Universiteit Leiden
The Netherlands

Opleiding Informatica

Polyomino Packing
using a Tetris-like Ruleset

Max Blankestijn

Supervisors:
W.A. Kosters & F.W. Takes

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

24/07/2019

Abstract

We look at a version of the Polyomino Packing problem in an environment not unlike the game Tetris. We will explore methods to pack a given set of polyominoes into any given target shape using the ruleset provided here. The direction from which pieces may be dropped is expanded, but no row removal will be possible.
We discuss exhaustive methods and ways to prune them based on unsolvable board states, eventually discussing non tileable shapes using polyominoes. There is also some exploration of a Monte Carlo method and effects of altering the order in which polyominoes are supplied. At the end we leave some methods open to be explored, including an approach that could solve parts of the target separately and an approach that would solve the problem by reversing it.

Contents

1 Introduction

A polyomino is a polyform constructed from squares that are all adjacent to at least one other square in the polyomino. It is built by joining a certain amount of them edge-to-edge. Polyominoes (here often referred to as pieces) may be classified by their size; the amount of squares needed to construct it. Figure 1 contains the sets $PS(4)$ and $PS(5)$, the tetrominoes and pentominoes respectively, figure 2 contains $PS(6)$, the hexominoes, and Figure 3 contains a few elements of $PS(10)$.

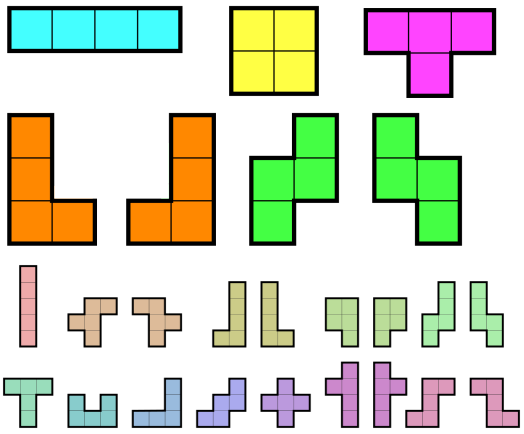


Figure 1: Tetrominoes ($PS(4)$) and Pentominoes ($PS(5)$). Source: https://en.wikipedia.org/wiki/Polyomino

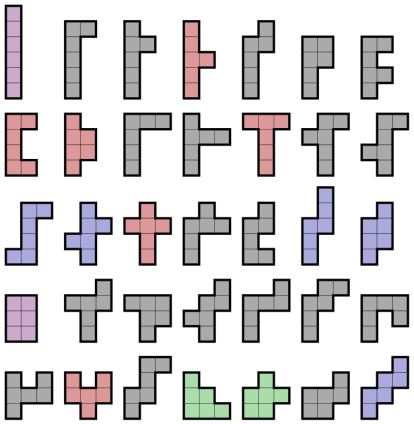


Figure 2: Hexominoes ($PS(6)$). Source: https://en.wikipedia.org/wiki/Polyomino

This thesis will concern itself with a 2D-Tiling Problem using polyominoes. The goal will be to put pieces on our *Board*, chosen from a set of given polyominoes in such a way that at the end the *Board* will have the same tiling as a given *Target Image*. The rules for placing pieces can be found in Section 1.3, definitions for the *Board* and *Target Image* are provided in Section 2.
The accepting state of the problem will consist of a list of *Moves* that will result in the desired two dimensional tiling if it can be constructed using the provided polyominoes, otherwise we reject.

1.1 Type of Problem

A general tiling problem concerns itself with placing certain given objects into a given shape or container such that none of the objects overlap and there are no gaps between them. A polyomino tiling, or packing, problem uses the two dimensional shapes known as polyominoes as objects to pack into a shape. Current studies of polyomino tiling generally focus on tiling a rectangle with congruent polyominoes. The problem in this thesis seeks to produce a tiling of an arbitrary set of polyominoes in an arbitrary container shape.
Tiling is not unrelated to the Bin-Packing Problem where a given set of objects must be fit into one or more containers such that a minimal number of containers is used.

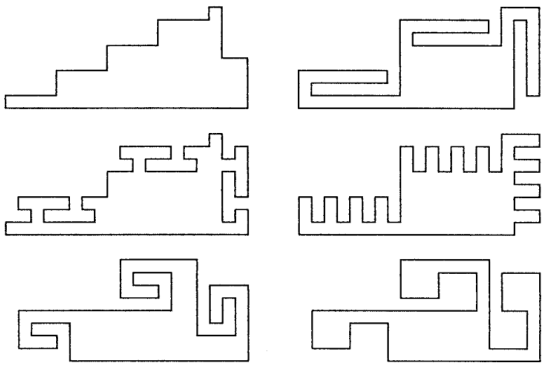


Figure 3: Some polyominoes of size 10 [Mar97].

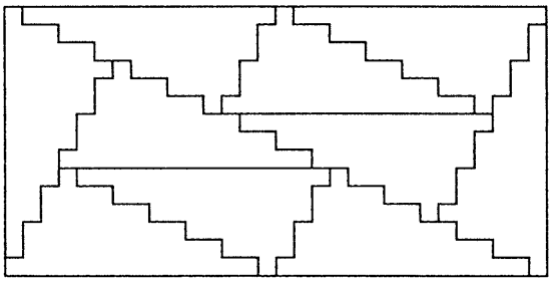


Figure 4: Fitting a big polyomino in a rectangle [Mar97].

1.2 Polyominoes used

For an instance of this problem a set of polyominoes is given. This thesis will largely concern itself with given sets containing one size of polyomino. The concepts discussed can be extended to multiple polyomino sets begin provided.

Large piece sizes greatly increases the amount of pieces available for any move (Table 1). Many of these larger sizes could have a reduced flexibility in regard to fitting them together. When looking at some polyominoes of size ten in Figure 3 can only be perfectly fitted together with very specific other pieces and given the large amount of them available it is highly unlikely for any one of them to be that perfect fit. This could make it take a long time for any valid move to be made. Some of those pieces even create places that are not reachable using this rule set. These would only be useful if the given T requires a shape that exactly matches such a piece. This again is highly unlikely, any such piece would be continuously considered but almost always be useless. If it takes a long time to find a piece to place and continue this can make it take longer to find a solution and the great amount of them available also greatly increases the search time on its own.

Because of this we will mostly be looking around size four. Intuitively the tetrominoes seem to be a good baseline for our packing problem. They are not overly complex but their shape still poses a challenge and they have variation. When we move to smaller sizes it becomes easier to fit the pieces into any shape.

A size of three has some decision making but not much, two seems to make tiling much less interesting with only two ways to place a piece. Tiling with pieces of size one becomes completely trivial.

Because of this this thesis will be using the tetrominoes only, giving $Pieces = PS(4)$ and $Piece_{small} = 4$.

1.3 Piece Placement

This problem does not generally allow the placement of pieces at any coordinate like one would in a puzzle. Instead it follows a similar principle to Tetris where pieces are dropped and then fall into a place on the board. We do differ from Tetris by limiting some parts of the placement but expanding some others.

Size	No. Fixed Polyominoes	No. Moves
1	1	4
2	2	8
3	6	24
4	19	76
5	63	252
6	216	864
7	760	3,040
8	2,725	10,900
9	9,910	39,640
10	36,446	145,784
11	135,268	541,072
12	505,861	2,023,444
13	1,903,890	7,615,560
14	7,204,874	28,819,496
15	27,394,666	109,578,664
16	104,592,937	418,371,748
17	400,795,844	1,603,183,376
18	1,540,820,542	6,163,282,168
19	5,940,738,676	23,762,954,704
20	22,964,779,660	91,859,118,6401

Table 1: Number of fixed polyominoes per size [Jen09].

1.3.1 Initial Placement

When faced with an empty board any available piece may be placed on any pair: $(row,\ column) \in Board$. Initially we allow the placement of one piece that does not belong to P. This piece may of any size smaller than p. This will allow more shapes to be made in the case that the given sizes do not fit the amount of squares in the target.
If multiple sets are given any piece size in between two given sets can be recreated by p with and any piece of $size < p$. This means only pieces smaller than p would need to be created. Allowing the usage of these "extra" pieces removes the obvious constraint of not being able to create a shape simply because the amount of squares that are needed by the target cannot be constructed by combining the given pieces.

1.3.2 Subsequent Dropping

For the other piece placements we will be using a dropping system. Pieces are placed using the following method:

1. In Tetris pieces are always dropped from the top of the board and they fall down. We allow this but additionally we allow a similar dropping from the left, right and bottom of the board. The piece would then move into the direction opposite to which it was dropped.

2. Instead of a row and column we only select a column. When choosing to drop a piece from the side this would technically be a row from the original perspective.

3. A placement will stop if:

   (a) It encounters a square filled by another piece. This is only the case if during the falling it would move onto a square that is already filled by another piece. So not if it moves alongside another piece. That movement of the piece is then cancelled and the piece is placed. We record this and move onto the next move.

   (b) It encounters the side of the board i.e., it would go out of bounds. The move can be attempted but it is an immediate failure and the piece is removed from the board if it encounters an edge.

Since moves that attempt to fill squares not in the *Target Image* immediately indicate a non-accepting state these are not worth making. Such a move is considered invalid as if it was unable to be placed at all.
These rules notably differs from traditional Tetris rules. A piece cannot be controlled after it drops like in Tetris. After one is dropped in a certain column it will move straight down (or left, or right, or up) until it is stopped. Moreover there exists no rule that makes a row disappear once every square of it has been filled.
These two restrictions in comparison to Tetris do seem to limit the options presented. To contrast this the falling-directions have been expanded.

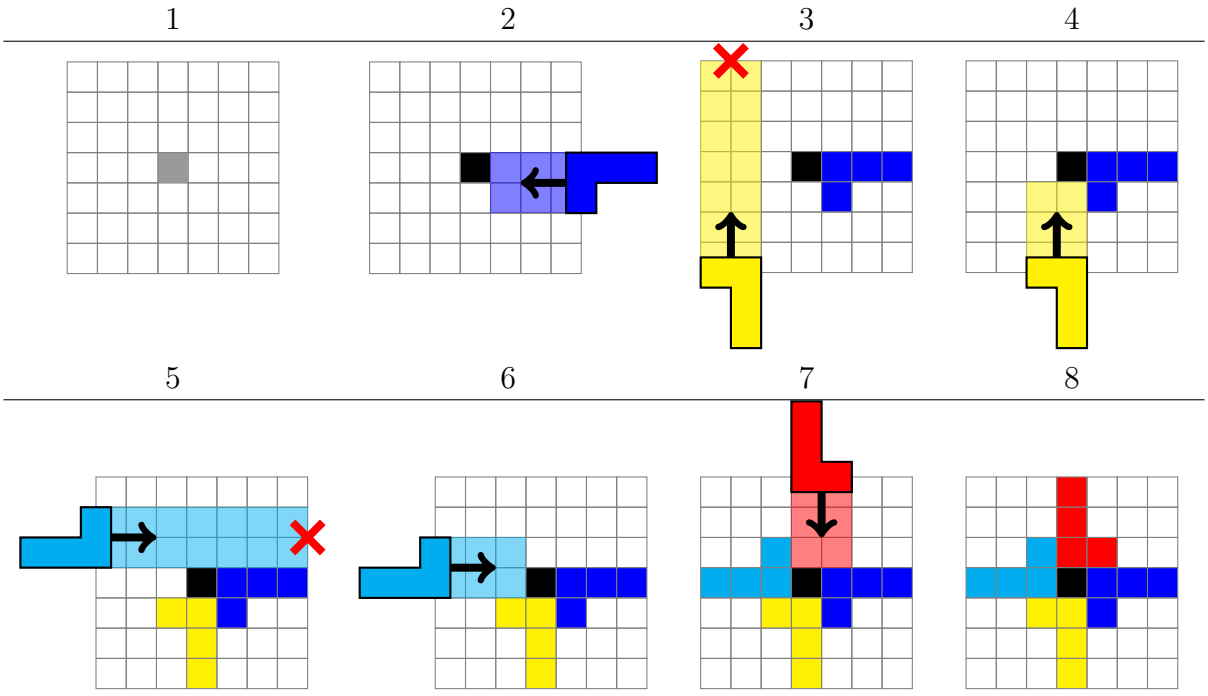


Figure 5: Sequence of moves made. Move 1 is the initial move, where we allow a monomino to be placed in the middle. Moves 3 and 5 miss the currently placed pieces and fail.

1.4 Methods

Several methods are explored in this thesis in regards to solving the proposed problem. They will mainly be different methods of exploring possible moves, but also some investigation into the effect of ordering the input sets.

1.4.1 Brute Force Method

For a first attempt and as a benchmark we introduce an algorithm that simply exhausts all possible moves that can be made.
This would give $|I|$ possible pieces for an initial move. They may be placed on at most, but often less than, mn places.
For each of these moves there will be $|Pieces|$ pieces that can be dropped from every column, from up or down, or every row, from left or right. This means there are up to $2m + 2n$ moves for each piece.
The amount of these moves that can be made is bounded by $|Target|$, the size of the chosen initial piece i and $Pieces_{small}$. There can be $\frac{|T|-i}{Pieces_{small}}$ moves at most. This is $\frac{|T|}{Pieces_{small}} - 1$ if the initial piece was of size p. Both of these situations may be described by $\lceil \frac{|T|}{Pieces_{small}} \rceil - 1$.
This bounds the amount of subsequent moves after an initial move by $((2m+2n)\cdot|Pieces|)^{\lceil |T|/Pieces_{small} \rceil - 1}$ and provides the following upper bound for the total amount of moves to explore:

$$Imn \cdot (2P(m+n))^{\lceil T/(p-1) \rceil} \tag{1}$$

Where $P = |Pieces|$, $p = Pieces_{small}$, $I = |I\text{-}Pieces|$, and $T = |Target|$.

1.4.2 Pruning Method

Since brute forcing does have a 100% success rate it can be of worth to improve on it. This method will explore some branches of the algorithm that has no solution on it. If these can be detected they may be used to identify board states that can be pruned. This method will detail two of these situations and propose functions to detect them. The original brute force method is than modified to stop searching along a branch that includes them. The exact method for this is described in Section 4.2.

1.4.3 Improved Pruning Method

In the original pruning method there are instances where the algorithm takes longer than in others. When investigating the board states during and after runtime there seemed to be certain situations that could naively be considered a pruning opportunity and according to the pruning functions they were. On closing investigation there were certain squares that seemed reachable but where not. There also seemed to exist certain shapes that should be able to be created when looking at the size of the shape and the size of polyominoes available, but are actually unbuildable.
Section 4.3 proposes adjustments to the original pruning functions to detect these instances. They may however be costly. This section will also attempt to explore and define non-trivial shapes that cannot be created by combining polyominoes of a given size.

1.4.4 Monte Carlo

This approach will use a version of the Monte Carlo method. This will involve the evaluation of every move possible at a certain board state through a semi random method. The best move according to this will be added to the solution and the algorithm continues on this board state. This is done until a solution is found or we are considering an unsolvable state.
This algorithm includes no form of backtracking once a move is made that creates an unsolvable state there is no way for the algorithm to revert this. Paired with the fact that the evaluation method will contain a random component there is a chance for a non-optimal move to be chosen to be part of the solution or a wrong one that can result in the algorithm terminating in a non-accepting state while an accepting one may exist.
A metric for the random evaluation will be proposed in Section 4.4, partly using the pruning functions already present.

1.5 Thesis Outline

Section 4.1 we will be discussing the implementation and results of the Brute Force Method, Section 4.2 will discuss the Pruning Method, Section 4.3 will expand on the previous and detail an Improved Pruning Method, and Section 4.4 looks at a Monte Carlo implementation. Additionally Section 4.6 will explore the concept of an algorithm that uses a different perspective of the same problem. This does not have an implementation as of yet.
This thesis was made for the bachelor program for Computer Science at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University. It was supervised by Dr. W.A. Kosters and Dr. F.W. Takes.

2 Definitions

This section defines certain concepts regarding this problem that are used in the thesis.

2.1 Board States

These will be definitions concerning the state of the board and the desired board state.

***Definition* 1.** $\mathit{Board} = \{(i, j) \mid 0 \leq i < m \land 0 \leq j < n\}$

Pieces may be placed on a field, it is an $m \times n$ grid consisting of *squares*. Each square can be either empty or filled by a placed piece. All squares on the board are initially considered empty.

***Definition* 2.** A square $s \in \mathit{Board}$ is considered "filled" $\iff s \in \mathit{Current\ Image} \subseteq \mathit{Board}$.

When a move is "made" the squares corresponding to the given polyomino at the given location are added to the *Current Image*.

***Definition* 3.** Given a set of polyominoes *Pieces* and a set of squares $\mathit{Target\ Image} \subseteq \mathit{Board}$ $\mathit{Problem\ Accepts} \iff \mathit{Target\ Image} = \mathit{CurrentImage}$

The *Target Image* is as set of different squares that must be filled after a certain amount of moves. It describes the shape we want to construct. The board only ever needs to be large enough to encompass the *Target Image* and as such the size is determined by it.
The target does have constraints. Because we can only place pieces by connecting them to other pieces a given target is only able to be constructed if it is interconnected. The target itself should be a single polyomino of arbitrary size.
Since any piece that fills a square outside of the target is always a fail state the only valid moves that are allowed to be made are moves that fill only squares in the target. Because of this any square that has been filled already also belongs to *Target Image*.

2.2 Illustration Example

Figure 6 and 7 show how board states will be represented in this thesis. It contains examples of the terms from Section 2.1. It is an image that is to be created using tetrominoes. The following shorthands will be used to describe it: $T = \mathit{Target\ Image}$, $C = \mathit{Current\text{-}Image}$, $P = \mathit{Pieces}$ and $I = \mathit{I\text{-}Pieces}$

- White squares denote squares that have not been and should not be filled:
  $\mathit{Board} - T - C$.
- Green squares denote squares of the target that are not yet filled:
  $T \cap \neg C$.
- Black squares denote C. Because we only allow placement of squares that cover T it more specifically describes $C \cap T$.

- This means $T = green \cup black$.

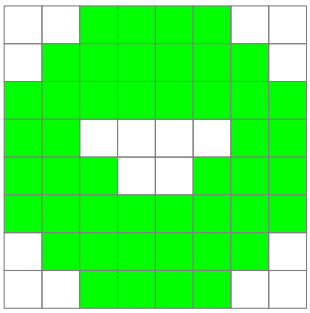


Figure 6: Example of a board state when trying to create a circle with a hole.

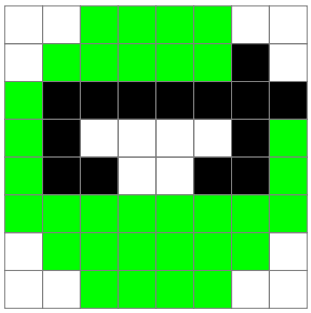


Figure 7: The same target image as in Figure 6 but the board is partially filled.

2.3 Pieces

These will be definitions regarding the polyominoes we use for placing.

***Definition* 4.** For any given size $s \in \mathbb{N}$, the polyomino set $PS(s)$ contains all polyominoes of size s. Here every rotation or reflection is considered a separate polyomino.

Polyominoes will be grouped and identified by their size as described above.

***Definition* 5.** $Pieces = \bigcup_{i=1}^{n} PS(L_i)$, where each L_i is a positive integer. They describe the different sizes of polyominoes present.

The set of polyominoes that may be used for moves will be called the *Pieces*. It can contain polyominoes of multiple sizes, but always contains the entire polyomino set if a certain size is included. This set is provided as input for an instance to the problem.

***Definition* 6.** $Pieces_{small} \in \mathbb{N}$ denotes the size of the smallest polyomino set available in *Pieces*, $Pieces_{small} = min(L)$.

This specific number will be useful in future definitions. Note that in this specific instance of the problem $Pieces_{small}$ describes the only size available in *Pieces*. This ensures the upper bound provided in Section 4.1 holds in the case multiple sizes exist in L.

***Definition* 7.** *I-Pieces* denotes the set I of pieces that may be used as the initial move. $I = \bigcup_{i=1}^{Pieces_{small}} PS(i) \cup Pieces$.

The sets of pieces used for the initial move are generated based on the *Pieces*. They consist of the *Pieces* and polyomino sets of any size smaller than the smallest available in *Pieces*.

In Figure 6 and 7 the following may be observed about the board state:

- This figure has to be filled with only tetrominoes so $P = PS(4)$. This gives $Piece_{small} = 4$.
- $|T| = 46$.
- $I = \bigcup_{i=1}^{3} PS(i) \cup Pieces = PS(1) \cup PS(2) \cup PS(3) \cup P$.

2.4 Pruning

The definitions in this section will define certain situations that may occur on a board state. These will mainly be used in the description of the pruning method.

***Definition* 8.** Given a square $s \in Board$ and a set of squares $S \subseteq Board : LOS(s) = \{d : d \in \{Up, Down, Left, Right\} \wedge \nexists x \in S$ on a linear path from s into direction $d\}$.

***Definition* 9.** A square $s \in Board$ is a *hole* in a polyomino described by $S \subseteq Board \iff LOS(s, S) = \emptyset$. Any square that is a *hole* in the *Current Image* cannot be filled with any future moves.

After a piece is dropped there is no movement while the piece is dropping. There are no ways to remove a square through a move after it has been placed. This means that any square that is blocked of on all four sides can never be reached through any series of moves. These squares will be referred to as the *holes*.

***Definition* 10.** A square $s \in Board$ is an *Unreachable Hole* $\iff s \in$ *Target Image*$\wedge$ s is as a *hole*.

This is a subclass of the holes that indicate an unsolvable board state. They are required to be filled so we need some move to reach them, but none exists. Figure 8 provides an example of holes and unreachable holes on a board state. This state can no longer result in an accepting state.

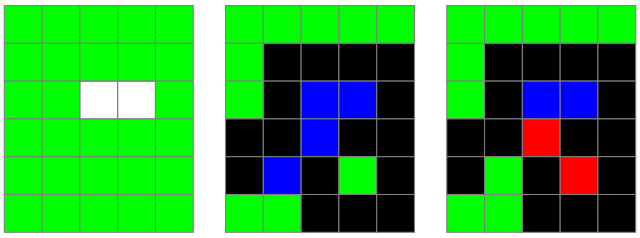


Figure 8: A *target*, partially filled *target* with four holes (blue) and the same partially filled *target* but with the unreachable holes in red.

***Definition* 11.** A **Gap** is a maximal set of squares $G \subseteq T - C : G$ describes a polyomino. Here $T = Target\ Image$ and $C = Current\ Image$.

The gaps will denote parts of the board that are not yet filled. Informally a gap is a collection of squares that are yet to be filled, completely surrounded by either filled squares or squares that must not be filled.

***Definition* 12.** A **gap** $G \subseteq \neg C \cap T$ is called *Untileable* $\iff$ FIT(—G—, P) = *false*.

The gaps also have a specific type that indicates a board state that cannot be solved. When they do not contain an amount of squares that can be filled with the given pieces. Figure 9 has examples of a gap and the splitting of a gap into multiple that are untileable.. The last board state cannot result in an accepting state.

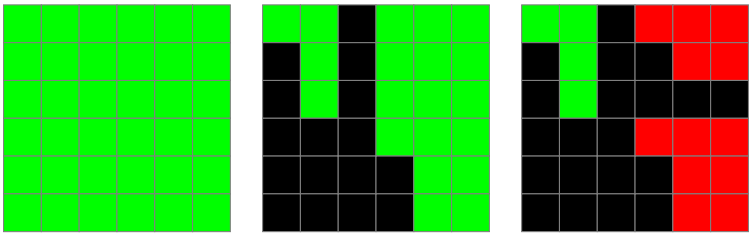


Figure 9: A *target* and partially filled *target* with *gaps* in green. The third contains one more move compared to the second and the *gaps* into two *gaps* that are both *untileable* (red).

3 Related Work

This section will list some previously done work that is related to the proposed packing problem or polyominoes.

3.1 Frobenius Coin Problem

The Coin Problem asks what is the greatest amount that cannot be created by a conical combination of a given series of n numbers $a_1, a_2, ...a_n$ where their greatest common denominator is 1. In other words the greatest b to which there is no solution to the equation $\sum_{i=1}^{n} a_i \cdot x_i = b$. Sylvester [Syl82] has defined this b, known as the Frobenius number for $n = 2$ as $b_{frobenius} = (a_1 - 1)(a_2 - 1) - 1$. Formulae exist in the case $n = 3$ for any a_1, a_2, a_3, [Tri17]. While there is no universal method for any n, it has been proven that for any fixed n there exists a polynomial time algorithm [Kan92].

3.2 Creation of any (reasonable) Tetris-configuration

Hoogeboom and Kosters [HK04] have shown that every configuration of squares, pertaining to some limitations, may be constructed using TETRIS rules for placement using a simple parity argument. We modify original TETRIS rules, most notably removing the removal of filled rows and adding multiple directions. But this limitation seemingly cripples this ability to create any shape, even with the added directions.

3.3 Counting Polyominoes

The highest count we are aware of at the moment was reached by Jensen [Jen03], where he reached up to a size of 56. The total count from size 1 to 56 can be found on his website [Jen09].
The amount of polyominoes that exist for a given size has an exponential growth [Kla67]. The

growth constant λ is not precisely known. With the latest counts Jensen also updated the bounds for λ: $3.927378 \leq \lambda \leq 4.0625696$ [Jen03].

3.4 Packing Congruent Polyominoes into Rectangles

Liu [Liu03] shows what rectangles can be packed with some polyominoes. Marshall [Mar97] summarizes discoveries of what rectangles may be packed notably for higher order polyominoes. BodLaender and van der Zanden [BvdZ18] provide several bounds for complexities of polyomino packing based on whether the target shape is a $2 \times n$ rectangle, a rectangle of area n and if tiling a rectangular area with rectangular polyominoes. Most notably they show that Polyomino Packing can be solved in $2^{O(\log n)}$ for any target shape.

4 Experiments

The experiments run for several target sizes: starting at 5, with increments of 5, until 50. For each target size we run 100 targets and we track how many moves were considered, how many were actually placed, how many times the value of a square was looked up, what the success rate was, and how many times the algorithm timed out in the 100 targets.
The targets were generated through a simple method. It starts out at one square. It adds all current neighbours to a pool and picks one to add to the target from this randomly, the neighbours of the recently added are then also added. This is done until the target has a desired size.
In order to save time we implement a time limit on every run, this will be 420 seconds.
Most figures and tables resulting from these experiments may be found in the Appendix A.

4.1 Brute Force

The method supplied for the brute force test simply iterates through all pieces, (rows for the initial move,) columns and falling directions. It has a trivial optimization: $PS(s)$ is only used if $s \bmod |\mathit{Target\ Image}| = 0$. Otherwise it attempts every move possible until a solution is found or no more moves are available.

4.1.1 Results

Brute Force has a good success rate for the targets it has run. It is however very slow, as can be seen in Figure 26 in Appendix A where it already starts timing out on size 35 and in Figures 27 and 29 where it already does more work than the other algorithms on higher target sizes.
Because of the rapid increase in runtime it has proven impractical to run Brute Force for target sizes higher than 25 and the lower quality of runtime compared to the other algorithms has already been shown.

4.2 Brute Force with Pruning

This algorithm is similar to the brute force method. It iterates through all moves in the same fashion. But when a board state is considered prunable the move that created is removed and the next move is considered instead.

The cases that may be pruned are detailed below, along with methods that detect them. After a piece is placed all squares around and within its bounding box are possibly affected in some way. This includes the gaps these squares belong to. Essentially all squares in the gaps adjacent to the just placed piece will have to be evaluated.

4.2.1 Unreachable Holes

The *unreachable holes* from Section 2.4 pose a problem since all squares in T must be filled in order to accept, but by definition any square that is a hole cannot be. For any branch that contains an *unreachable hole* it is not useful to further investigate further.
The pruning function would determine the LOS, as defined in Definition 8, for all squares and detect a pruning situation if for a square s, $LOS(s) = \emptyset$.

4.2.2 Determining Fillability

Algorithm 1 on page 13 provides a potential method to determine if a number of squares could be filled through combination of the available sizes. Any recursive call is passed "$\rho - set$" (line 13) and any call where $\rho = \emptyset$ immediately returns false, if there is only one set of polyominoes in P this algorithm reduces to a Boolean function that calculates "$\delta \bmod size = 0$" and then returns *false* if this is not the case. Where "*size*" is the only element of ρ. If there are multiple sizes of polyominoes the algorithm evaluates "$\delta \bmod size = 0$" (line 9) for one of the sizes and attempts to fit the other sizes in the remaining gap, if there is no remainder we evaluate "true" (line 13).
We try to fit the remainder using the same function but without the current set (line 13), since we already determined we could not fill the gap with only the current set. Afterwards we "take out" a piece in line 15 and retry with a larger remainder, eventually considering all combinations of pieces in this fashion.

The solutions of the coin problem mentioned in Section 3.1 could improve on this exhaustive method. Any value of $\delta > b_{frobenius}$ would automatically be tileable. The calculations are however specific to distinct n and do not work in a general case and the polynomial time algorithms also only work for specific n.

4.2.3 Unfillable Gaps

While it is determined at the start of the algorithm if there are enough squares so the target can be filled, after creating one or more gaps this is not necessarily the case for every one of them. If the remaining *target* is split into multiple gaps by the placed piece a board state may be reached such that $\textsc{fit}(|T|, P) = true$ (Algorithm 1) but for a certain gap $G_i \subseteq \neg C \cap T$, $\textsc{fit}(|G_i|, P) = false$. See Figure 9 in Section 2.4 for an example of this.
When going through all affected squares the size of every gap encountered is counted along with it. If such a gap has a count c such that $\textsc{fit}(c, P) = false$ we detect a prunable state.

4.2.4 Dropping Bounds

After the initial move any piece may only be placed if it collides with an already placed piece. At the start of the packing there will be many moves that are not worth attempting since several rows

Algorithm 1 Can $\delta \in \mathbb{N}$ be constructed through combination of the list of sizes $\rho \subseteq \mathbb{N}$

```
1: function FIT(δ, ρ)
2:     if δ = 0 then return true
3:     end if
4:     if ρ = ∅ then return false
5:     end if

6:     for all set ∈ ρ do
7:         size ← set

8:         if δ ≥ size then
9:             remainder ← δ mod size
10:            if remainder = 0 then return true
11:            end if

12:            while remainder ≠ δ do
13:                if FIT(remainder, ρ \ set) then return true
14:                else
15:                    remainder ← remainder + size
16:                end if
17:            end while
18:        end if
19:    end for
20: end function
```

and columns are empty. By tracking the horizontal and vertical bounds of the currently placed pieces we can ensure only moves that will collide are made. It can simple be done by noting the first and last horizontal and vertical squares of a piece once it is placed and seeing if this exceeds the existing bounds. If it does they must be updated.
This will not change the amount of moves made since a move in empty columns is invalid, but it will reduce the amount of moves considered. This would reduce the amount of columns where a piece can be placed to $2m' \cdot 2n'$, where $m' \leq m$ and $n' \leq n$.

4.2.5 Results

Pruning has significantly better results than general brute forcing. The success rate remains on par with brute forcing, as can be seen in Figure 25 in Appendix A. Pruning does require a very significantly lesser amount of work than Brute Force, it already requires around the same amount of square look ups on size 25 as pruning does on size 45, see Figure 29 in Appendix A. There are evidently many ways to create an unsolvable board state and thus opportunities to prune. Given that pruning is objectively better than regular brute forcing, there is no reason to choose it over the pruning method.
Figure 26 in Appendix A does show that the algorithm starts performing too long to be considered reasonable for us. From size 35 onward it starts timing out, quickly doing so a significant amount of times. When investigating such cases there seemed to be some reoccurring situations. On further

investigation they were board states that should have been pruned but were not. They are detailed and resolved in the Improved Pruning Algorithm in Section 4.3.

4.3 Improved Pruning Method

The faulty cases from Section 4.2.5 are discussed here using some of the definitions from Section 2.4. Methods for detecting these in a pruning function are also proposed and the performance of this new method is tried.

4.3.1 Type I Pruning Errors

Board states, such as in Figure 10, were reached where every single square in a gap can be reached individually, thus containing no holes to detect. However, when looking at the squares together not all of them can be reached from the same direction. This means that while all squares are technically reachable there is no actual move that can reach it.

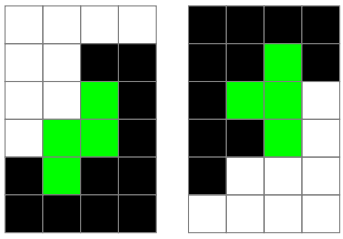


Figure 10: Examples of a Type I Error when using only tetrominoes.

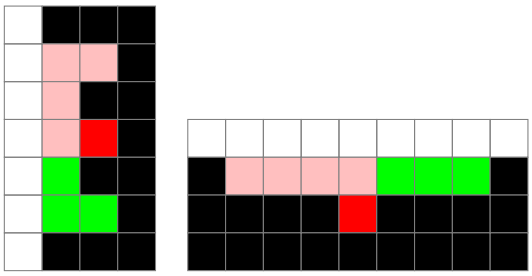


Figure 11: Examples of a Type II Error when using only tetrominoes. The moment a piece described by the pink squares is placed, the red square becomes unreachable.

***Definition* 13.** We speak of a *Type I error* in a gap $G \subseteq \neg C \cap T \iff \exists s \in G : (\nexists G' \subseteq G : s \in G' \wedge G' \in \mathit{Pieces} \wedge \bigcap_{i=1}^{|G'|} LOS(G'_i) \neq \emptyset)$

Informally this means that if we detect some square for which we cannot find a group of squares describing a polyomino available to us that includes that square we encounter a *Type I Error*, where all of that polyomino's squares can be reached from the same direction. This definition also provides us with a method for detecting them: finding such a grouping means a move can be made and there is no need to prune.
To detect this we propose Algorithm 2 that recursively builds a piece from the neighbours of the currently included pieces (line 14) if such a piece is: empty and has a line of sight on at least one of the directions of the current piece. The line of sight then becomes the intersection between the current piece and the new square. That square is added to the piece and we continue recursively (line 16). This continues until a piece of suitable size is reached, which gives us a polyomino that exists in *Pieces* and thus describes a piece that can be placed. We may return true (line 7) at this point to indicate there is no *Type I Error*.

4.3.2 Type II Pruning Errors

When solely relying on the size of a gap to determine if it can be filled it may fail when that gap has a shape that cannot be tiled with the given polyominoes. These are further investigated in Section 4.3.3. When attempting to fill this with a piece, there will always emerge some hole or the gap will be split into untileable ones.
The algorithm might attempt to do one of these moves and will immediately prune it. Since this is not the move that created the gap it will remain.
These formations are especially problematic on larger boards. After creating one the algorithm will continue along a branch that cannot contain a solution, but still exhausts moves on the branch. All moves would be attempted until the untileable gap is the only one remaining or the other gaps prove untileable and only then we slowly start backtracking.

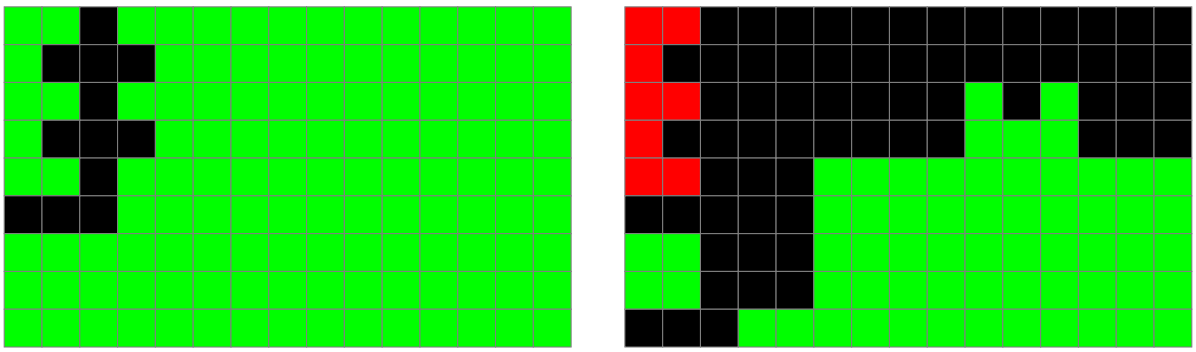


Figure 12: A *Type II error* is created and not pruned. The board state is then expanded upon even though every single move still results in an unsolvable state.

The method from Section 4.3.1 can easily be extended to include the detection of *Type II Errors*. *Type I* essentially asks if a piece could be placed in a given board state. *Type II* essentially asks if there exists a placement of a piece that does not result in an unsolvable, or prunable, state. Applying the pruning function on a board state where the piece from Algorithm 2 instead of returning true, would detect this.
While this does identify all *Type I and II Errors* it also gives some false positives. When gaps we consider are relatively big, a situation similar to Figure 13 may occur. Here there exists a way to tile the remaining gap with tetrominoes, as shown in the second image, but when a piece is tried by Algorithm 2, it detects a gap, shown in red in the third image. The algorithm does not consider the order in which pieces may be placed, so while it could have been filled before it still sees it as a hole.

Luckily this can be solved with existing methods. Because this takes place in a pruning function, any board state that is evaluated can be assumed to not have any unreachable holes in it. So any hole that is detected in Algorithm 2 must have been created by the piece that has been tried there. If we count all squares in the gap that this hole belongs to and it is of a size that is considered tileable by Algorithm 1 this means a move could have been made before and the hole is not considered untileable.
When applied to the board state in Figure 13 and this method tries a pink piece it would count the

Algorithm 2 Given a gap $G \subseteq T \cap \neg C$ and a square $s = (x, y) \in G$. Can s be covered through a move that does not result in a prunable board state. (Uses the LOS function from Section 2.4)

```
1: T ← Target as described in section 2.1
2: L ← all different sizes available in Pieces as described in Section 1.2

3: function ERR1(Piece = {(x, y)}, PieceLOS = LOS((x, y)))
4:     if directions = ∅ then return false
5:     end if
6:     if |Piece| ∈ L then
7:         PLACE(Piece)
8:         ReturnValue ← PRUNE()
9:         REMOVE(Piece)
10:        return ReturnValue
11:    end if

12:    for all (i, j) ∈ Piece do
13:        if (i + 1, j) ∉ Current Image ∧ (i + 1, j) ∈ Target Image ∧ (i + 1, j) ∉ Piece then
14:            newLOS ← LOS(i + 1, j)
15:            if ERR1(Piece ∪ (i + 1, j), PieceLOS ∩ newLOS) then
16:                return true
17:            end if
18:        end if

19:        if Idem for (i − 1, j), (i, j + 1), (i, j − 1) then
20:            ...
21:        end if
22:    end for

23:    return false
24: end function
```

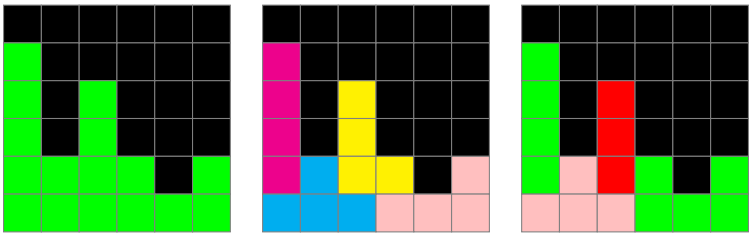


Figure 13: Incorrectly pruned state by Algorithm 2

red area consisting of eight squares (Figure 14). Since 8 mod $4 \equiv 0$ this area would be considered tileable and allow the placement of the pink piece.

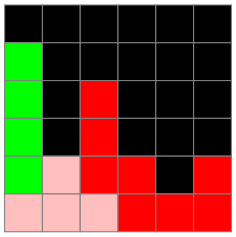


Figure 14: Counting the area of a hole during an adjusted Algorithm 2

4.3.3 Untileable Shapes

There are some figures that cannot be created by combining polyominoes of a single size p. Since shapes where $|T|$ mod $p \neq 0$ can never be created and are thus trivial to identify for this section, it will only detail shapes where $|T|$ mod $p = 0$ but that cannot be created regardless. The shapes detailed in this section are generally built from enough squares for only two polyominoes. We do briefly show how these may be extended to larger shapes with enough squares for more than two polyominoes.

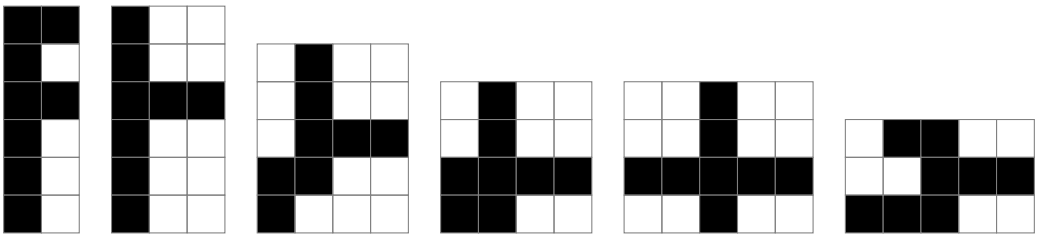


Figure 15: Shapes that cannot be created by tetrominoes

***Definition* 14.** Given a square $x = (i, j) \in Board$, $adjacent(x) = \{(i+1, j), (i-1, j), (i, j-1), (i, j+1)\} \cap Board$. I.e., all squares that are adjacent to x and within the bounds of the Board.

We define a shape that cannot be constructed from a single set of polyominoes $PS(x)$ in a general case. This definition is not exclusive to our rule set, but because it holds in a more general case it does still apply.
This shape will consist of three components, put together in Figure 4.3.3:

***Definition* 15.** A **key** piece. This is a single square and is denoted as striped in the example.

***Definition* 16.** Component **A** and **B** are shapes that both can be described as a polyomino. These are both connected to the **key** piece. They are only adjacent to the **key** piece; not to each other and not to any other component. **A** and **B** have sizes such that $|A| > 0 \wedge |B| > 0 \wedge (|A| + |B|) \bmod x = 0 \wedge |B| \bmod x \neq 0$.

***Definition* 17.** Two components **C** and **D** that can be described as a polyomino. They are also adjacent to **key** but may also be adjacent to each other. At this point it can be argued that they form a single polyomino instead of two where $|D| = 0$. It does not matter what perspective is held.
C and **D** are defined such that $|C| > 0 \wedge |D| \geq 0 \wedge |C| + |D| = x - 1$.

The orientation of the different components here is not fixed. They may be on any side of **key** relative to the other components given that the requirements in definition 15 through 17 are met.

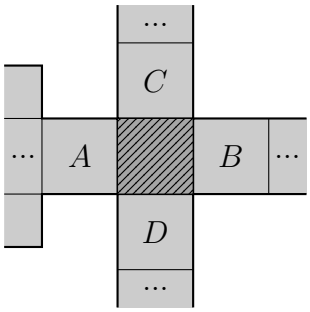


Figure 16: Illustration of a general shape that cannot be built with the polyominoes $PS(x)$.

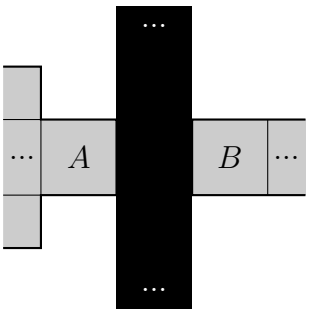


Figure 17: State that remains to be built after placing a piece on **C** and **D**. (Figure 16)

In order to construct the grey area in figure 16 every square must be covered by placing polyominoes of size x.
This requires all squares in $C \cup D$ to be filled. $|C| + |D| = x - 1$ and the only squares adjacent to C or D are the **key** or each other. So the only way to fill that area with polyominoes of size x is with a single polyomino of size x. Any polyomino that does not cover all of $C \cup D$ leaves an area to be filled that is smaller than x, which cannot be filled. A polyomino that does has one square left after covering $C \cup D$, which can only cover **key** since it is the only adjacent square. So any configuration that covers $C \cup D$ has a polyomino described by the area $C \cup D \cup key$.
This always results in the situation described in Figure 17, where the black area is a placed piece and the grey still has to be filled. Since A and B are both only adjacent to the **key** and it has just been removed this splits the remaining figure into two separate parts described by A and B. B is not tileable with polyominoes of size x by definition, so the initial grey areas could not be divided into polyominoes of size x.

For a shape S of this form: "S is filled $\iff$ $A \cup B \cup C \cup D \cup$ **key** is filled" with polyominoes from $PS(x)$. If we assume S is filled with polyominoes:
S is filled $\Rightarrow$ $C \cup D$ is filled by some polyomino $\Rightarrow$ ($C \cup D \cup$ **key** is filled) $\wedge$ (B is not filled) $\Rightarrow$ B cannot be filled $\Rightarrow$ S cannot be filled.
This provides a prove by contradiction that such a shape S cannot be filled.

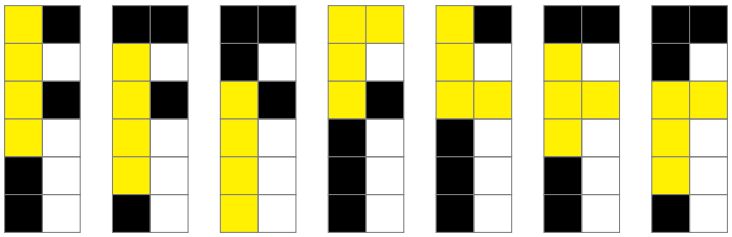


Figure 18: All possible piece placements for the first piece of Figure 15

There are larger shapes that are completely untileable. They follow the same logic: "A piece can be placed, but the remaining squares are then untileable". On larger shapes this remainder could be one of the smaller shapes described in this section. This is the case for the shape in Figure 19, where the only way to fill the red part is with one of the two yellow pieces shown. Where one creates an untileable section and the other creates a shape seen in Figure 15. Multiple pieces can be attempted to be placed, but they cannot fill the red part, so these could never complete the image.

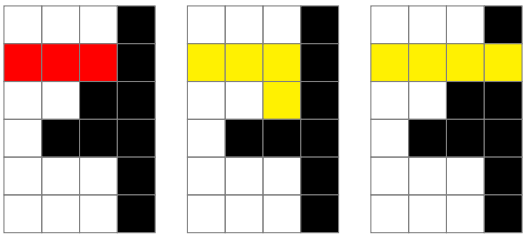


Figure 19: Larger shape reduced to an untileable state or a smaller state from Figure 15 that can also no be created

While the examples so far have shown small values for the size of A, larger ones are allowed as long as the other requirements are met. For an example of this see Figure 20.

4.3.4 Results

For the smaller target sizes the improved pruning method seems to have no significantly better success rate than the other algorithms, see Figure 25 in Appendix A. From a target size of 35 and onward the gap between the other algorithms starts growing and at a size of 50 we can see quite a significant improvement over the original pruning function.
The great improvement in regard to the pruning method can be attributed to the fact that the improved pruning method has fewer time outs, see Figure 26 in Appendix A. This would indicate that it accomplishes the goal it was designed for: discovering untileable shapes earlier on instead of spending a lot of time attempting useless moves. Not all of the removed time outs were untileable

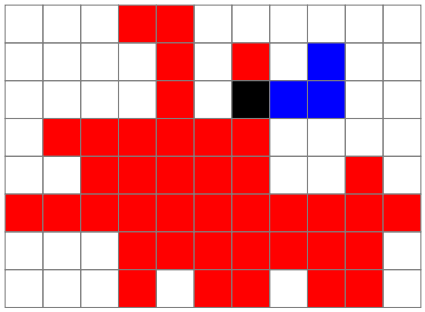


Figure 20: The general untileable construction with a very large $|A|$.

targets. It can for example be observed that the improved method had 32 fewer timeouts and 25 extra successes. Situations similar to Figure 12 on page 15 occur, where an untileable gap is created, but the whole target is still tileable.
The detection of the *Type II Errors* is intuitively a costly procedure, especially during the beginning of the algorithm where the gap that is to be evaluated is the entire rest of the target. This is shown in Figure 21, where we compare the relation between the amount of moves the algorithm considers and the corresponding amount of squares that need to be looked up for this, for the old pruning function and this new one. This clearly shows that while fewer moves and fewer look ups are generally required in the improved pruning function, many more look ups are required per move. An increase from around $1 : 25$ to $1 : 55$ can be observed.
This indicates that the ability to prune these situations is a great asset, the costly function even seems to be worth it. Reducing this cost would however still be a great priority when improving the pruning method.

It can be noted that the success rates of size 20 and 40 of the exhaustive methods drop significantly to what could be expected from the neighbouring target sizes. This is likely to be linked to the phenomenon of untileable shapes, the ones described in Section 4.3.3 can provide an explanation. Here shapes that cannot be created by using one type of polyomino are described. These definitions do not hold when multiple different sizes are available. Since we use tetrominoes and because 20 and 40 are divisible by 4, these targets would only be able to use tetrominoes and if such shapes are among the experiments they could not be tiled, lowering the success rate. But the other target sizes allow the usage of one other polyomino rendering some of them tileable, and therefore why these sizes have greater success rates.
This also explains the significant drop, especially compared to Monte Carlo, in Figure 28. The improved pruning function detects these early on, but Monte Carlo keeps attempting moves until none are available.

4.4 Monte-Carlo

This Monte Carlo implementation will iterate through every available move and evaluate them. The evaluation of a move will consist of a certain amount of randomly played games. These will consist of moves randomly chosen from the moves available at a certain board state. Because the moves that can be made are restricted to not be made on squares not in the *Target* there is some

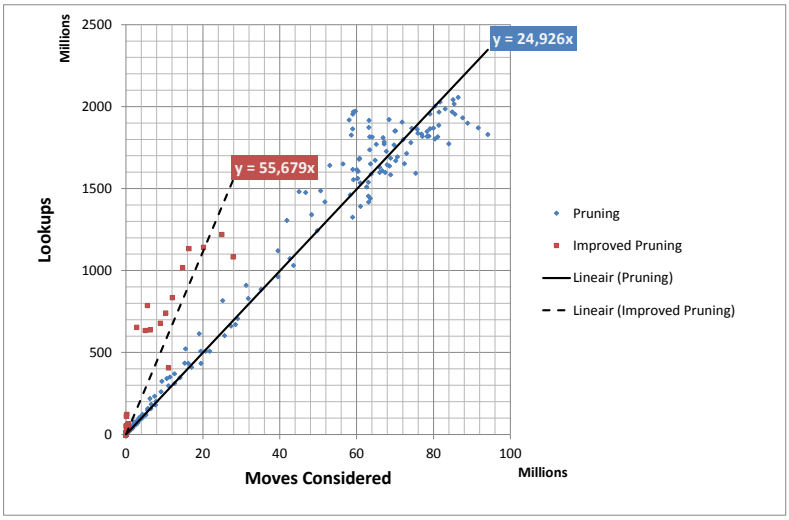


Figure 21: Relation between the amount of moves considered by the two pruning algorithms and the amount of coordinates of squares that are looked up.

pre-evaluation that ensures some potential in the moves, which causes another move to be chosen as the random move. There is a limit to this as to not get stuck if there are few to no moves to be found, the algorithm will assume a fail state in that case. The bounds used in the pruning algorithm from Section 4.2 can be used here to narrow the pool of potential random moves.
Since the holes and gaps are binary indicators of the quality of a board state, they would not prove useful for a metric for evaluation. Instead we will be playing random moves until a prunable state is encountered. The metric will be the amount of moves able to be made from the currently evaluated move.
The amount of these games that are played for each move are given as a parameter, *gamecount*. Because the algorithm has random components, experiments will be run multiple times for each provided target. Since this is time consuming the *gamecount* and runs have been limited to 30 and 10 respectively for these experiments.

4.4.1 Experiment Adjustment

This method contains a random element, unlike the others. To reflect the potential of runs reaching a fail state while an accepting state exists through pure chance, we make ten runs of the Monte Carlo method and the averages of all the metrics will be shown in Table 4 and the graphs. We also add another metric for Monte Carlo only, it is explained in Section 4.4.2 below.

4.4.2 Results

On small target sizes Monte-Carlo has a similar average success rate to the other algorithms, see Figure 25 on page 29. This becomes lower as the target sizes grows. An explanation for this is that as the size of the target increases the amount of possible moves also increases. This can make it

less likely for an accepting state to be found during random moves on bigger targets.
It does start to evaluate a significantly smaller amount of moves and square look-ups on larger board sizes than the other algorithms, see Figures 27 and 29. This means multiple runs could be made in the time the other algorithms normally terminate, even if not all of them are successful a solution may be found in similar time.
When looking at the method in a different way, the method becomes running the algorithm ten times and we accept when finding a solution. If we count the discovery of at least one accepting state among the ten tried as a success we obtain a different success rate. This new rate seems to be significantly better than before and better competes with the pruning methods. This seems to indicate the method has some potential, especially if multiple runs per instance are used.

4.5 Effects of Supply Order

The exhaustive methods employed have to iterate through the set of pieces that has been supplied in certain order. This means that adjustments to this order would have an effect on the order in which all of the different moves will be visited by these methods. Solutions may then be reached more quickly or more slowly depending on this order. This section will investigate the effects of using different orders of the tetrominoes on the amount of moves considered before a solution is found.
Intuitively the Square and I-pieces seem to be useful for easily constructing large rectangular surface areas. They leave few protruding squares when placed. This could potentially fill large convex parts more easily and leave smaller parts to be filled. Providing the proper shapes early on could improve the performance of our methods. In order to do this we will need a way to quantify or order the usefulness of a piece or the ease at which they fit together. We propose to count the number of faces a piece has to quantify the "*complexity*" of a polyomino. Here we define the faces as every line between two corners of the polyomino. This gives the Square and I-pieces a count of four as they are rectangular. The L-piece would have a count of six and the S-piece a count of eight in the case of the tetrominoes. See Figure 22 for an example of the count of the sides of these pieces and a larger example.

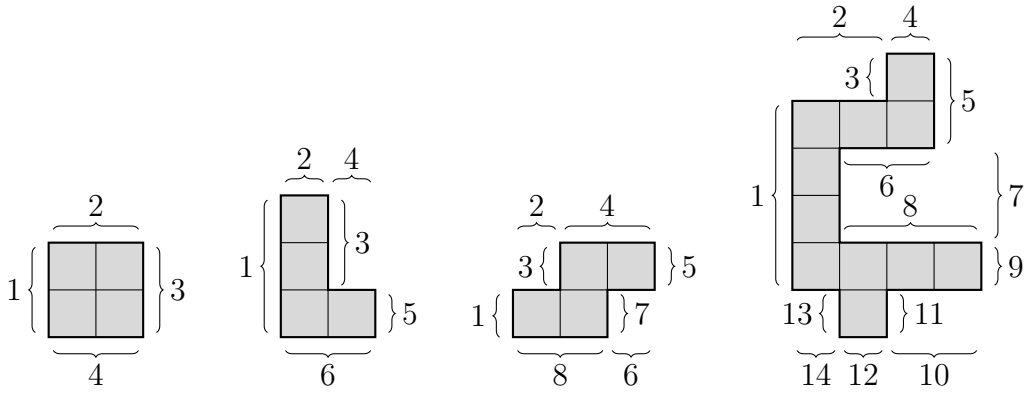


Figure 22: Examples of complexities for three tetrominoes and a undecomino (size eleven)

With this newly introduced metric we will be running tests using three different orders of tetrominoes:

- A **mixed** list of pieces. The tetrominoes contain three different complexities: four, six and eight.
- An **ordered** list, where the complexities are listed smallest to largest.
- A **reverse ordered list**, where the complexities are listed from largest to smallest.

4.5.1 Results

Figure 31 in Appendix A shows limited change in the success rate of the algorithms. Especially on the smaller target sizes there seems to be no difference, in the middle some marginal changes occur. The only significant deviation comes from the Ordered list on size 50. There is an improvement in success rate from $0,65$ to $0,73$. This is only the case in the normal pruning function, the improved method remains stable (and better).
This occurrence in the pruning method is paired with a drop in time outs for the Ordered list in Figure 32 on page 33. The fact that this again does not occur in the improved pruning methods seems to indicate that the Ordered list creates fewer untileable remainders in the targets, but only does so on larger target sizes. This relation cannot actually be shown with the current results, there seems to be only one instance of it, but if more tests could be run this might prove the idea that the lower complexity polyominoes are better for building larger shapes.
It can also be observed in the different entries in Table 2 on page 34 for Pruning, size 50 in "Moves Made", "Moves Considered", and "Elements Accessed", that this is also paired with lower counts for all of these. The reverse order does not show such a significant difference.

4.6 Inverse problem

In regards to the pruning methods the scope of all the moves that can be considered grows. Since the introduction of the search bounds based on currently placed pieces, the bounds grow as more pieces are placed and we come closer to a solution. Opposed to starting with an empty board and making moves until we have constructed a desired T, we could instead propose a method that starts with a complete T and we then take out pieces until it is empty. This section will explore the concept of an algorithm using this. Due to limited time this has not as of yet been implemented and tested.

4.6.1 Determining Column and Piece

Unlike the previous methods this method will evaluate all rows and columns combined with all possible pieces. The same idea of horizontal and vertical bounds introduced in the pruning algorithm (Section 4.2) can be used again and the current size of the target must be counted. If this count is of a size equal to one of an available piece we know we may choose such a piece as an initial move and then stop. If every filled square of the piece can be matched to a filled square on the board we may start to consider the move.

4.6.2 Determining Falling Direction

For the placement of a piece there is actually no more need to simulate a piece dropping down until it hits another. Instead we would need to consider from what directions a piece could have been dropped. We determine the intersection of the LOS of all squares in the piece. See the situation described in Figure 23. A piece also needs to fall on something. We would need to identify what directions have a square not belonging to the current piece bordering a square that does, we add the opposite of this direction to the Bordering Direction BD. This would mean it would have collided with this square should it have been placed and that it could have fallen from the opposite direction.

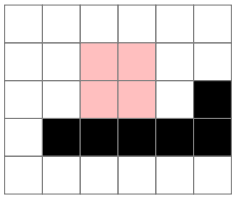


Figure 23: We consider the Pink piece for removal, the black is a still placed piece, LOS-Piece = {UP, LEFT}

In order to determine the possible Falling Directions FD for a certain piece, we thus identify it's LOS and Border Direction BD as described above. It is then determined as follows:

$$FD = LOS \cap BD \subseteq \{UP, DOWN, LEFT, RIGHT\} \tag{2}$$

FD can contain multiple directions but this is of no consequence, any of the directions it could have possibly dropped from are valid for a solution. This does actually cut down on some work the previous methods still did. If a piece can be placed in multiple different ways it would previously be considered multiple times, whereas only one would be needed here.

4.6.3 Inverse Pruning

Most of the same concepts for pruning can be maintained but they will now mostly apply to the remaining squares and the methods in which they are detected might have to change.

4.6.4 Inverse Holes: Detached Squares

Unlike in normal pruning we cannot create any holes, but we can cause the remaining image to split into multiple parts. It is easy to see this does not constitute a reachable board state as any shape created has a single origin, not multiple. We keep track of the current size of the target t. After a move we can then subtract the size of the piece *placed* from the target size. The algorithm then iterates through the board until a filled square is encountered, for every square considered we consider all adjacent filled squares (only once per square). We count the size s of this filled surface. If $t - placed \neq s$ then there must exist some filled square detached from the just considered surface. This means this state could not be created and should be pruned.

4.6.5 Inverse Gaps

In an inverse method the part of the board that still needs to be removed is always completely interconnected because of the rules enforced by the *Inverse Holes* part of the method. There is no separate function needed to count the size of the remaining area.

4.6.6 Type I Errors

Due to the new nature of the algorithm this type of error should no longer be encountered in the pruning function. The original problem arose when a section that should still be filled has no move that reaches it. No future move could remove it due to the given ruleset. Any move that cannot be made in an inverse implementation has some future move that clears the obstruction. A board state that contains such a piece can thus not be considered unsolvable and cannot be considered missed pruning case.

4.6.7 Type II Errors

These can still exist, there are still untileable shapes that could be encountered.

5 Conclusions

The Brute Force method is immediately outshined by any other method. The success rate remains high, as may be expected for exhausting all possibilities, but the sheer amount of time it takes to complete makes it an impractical solution.
Pruning this brute force approach has turned out to be very effective. The only real problem that arose was the occurrence of untileable shapes as remainders during the algorithm or as supplied target. This has technically been resolved and to great effect. But the implementation of this is still very rough, it is little better than playing out the gap that is being evaluated. The Improved Pruning algorithm can be considered the best performing in success rate and runtime, since Monte Carlo still requires multiple runs to approximate the same results. There is potential to improve the currently costly method of detecting *Type II Errors*, Section 6 proposes some directions to address this. Improving the greatest cost of the best algorithm could prove very fruitful.
The definition provided for untileable shapes seems to describe most of the unfillable gaps we encounter and is even able to predetermine some targets to be untileable. The definition is not proven to be complete however. There is no indication that every untileable shape adheres to it. Attempting to define all such shapes does seem to be an interesting problem.
Because Monte Carlo has not had extensive testing done as of yet, it cannot be said if it can be a better method. When viewing the method through the lens of finding at least one accepting run it starts to approach the performance of the improved pruning method. This combined with the speed it has would make it seem it has potential as a valid method.

6 Future Work

This thesis leaves some work to be done and during the writing some potential directions for future work surfaced, but time was lacking to implement them:

- Since polyominoes drastically blow up in size allowing multiple sizes of polyominoes can be very time consuming, especially with higher sizes. Tests using this have not been handled in this thesis, but all concepts and methods discussed should be extendable to multiple sets.

- This thesis has used a relatively naive random target generation method, see Section 4. Some squares have been in the pool longer and have thus more opportunity to be chosen. This lends itself to shapes that have a decently filled core, with some protruding parts. The shapes are not truly random. Finding a way to obtain a better distribution could lend itself to better testing, but this may very well be a problem on its own.
  Additionally the effect of the different methods on certain types of shapes might be worth investigating; think convex shapes, shapes filled with holes, or shapes that mainly consist of lines of a square thick. The Supply Order method for example could see greater variance if in different types of shapes. Methods to generate specific types of targets could allow a delve into this.

- The tests conducted so far have been relatively limited. Effects of larger targets may be very time consuming but are worth investigating. Especially in the case of the Ordered supply order, where a potential improvement has appeared on a target size of 50.
  This thesis has also not delved deeply into different values for the parameters for Monte Carlo. Time constraints prevented extensive testing of high gamecounts. A thorough investigation into the optimal parameters and perhaps their relation to certain shapes, or sizes could prove interesting.

- By their nature the exhaustive methods produce some uniformity in their results. Since pieces and falling directions have a predetermined order many of the same piece or same direction are often found in the solution. Varying this through some method might have effects on the solutions found.

- The current method of finding untileable sections in the target is the most costly part of the algorithm and does not yet abuse the description we provided. We propose a possible method to leverage this information.
  In the case were the provided pieces consist of a single polyomino set, the definition for untileable shapes from Section 4.3.3 could be used to pre-calculate shapes that cannot be created. Recall from the definition for untileable shapes that the sections C and D have limited possibilities in regards to the size and thus shape they may take. If these could be properly matched to the provided *Target Image* untileable targets or board states could be discovered, given x possibilities for combinations of C and D, in a time bounded by mnx.
  To ensure optimal performance of such a method further investigation of the untileable shapes may be required.

- The current definition of untileable shapes using a single polyomino set does not seem to be complete as of yet. It has described all lot of the encountered states relatively well, but Figure 24 already shows an unbuildable shape that is not described by our construction from Section 4.3.3 (its untileability will be left as an exercise to the reader).
  This would indicate our construction has room to be more generalized or additional definitions may be required.

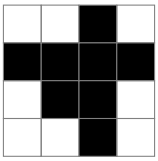


Figure 24: Untileable shape not described by our definitions

- Note that the method for properly detecting *Type II Errors* nearly solves a gap it evaluates. This is a very costly procedure to detect these errors. We could lean into this concept and adjust the detecting function so it properly plays out the gap, having the pieces that are normally tried and deleted remain. This would require the algorithm to deal with the order of placement in a way it currently avoids.
  Instead of continuously attempting to solve the whole target, this is could solve the *gaps* of a board state separately.
  Such an algorithm would start with the empty board, essentially one big gap, and attempt a move. Where normally the pruning function would detect the error in a gap, it now plays it out. Were the gap would normally be considered tileable it will now have been tiled already. Then it continue to do the same for all new gaps that have been created by this move.
  Something to consider when attempting such a method would be the order in which gaps are evaluated, completely filling a large part of the target and then discovering the second one is untileable means discarding a large part of the work that had just been done.

References

[BvdZ18] Hans L. Bodlaender and Tom C. van der Zanden. On the Exact Complexity of Polyomino Packing. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, 9th International Conference on Fun with Algorithms (FUN 2018), volume 100 of Leibniz International Proceedings in Informatics (LIPIcs), pages 9:1–9:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[HK04] H.J. Hoogenboom and W.A. Kosters. How to construct tetris configurations. International Journal of Intelligent Games Simulation, 3, No 2:94–102, 2004.

[Jen03] I. Jensen. Counting polyominoes: A parallel implementation for cluster computing. Lecture Notes in Computer Science, 2659:203–212, 2003.

[Jen09] I. Jensen. Series for lattice animals or polyominoes: Number of animals. https://researchers.ms.unimelb.edu.au/~ij@unimelb/, April 2009. Accessed on 2019-7-24.

[Kan92] R. Kannan. Lattice translates of a polytope and the frobenius problem. Combinatorica, 12:161–177, 1992.

[Kla67] D.A. Klarner. Cell growth problems. Canadian Journal of Mathematics, 19:851–863, 1967.

[Liu03] A. Liu. Packing rectangles with polynominoes. Mathematical Medley, 30, No 1:3–10, 2003.

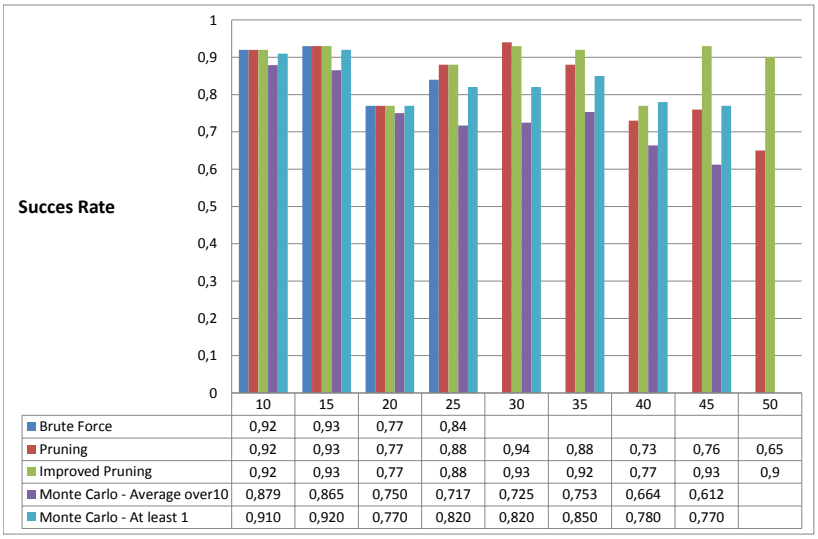
[Mar97] W.R. Marshall. Packing rectangles with congruent polynominoes. Journal of Combinatorial Theory, Series A, 77:181–192, 1997.

[Syl82] J.J Sylvester. On subinvariants, i.e. semi-invariants to binary quantics of an unlimited order. American Journal of Mathematics, 5:134, 1882.

[Tri17] A. Tripathi. Formulae for the frobenius number in three variables. Journal of Number Theory, 170:368–389, 2017.

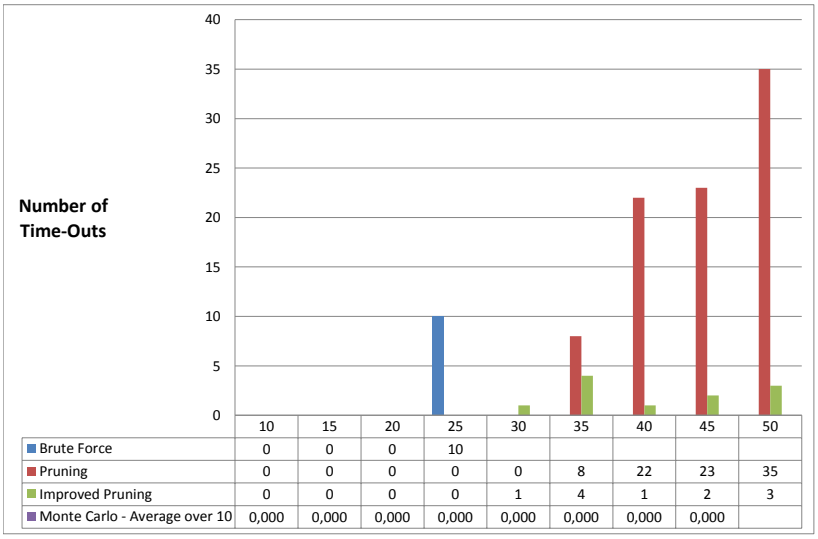
A Appendix

This appendix contains the graphs and tables from the experiments in Section 4.



	10	15	20	25	30	35	40	45	50
Brute Force	0,92	0,93	0,77	0,84					
Pruning	0,92	0,93	0,77	0,88	0,94	0,88	0,73	0,76	0,65
Improved Pruning	0,92	0,93	0,77	0,88	0,93	0,92	0,77	0,93	0,9
Monte Carlo - Average over10	0,879	0,865	0,750	0,717	0,725	0,753	0,664	0,612	
Monte Carlo - At least 1	0,910	0,920	0,770	0,820	0,820	0,850	0,780	0,770	

Figure 25: The success rate out of 100 targets for every algorithm, for every target size tested. Note that due to time constraints Brute Force has only run until size 25 and Monte Carlo until 45.



	10	15	20	25	30	35	40	45	50
Brute Force	0	0	0	10					
Pruning	0	0	0	0	0	8	22	23	35
Improved Pruning	0	0	0	0	1	4	1	2	3
Monte Carlo - Average over 10	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	

Figure 26: The amount of times every algorithms has timed out (420 seconds) during testing. These are considered a non-accepting instance. Note that due to time constraints Brute Force has only run until size 25 and Monte Carlo until 45.

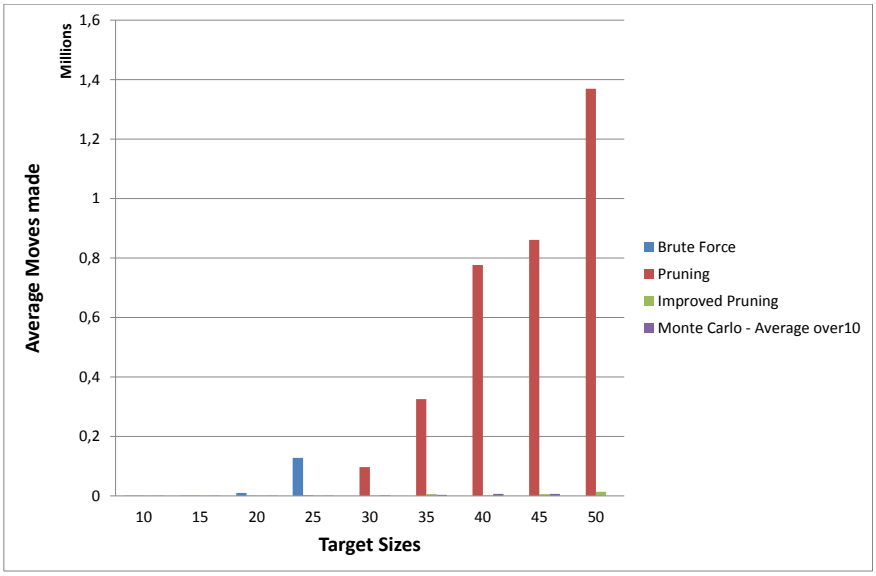


Figure 27: The amount of moves every algorithm has made for every target size tested. Note that due to time constraints Brute Force has only run until size 25 and Monte Carlo until 45.

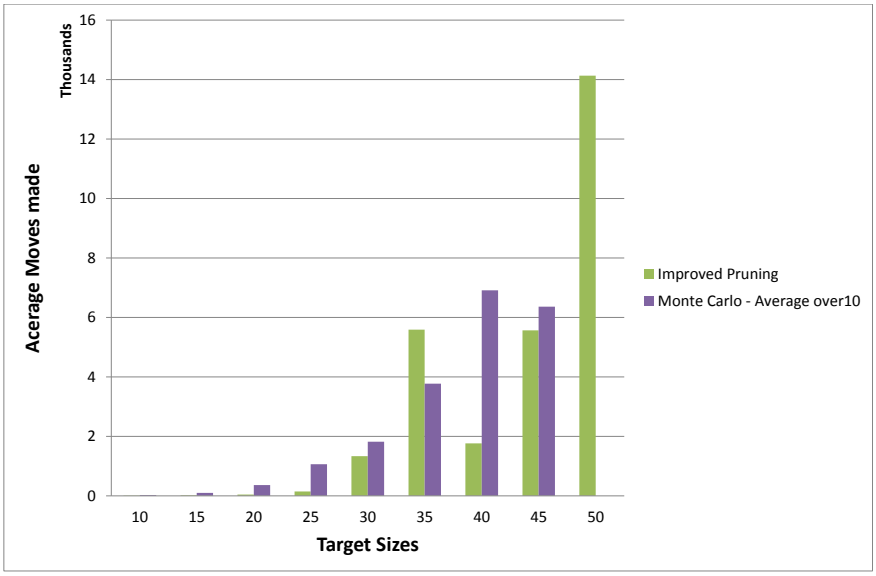


Figure 28: The same results as Figure 27, but with the Brute Force and Pruning method (until size 45) removed for a clearer view of the other results.

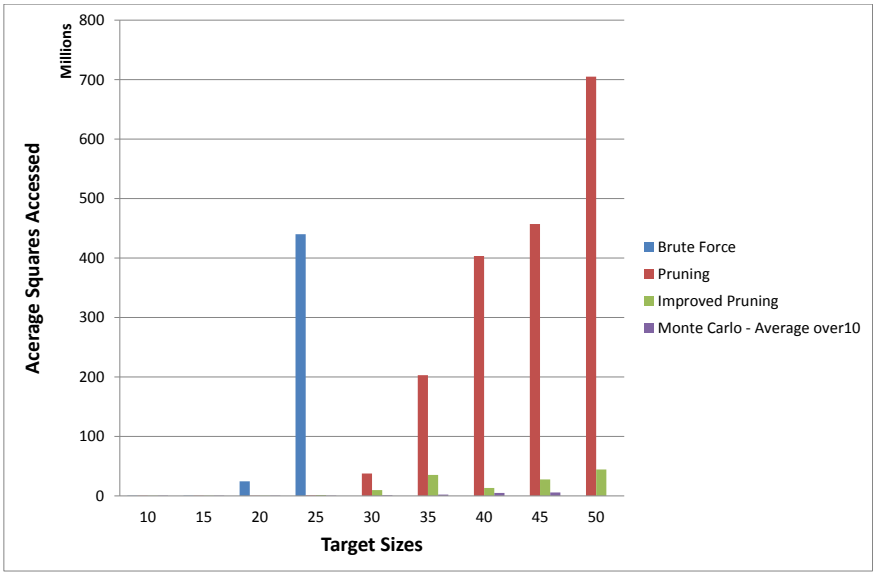


Figure 29: The amount of squares looked up for every algorithm, for every target size tested. Note that due to time constraints Brute Force has only run until size 25 and Monte Carlo until 45.

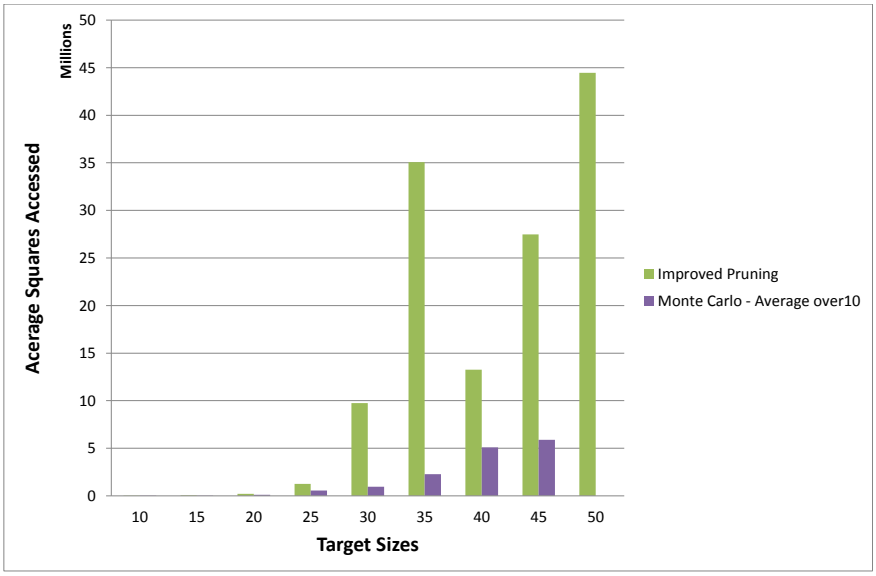
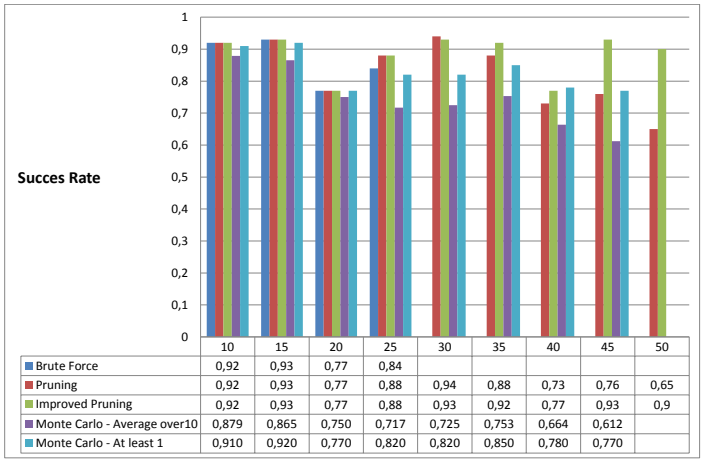
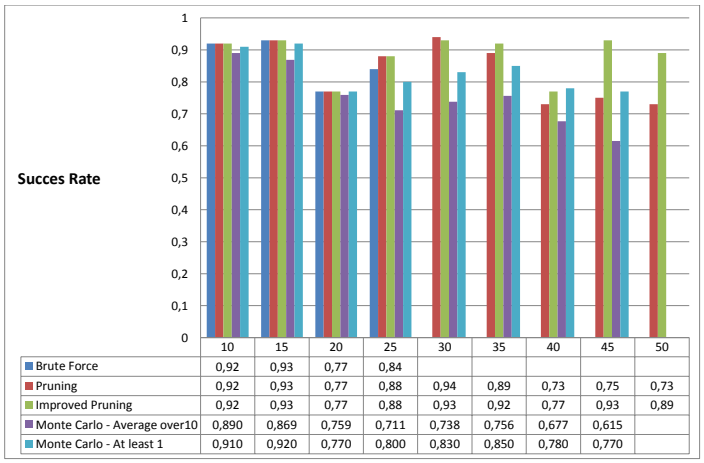


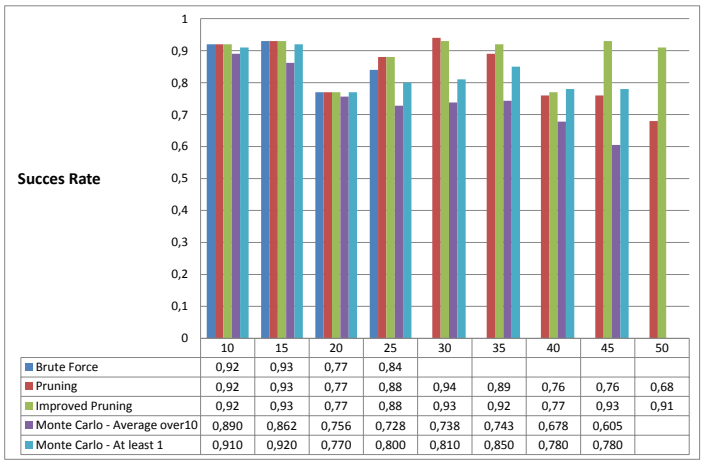
Figure 30: The same results as Figure 27, but with the Brute Force and Pruning method (until size 45) removed for a clearer view of the other results.



	10	15	20	25	30	35	40	45	50
Brute Force	0,92	0,93	0,77	0,84					
Pruning	0,92	0,93	0,77	0,88	0,94	0,88	0,73	0,76	0,65
Improved Pruning	0,92	0,93	0,77	0,88	0,93	0,92	0,77	0,93	0,9
Monte Carlo - Average over10	0,879	0,865	0,750	0,717	0,725	0,753	0,664	0,612	
Monte Carlo - At least 1	0,910	0,920	0,770	0,820	0,820	0,850	0,780	0,770	

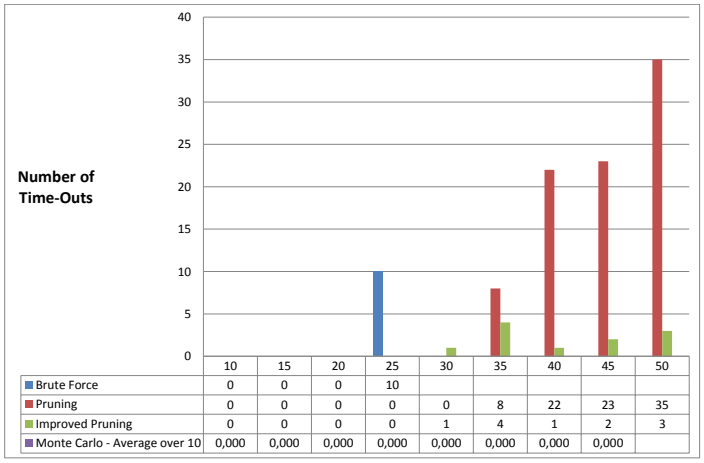


	10	15	20	25	30	35	40	45	50
Brute Force	0,92	0,93	0,77	0,84					
Pruning	0,92	0,93	0,77	0,88	0,94	0,89	0,73	0,75	0,73
Improved Pruning	0,92	0,93	0,77	0,88	0,93	0,92	0,77	0,93	0,89
Monte Carlo - Average over10	0,890	0,869	0,759	0,711	0,738	0,756	0,677	0,615	
Monte Carlo - At least 1	0,910	0,920	0,770	0,800	0,830	0,850	0,780	0,770	

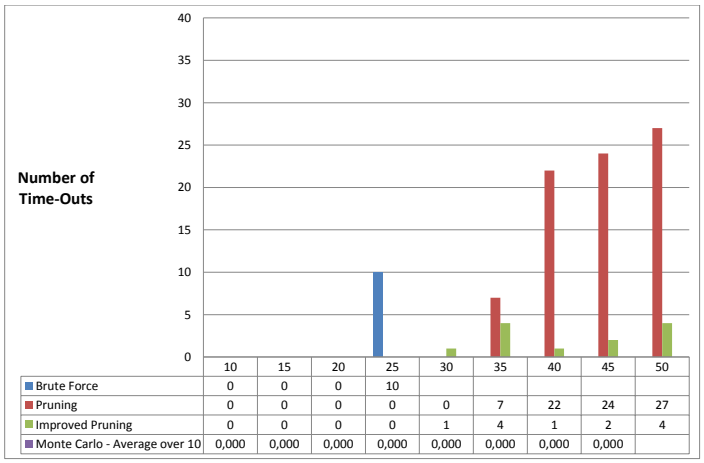


	10	15	20	25	30	35	40	45	50
Brute Force	0,92	0,93	0,77	0,84					
Pruning	0,92	0,93	0,77	0,88	0,94	0,89	0,76	0,76	0,68
Improved Pruning	0,92	0,93	0,77	0,88	0,93	0,92	0,77	0,93	0,91
Monte Carlo - Average over10	0,890	0,862	0,756	0,728	0,738	0,743	0,678	0,605	
Monte Carlo - At least 1	0,910	0,920	0,770	0,800	0,810	0,850	0,780	0,780	

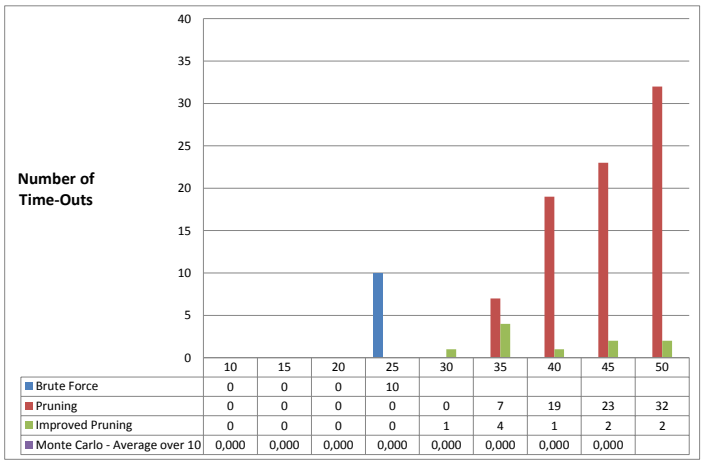
Figure 31: The success rate results of the supply orders Mixed, Ordered, and Reverse Ordered respectively.



	10	15	20	25	30	35	40	45	50
Brute Force	0	0	0	10					
Pruning	0	0	0	0	0	8	22	23	35
Improved Pruning	0	0	0	0	1	4	1	2	3
Monte Carlo - Average over 10	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	



	10	15	20	25	30	35	40	45	50
Brute Force	0	0	0	10					
Pruning	0	0	0	0	0	7	22	24	27
Improved Pruning	0	0	0	0	1	4	1	2	4
Monte Carlo - Average over 10	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	



	10	15	20	25	30	35	40	45	50
Brute Force	0	0	0	10					
Pruning	0	0	0	0	0	7	19	23	32
Improved Pruning	0	0	0	0	1	4	1	2	2
Monte Carlo - Average over 10	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	

Figure 32: The time outs of the supply orders Mixed, Ordered, and Reverse Ordered respectively.

Algorithm	Supply Order	Target Size	Moves Considered	Moves Made	Elements Looked up	Success Rate	Time Outs
Brute Force	**Mixed**	10	557,68	18,53	20.866,36	0,92	0
		15	14.642,42	280,58	517.669,36	0,93	0
		20	840.517,05	9.527,7	24.258.807,56	0,77	0
		25	18.660.625,18	127.551,37	440.200.063,9	0,84	10
	Ordered	10	582,04	19,31	21.928,27	0,92	0
		15	14.499,76	274,42	508.210,16	0,93	0
		20	768.469,48	8.521,74	21.836.685,8	0,77	0
		25	18.201.149,97	123.667,13	427.910.576,3	0,84	10
	Reverse Ordered	10	561,34	18,63	21.021,22	0,92	0
		15	14.513,07	276,78	511.057,12	0,93	0
		20	758.153	8.524,47	21.731.214,47	0,77	0
		25	17.969.517,3	125.334,23	428.018.222,6	0,84	10
Pruning	**Mixed**	10	68,00	9,59	2.323,56	0,92	0
		15	251,26	33,57	7.861,68	0,93	0
		20	2.371,68	257,59	70.257,80	0,77	0
		25	36.054,96	1.982,97	954.001,36	0,88	0
		30	1.330.932,35	96.789,93	37.423.817,24	0,94	0
		35	7.643.103,23	325.320,13	203.123.065,51	0,88	8
		40	16.018.737,91	776.401,68	403.312.114,50	0,73	22
		45	18.034.758,54	861.263,22	457.222.342,59	0,76	23
		50	28.242.850,33	1.369.152,44	705.211.163,17	0,65	35
	Ordered	10	77,09	10,37	2.481,46	0,92	0
		15	246,91	32,25	7.575,77	0,93	0
		20	2.370,10	250,83	69.311,56	0,77	0
		25	36.276,01	2.006,27	959.286,45	0,88	0
		30	1.333.875,89	96.923,33	37.495.153,79	0,94	0
		35	7.033.052,85	297.211,95	186.076.122,64	0,89	7
		40	17.084.212,50	839.413,87	423.741.491,11	0,73	22
		45	19.510.852,74	965.780,71	499.024.906,90	0,75	24
		50	22.596.312,55	1.055.516,04	551.219.913,09	0,73	27
	Reverse Ordered	10	68,69	9,69	2.396,03	0,92	0
		15	233,11	32,07	7.494,91	0,93	0
		20	2.557,66	254,13	75.388,92	0,77	0
		25	36.172,60	1.972,26	956.841,82	0,88	0
		30	1.338.107,69	97.078,36	37.587.224,53	0,94	0
		35	7.157.653,77	305.355,40	188.906.555,07	0,89	7
		40	13.787.989,81	610.384,25	344.338.152,73	0,76	19
		45	20.179.053,73	956.866,86	519.673.298,21	0,76	23
		50	25.628.331,19	1.314.480,48	649.074.745,26	0,68	32

Table 2: Testing results part 1: Moves Considered, Made, and Elements Looked Up are the average out of 100 test targets. Time Outs is a count out of 100 tests.

Algorithm	Supply Order	Target Size	Moves Considered	Moves Made	Elements Looked Up	Success Rate	Time Outs
Improved Pruning	**Mixed**	10	53,08	7,57	14.029,32	0,92	0
		15	137,88	16,48	59.613,01	0,93	0
		20	476,27	45,14	195.096,92	0,77	0
		25	6.598,20	145,39	1.259.009,42	0,88	0
		30	63.420,89	1.337,19	9.753.474,88	0,93	1
		35	426.286,26	5.587,93	35.063.673,01	0,92	4
		40	178.113,81	1.761,32	13.255.046,09	0,77	1
		45	353.358,76	5.568,39	27.479.579,90	0,93	2
		50	672.018,45	14.130,23	44.468.552,78	0,90	3
	Ordered	10	62,17	8,35	14.189,33	0,92	0
		15	136,74	15,58	58.943,13	0,93	0
		20	537,13	44,89	187.680,83	0,77	0
		25	6.603,54	144,09	1.243.486,82	0,88	0
		30	61.707,51	1.293,44	9.381.623,24	0,93	1
		35	426.084,12	5.449,77	34.831.109,52	0,92	4
		40	51.014,02	651,51	8.936.554,69	0,77	1
		45	304.159,82	5.352,52	27.306.622,35	0,93	2
		50	701.045,72	14.708,65	45.866.113,62	0,89	4
	Reverse Ordered	10	53,77	7,67	14.163,23	0,92	0
		15	123,35	15,44	59.427,84	0,93	0
		20	699,09	53,71	228.387,90	0,77	0
		25	6.612,38	143,57	1.265.681,91	0,88	0
		30	63.307,70	1.337,42	9.797.934,59	0,93	1
		35	405.348,33	5.347,16	34.483.007,90	0,92	4
		40	67.547,41	807,44	10.120.772,50	0,77	1
		45	388.978,70	5.720,95	28.677.203,89	0,93	2
		50	378.563,27	6.770,05	28.800.646,97	0,91	2

Table 3: Testing results part 2: Moves Considered, Made, and Elements Looked Up are the average out of 100 test targets. Time Outs is a count out of 100 tests.

Algorithm	Supply Order	Target Size	Moves Considered	Moves Made	Elements Looked Up	Success Rate	At Least 1	Time Outs
Monte Carlo	**Mixed**	10	288,92	24,62	12.806,15	0,88	0,91	0
		15	869,45	95,69	30.653,70	0,87	0,92	0
		20	2.761,11	357,99	108.623,62	0,75	0,77	0
		25	7.307,74	1.064,16	546.511,46	0,72	0,82	0
		30	12.083,91	1.819,37	962.345,74	0,73	0,82	0
		35	20.818,52	3.772,22	2.270.938,57	0,75	0,85	0
		40	36.262,95	6.908,93	5.084.681,33	0,66	0,78	0
		45	39.584,80	6.365,91	5.887.824,74	0,61	0,77	0
	Ordered	10	282,15	24,17	12.511,40	0,89	0,91	0
		15	894,84	96,28	31.345,27	0,87	0,92	0
		20	2.815,27	360,97	110.136,46	0,76	0,77	0
		25	7.349,19	1.057,39	545.516,83	0,71	0,80	0
		30	12.023,71	1.804,55	968.299,29	0,74	0,83	0
		35	20.426,02	3.724,07	2.248.352,59	0,76	0,85	0
		40	36.577,17	6.904,85	5.092.237,65	0,68	0,78	0
		45	40.187,58	6.390,62	5.924.523,18	0,62	0,77	0
	Reverse Ordered	10	282,68	24,15	12.642,03	0,89	0,91	0
		15	872,67	95,02	30.706,35	0,86	0,92	0
		20	2.849,37	358,35	111.454,09	0,76	0,77	0
		25	7.485,34	1.085,99	557.762,45	0,73	0,80	0
		30	12.046,99	1.822,41	963.699,96	0,74	0,81	0
		35	20.663,72	3.718,50	2.251.698,86	0,74	0,85	0
		40	36.517,90	6.944,21	5.097.992,51	0,68	0,78	0
		45	39.798,79	6.422,94	5.934.727,70	0,61	0,78	0

Table 4: Testing results part 3: Moves Considered, Made, and Elements Looked Up are the average out of 100 test targets. Time Outs is a count out of 100 tests.