**
81	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
82	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**1**
83	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
84	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**1**
88	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	**2**
89	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	**2**
91	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	**1**
94	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**1**
97	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	**2**
98	0	0	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
99	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	**2**
100	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	**1**
101	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**1**
103	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	**3**
104	0	0	0	0	0	0	0	0	0	1	1	2	1	0	0	0	0	0	0	**5**
105	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	**4**
106	0	0	0	2	1	2	1	0	0	1	0	0	0	0	0	0	0	0	0	**7**
107	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	**2**
109	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	**2**
111	0	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	**4**
112	0	0	1	1	5	0	1	0	2	1	0	0	0	0	0	0	0	0	0	**11**
113	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	**4**
114	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	**1**
115	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	**3**
117	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	**1**
118	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	**4**
119	0	0	0	0	0	0	0	1	0	0	1	2	0	0	0	0	0	0	0	**4**
120	0	0	0	2	1	0	2	0	0	0	1	2	0	0	0	0	0	0	0	**8**
121	0	0	0	0	0	0	1	0	0	0	3	0	0	0	0	0	0	0	0	**4**
122	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	**3**
123	0	0	0	0	0	0	0	0	0	0	1	2	0	1	0	0	0	0	0	**4**
124	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	**1**
125	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	**4**
126	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	0	**3**
127	0	0	0	0	0	0	2	0	0	0	1	0	0	0	0	0	0	0	0	**3**
128	0	0	0	0	0	0	1	0	1	2	0	1	0	2	1	0	0	0	0	**8**
129	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	0	0	**4**
130	0	1	0	0	0	0	0	0	0	1	0	2	1	0	0	0	0	0	0	**5**
132	0	0	0	0	0	0	0	0	0	2	0	0	1	1	0	1	0	0	0	**5**
134	0	0	0	0	0	0	1	0	1	1	0	1	1	2	0	0	1	0	1	**9**
135	0	0	0	0	0	0	1	0	1	2	1	0	0	1	0	1	0	0	0	**7**
136	0	0	0	0	0	0	0	1	2	3	0	1	0	1	0	0	0	0	0	**8**
137	0	0	0	0	1	0	0	1	0	0	1	1	2	0	1	0	0	0	0	**7**
138	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	**3**
139	0	0	0	0	0	0	1	0	0	0	2	1	0	2	2	1	0	0	0	**9**
140	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	1	0	0	**5**
141	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	**4**
142	0	0	0	0	0	0	0	0	0	1	1	0	0	0	2	0	0	0	0	**4**
143	0	0	0	0	0	1	0	2	0	1	0	1	1	0	1	0	0	1	0	**8**
144	0	0	0	0	0	0	0	1	2	0	7	0	2	1	2	1	2	0	0	**18**
145	0	0	0	0	1	5	0	0	0	2	0	1	1	1	1	1	2	0	0	**15**
146	0	0	0	0	0	0	0	0	0	1	0	3	1	4	0	0	0	0	0	**9**
147	0	0	0	0	0	0	0	0	1	1	0	0	4	0	0	0	0	0	0	**6**
148	0	0	0	0	0	0	0	1	0	1	1	0	1	6	1	1	0	2	0	**14**
149	0	0	0	0	0	0	0	2	0	2	0	4	4	0	2	0	0	0	0	**14**
150	0	0	0	0	0	0	0	1	0	1	2	0	2	1	1	1	1	0	0	**10**

Table A.1: *The complete enumeration of SPSRs up to width 150.*