Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Rectangle packing in practice

Petar Borisov Stoykov

Supervisors:
prof.dr. Hans Zantema
dr.ir. Bart Mesman
ir. Albert van den Bosch

Eindhoven, August 2017

Abstract

Various fields of practice face the problem of arranging rectangular items in larger rectangular frames, thus different ways of automating that process have been found through the years. For instance, while printing images on large scale materials, that arrangement determines how much of the material is wasted while cutting out the separate prints.

During this project, an approach using an SMT solver is developed with three different variations. Along with the SMT methods, four others are implemented, namely: using a simple direct algorithm, backtracking, genetic algorithm and particle swarm optimization.

Improvements on all methods are tested out and their best versions are compared on a set of benchmarks. As the thesis is closely related to a printing company, it uses examples from practice along with random test cases to determine the results and draw conclusions from them.

The end product of this project is an application that uses different sets of the discussed methods, based on the input, to deliver the best frame coverage they find.

Preface

I would like to thank Hans Zantema for everything. He was the reason I started working with BigImpact and when that work became the basis of this thesis he helped in every step of the way, from the start till the very end. His experience in the field was invaluable to this project in the form of frequent discussions and advices.

The other major influence through my work was Albert van den Bosch. Not just for hiring me but for being an amazing employer, having the right attitude and providing a great workspace. His understanding of the problem faced in practice helped the advances of most methods.

On the other side of the continent, my parents provided enough moral support to keep me going. They were also the main reason for me to be in this university in the first place.

Special thanks goes to all my friends that had to listen to me talk about rectangles. The ones that suffered the most : Paulina, Rodrigo and Petya.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This project addresses a problem posed by BigImpact, a large-scale printing company in Eindhoven. They own several machines that can print on materials, as wide as 5 meters. Even though the printing machines use materials of fixed sizes, the company offers services for images of all dimensions. This customer's convenience leads to the question: how does one fit the different images on the material so they can be printed together efficiently?

After a worker has figured out how to arrange the images and combine them in one big pdf that is sent to the large-scale printer, the resulting print needs to be cut so the individual images can be extracted. Throwing some of the material as waste in the process of cutting the images is virtually unavoidable.

There are two aspects of the production that can be improved, namely reducing human labour while ordering the images and minimizing the wasted material. A benefit that comes with automating the process is eliminating human error, such as forgetting to print an image.

For a simple job, containing a few images, a human can spot the best order in a manner of seconds. If the job contains 20+ images with various sizes it could take a minute for the worker to find an order that provides a reasonable use of the material, which is still probably not optimal.

On the other hand, there are the machines. Every time a printer finishes a job it needs to start another one as every idle minute in which the machine could have been working is undesired. Because of this, a worker has limited time in which to prepare the next printing job.

The problem of fitting rectangles in a larger rectangle is known as *two-dimensional bin packing* (according to Lodi et. al [1]), or more precisely *rectangle packing*, as no irregular shapes are addressed in the project.

This is not a new problem and much work has been done in developing solutions for it, both practical and theoretical. It has been studied extensively in the past because of its popularity in a large number of real-world applications. Such applications include cutting sheets of any material (wood, metal, glass, etc.) into smaller rectangles, placing ads in a newspaper, packing trucks or cargo ships, and even at arranging images on certain websites.

Each area of practice comes with its own specifications, but the general idea is the same: Given a set of N rectangular items $j \in J = 1, ..., N$, each of width W_j and height H_j, and an unlimited number of rectangular bins of height H and width W [1]. Assuming that the items have a fixed orientation, what is the minimal number of bins needed to allocate, without overlapping, all of the items with their edges, parallel to those of the bins?

To fit the needs of BigImpact some aspects of that definition need to be changed:

- The items' orientation can change, as items should be able to rotate by $90°$.
- The height of the bin may be a variable, as some materials come in rolls instead of specified sheets.

In the above description of the problem, the rectangles (images) are to be placed in bins. A bin with the rectangles fitted in it will be called a ***nest*** further in the document. The bin itself will also be referred to as a ***frame***. Therefore ***nesting*** describes the process of fitting items inside a frame.

There are a number of approaches solving the general specification of the problem, found in literature. Some of the most popular ones are Alternate Directions (AD) [1], Bottom-Left Fill (BLF) [4], Improved Lowest Gap Fill (LGFi) [10], Touching Perimeter (TP) [11], Hybrid Next-Fit (HNF) [8], Hybrid First-Fit (HFF) [5], Finite Best-Strip (FBS) [2], and Floor Ceiling (FC) [1].

They have proven themselves helpful over the years, but because of recent advances in computer science, there might be new competitive approaches available. In particular, the use of satisfiability modulo theories (SMT) is researched in this project. The SMT solver that will be used is Microsoft's Z3 [14]. It is an open-source tool that is constantly improving.

The literature also describes methods that use genetic algorithms with comparable results. A similar approach will be implemented in this project aiming to improve the existing ones.

1.1 Problem categorizations

Often, not all given items can fit in a single frame so in order to not leave out any items, more than one frames are used for a solution. So in general the problem question is:

`How to order the given items in one or more frames with the least combined frame areas as possible, where the frames have defined requirements?`

Based on the company's specifications, the problem itself can be split into two variations depending on those requirements:

- Sheet problem: The width and height of the frame are given as input. (material is in the form of a sheet)
- Roll problem: Only the width of the frame is given while the height is variable that must not exceed a given maximum value. (if a roll of the material is used)

The main focus of the research is on the first variation. If a method can perform well solving the classic (sheet) problem then it usually requires small modifications to be used with the second variation. Later, in the descriptions of each method, there will be a section describing if and what needs to be changed to make it work for a material with a variable length.

1.2 Project's aim

The project, described in this paper, compares different methods for solving the rectangle packing problem that BigImpact is facing. The criteria to evaluate the methods is their ability to consistently deliver results with good, close to optimal, coverage in limited time.

For our purposes the time limit is set to no more than 40 seconds spent on creating one nest. As stated before, the printers should not wait for the creation of the next printable file. It should be readily available on demand. Even though the average time to print a file is lot longer than 40 seconds, one needs to take into account that the worker's job is not automated and nesting is just one stage of the process.

It is preferred to deliver the printouts to the customers on time even if that means some wasted material. The additional cost for that waste is less than the cost of delaying production.

The following is a list of the methods that have been implemented and compared during this project.

- Floor Ceiling algorithm (FC)
- Basic SMT approach

- Best coverage SMT approach
- SMT Hybrid approach
- Backtracking algorithm
- Particle swarm optimization (PSO)
- Genetic Algorithm

1.3 Benchmarking

The nature of the problem is that the results depend greatly on the number of items and their dimensions. There probably isn't a single method that gives the best results for every input. As such, a test set of 120 inputs has been selected from practice and to avoid creating a method that works perfectly for specific inputs, a different set of 300 randomly generated cases are also used. Each method runs on all test cases, produces results and the average of the resulting coverages is used to determine which one works best. After that the results for individual cases can be examined to find relations between inputs and method performance.

Coverage is used as criteria instead of "number of bins" mainly because of the second variation of the problem. When printing on a roll, the number of nests produced is not of great importance, their combined length is. Or in other words, the overall coverage that has been achieved determines the winner. When evaluating problems working on a sheet both the number of nests and the coverage are equally relevant, thus the coverage is preferred as it serves both cases.

The definition of ***coverage*** is the ratio of the sum of all item areas to the sum of all nest areas. Where a coverage equal to 1 means that the whole area of the nests is covered by items and there is no waste. If the only half of the area is used and the other half is considered waste then the coverage equals 0.5.

The implementation of all the methods is done in Python 3.5 and the tests are conducted on a machine with the following specifications:

```
Lenovo ThinkPad W541
CPU: Intel Core i7 (4th Gen) 4710MQ / 2.50 GHz
RAM: 8 GB DDR3 1600 MHz
```

1.4 Paper's structure

This thesis is divided into 11 chapters. Chapters two through eight present each implemented method with a description on why it is chosen and how exactly it is implemented. Intermediate results gathered from those methods are shown and discussed in chapter 9. While chapter 10 describes how these methods can be used in cooperation to produce even better results and what these results are. And the paper ends with the conclusions made during the project (chapter 11).

1.5 About the methods

The final goal of each method is to fit all given items in as many as necessary number of nests and achieving the highest coverage possible within a given period of time.

As stated before, the problem has two variations: nesting on a roll or on a sheet. The difference of how these methods solve both variations is usually a minor change in the algorithm or the final formatting of the nest. Each method contains a section explaining those differences, but in order to make the explanations more clear the main focus is on just one of the variations. It is stated at the start of each method's chapter which one is being used.

While in practice the rotation of items can be disabled in special cases, the methods are explained with the idea that all items can be rotated. If the description does not define how disabling rotation would affect the algorithm then the change is trivial.

Chapter 2

First Fit and Floor Ceiling

The two algorithms are described in the same section as Floor Ceiling is an improvement on First Fit, thus only FC is used in the later comparison of methods.

It is possible to construct all necessary nests for all items with these methods. The choice of problem variation to describe these algorithms is irrelevant as both of them create levels which are then used to form the necessary nests independently of the algorithms.

2.1 First Fit

FF is one of the oldest and simplest that solves the rectangle packing problem. It is also referred to as Best-Fit Decreasing Height (BFDH) [13]. It's aim was to provide a quick solution to a complicated problem that had no automated solvers at the time. Provided an optimal solution is not necessary, it is useful as the algorithm runs in a few milliseconds even for complicated problems (ones with over 200 files).

This is an overview of how the algorithm works: First the items are sorted by height. The highest item is placed in the bottom left corner of the frame. The second highest item is tested if it fits to the right of the first one. If it doesn't fit, the item is placed above the first, creating a new **level**. Then the algorithm adds every next item (in non-increasing height) on the first level where it fits. If no level can accommodate the item, a new level is created and it is placed at the start of it.

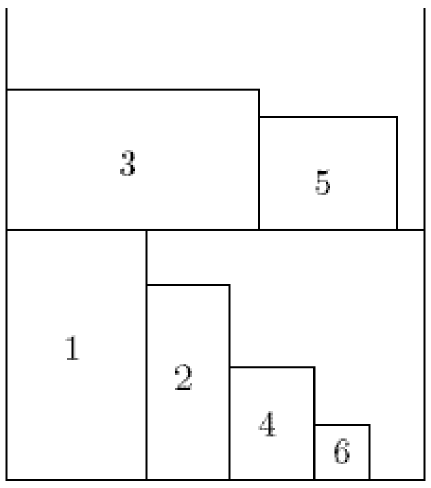

Figure 2.1: A resulting nest from the First fit algorithm [13]

Figure 2.1 shows how a nest will look like with the present 6 rectangles. The numbers in them represent how the items are ordered by height, 1 being the highest and 6 the 'shortest'. If a simple guillotine is used to cut the items out of the frame, then this is a valid solution but with the improvement in cutting machines this is no longer the case.

It is obvious that item 5 can fit in the first level of the nest. Even though that would not improve the overall coverage for this particular problem, if item 3 was not included item 5 would still be placed on a second level. Thus came the improvement shown in the Floor Ceiling algorithm.

2.2 Floor Ceiling

The Floor Ceiling algorithm consists of two phases. The first is constructing the levels as described in the First fit algorithm. The second is trying to place the last items at the ceilings of the existing levels starting from right to left. An example can be seen in figure 2.2.

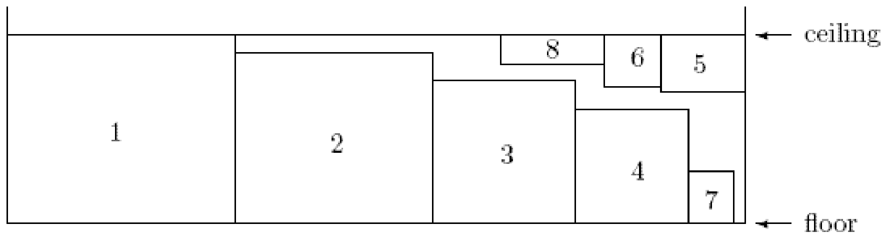

Figure 2.2: One resulting level from the Floor Ceiling algorithm [13]

As with every method, there are input instances that end up with a near optimal coverage. In this case if most items don't differ much in height, for example, if the highest item has a height H and the shortest, a height 0.6*H, there won't be any item that could fit on the ceiling of any level. This often occurred in practice during the research on this project.

From the initial tests of the selected methods for the project it was determined that the FC approach does not produce satisfactory results compared to the other methods which were still to be improved.

2.3 Solving both problem variations

Both methods above provide solutions in terms of levels with certain height that need to be nested. These levels can be arranged in different ways to form the nests for both instances of the problem, Sheet and Roll. This process is explained in detail in section 5.2 where the other method deals with *strips* rather than *levels.*

Chapter 3

Basic SMT approach

If a problem can be fully described by mathematical formulas expressing constraint over some variables, then it is possible to formulate it as a satisfiability problem which can be given to an SMT solver. There are a number of STM solvers developed through the years, that even have annual competitions [12] so they are constantly improving.

This section shows that using only SMT with a relatively simple description of the problem, not all cases are solvable, while section 4 gives a more complex version that is able to solve all cases but has other drawbacks.

The following explanation is given for the Sheet variation of the problem.

3.1 About SMT

The SMT solver takes as input a set of variables and formulas that describe a problem. The formulas normally express constraints on the variables' values, which can be either boolean, integer or real. Different formulas can be constructed using the variables, conjunctions (AND), disjunctions (OR), negations (NOT) and parentheses. The syntax of the tool also allows the use of linear inequalities.

To show some of the tool's capabilities, here is an example input that can be given:

```
integer x, integer y

satisfy: ((x ≥ 3) and (x + 2*y ≤ 10) and ((y == 4) or (y == 2)))
```

If the formulas can be satisfied, the tool returns a set of values for the variables satisfying the given conditions. The tool also reports if the formulas cannot be satisfied, letting the user know that there is no possible solution to the stated problem.

In the above example, there are two variables, namely x and y, whose values must be determined in such a way that the formula is *true*. If the SMT solver is given this exact problem it would yield the following solution:

```
sat
x=3, y=2
```

Stating that the formula is satisfiable and that the values for x and y are 3 and 2, respectively.

3.2 Rectangle Packing in SMT

The following description considers the simplest form of the rectangle packing problem where the question is: "Can all given items fit in the frame without overlapping and what would their positions be if they did?". Even though that question is more specific than the research question, it has been useful throughout the project and is even used in the final application shown in chapter 10.

The benefit of the SMT solver in that situation is that it will give a solution if and only if one exists and if it states that the set of formulas are unsatisfiable then a solution does not exist.

First let's look at the simplest variation of the problem, where a set of items needs to be placed in a frame with fixed dimensions (such as a sheet) and the question is whether it is possible to fit them all without overlapping.

Each item is described by a set of variables. For the i^{th} item the following are needed:

- its position, namely the x and y coordinates of its bottom left corner, (x_i and y_i, both integers)
- its dimensions: width (w_i, integer) and height (h_i, integer)

The reason for the item's dimensions being variables instead of constants is that rotation of the items is taken into account. Due to limitations of the machines that cut the materials after the printing is done, the allowed rotation is only 90°. In other words the width and height of the item can swap values if it would result in a better coverage.

Even though rotation is considered, there are cases in which a material does not allow for rotation, or an item's orientation is strictly preferred, thus there needs to be a check if the item can indeed be rotated.

These definitions are illustrated on figure 3.1.

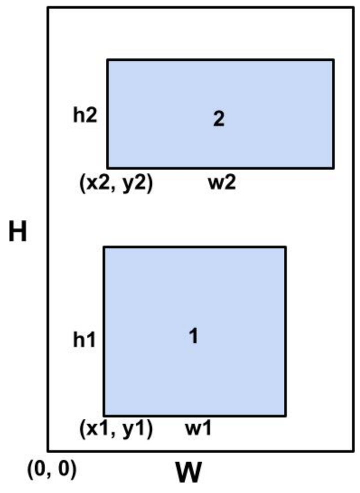

Figure 3.1: Two items in a frame with the variables describing the system.

Here are the constraints that use these variables:

- Item's dimensions are set. Let's say that item i has width $width_i$ and height $height_i$.
  For each item i:
  If the item can rotate :
  $((w_i == width_i \text{ and } h_i == height_i) \text{ or } (w_i == height_i \text{ and } h_i == width_i))$
  else:
  $(w_i == width_i \text{ and } h_i == height_i)$
- The items need to be inside the boundaries of the frame.
  For each item i:
  $((x_i + w_i \leq frame\_width) \text{ and } (x_i \geq 0) \text{ and } (y_i + h_i \leq frame\_width) \text{ and } (y_i \geq 0))$

- The items must not overlap with each other. That constraint basically states that item i needs to be either to the left, to the right, below or above item j.

  For each distinct pair of items i and j:
  $(((x_i + w_i) \leq x_j)$ or $(x_i \geq (x_j + w_j))$ or $((y_i + h_i) \leq y_j)$ or $(y_i \geq (y_j + h_j)))$

If there is at least one possible way of positioning the items in the frame according to these constraints then the solver will return the items' positions as a result.

Because of the time constraint, this method works for problems that have less than around 25 input items. This number actually differs for each distinct set of items, but during the tests that were conducted in this project the general case shows 25 as an acceptable threshold. If the items are more than that the SMT solver still produces correct results, but the time it takes to deliver them is unsatisfactory for the company's purposes.

This method is also limited by the fact that it can create only one nest. A partial solution to that issue is explained in the next chapter.

3.3 Solving both problem variations

Considering the Sheet problem, the above method gives a direct solution that can be used to construct the nest. Because the SMT solver answers if the items fit or not, their positions may seem randomly scattered in the sheet but that is still a valid solution on a sheet.

The nests for Roll problems have a maximum allowed height which can be used as a Sheet's height to make use of the above method but in order to optimize the height it should be lowered. The goal is to minimize the height of the roll so that the waste is minimized too. In that case a number of SMT calls can be used with different values for the frame's height until the optimal one is found. This can be achieved by doing a binary search over it, which is described in details in section 4.1.

Chapter 4

Best coverage SMT approach

As stated above, the Basic SMT approach is not enough to solve all instances of the problem. In fact, one of the most frequent problems observed in practice is the one where not all given items can fit in one specified frame.

There is a way to go around this restriction. Different combinations of items can be tested with the above approach as some of them will yield a satisfiable result and the one giving the best coverage will be chosen to be nested. And this process can continue until all items are used.

The issue here is that there is no way of knowing which combinations are good candidates to test with and testing all combinations, whose quantity is exponential to the number of input items, is not a viable approach as the time that would require is too high (also exponential).

A better solution is to encode the whole problem in SMT. Simply said, in that case, the solver needs to decide which of the items to use to get the best coverage. This decision can be expressed using additional variables and constraints.

The complete set of variables used to describe $item_i$ in this case is:

- its position, namely the x and y coordinates of its bottom left corner, (x_i and y_i, both integers)
- its dimensions: width (w_i, integer) and height (h_i, integer)
- if the item is present or not (p_i, boolean)
- item's area (a_i, integer)

To be able to maximize the coverage the additional variable must be added that represents the sum of the areas of all present items. Having a variable for each item's area helps with the constraint concerning this. As the areas are set to a specific number they do not slow down the process additionally.

With the addition of the new variables, the list of constraints expands to:

- Item's dimensions are set. Let's say that item i has width $width_i$ and height $height_i$.
  For each item i:
  If the item can rotate :
  $((w_i == width_i$ and $h_i == height_i)$ or $(w_i == height_i$ and $h_i == width_i))$

  else:
  $(w_i == width_i$ and $h_i == height_i)$
- The item's area is set. The area is calculated as $area_i = w_i * h_i$ beforehand.

  For each item i:
  $(a_i == area_i)$

- The sum of all present items must be higher than the minimum required area *min_fill_area*, which is later used to make the optimization of the coverage.

  $\sum_{i=0}^{n-1}(a_i * p_i) \geq min\_fill\_area$, where n is the number of input items and p_i is considered 0 if *False* and 1 if *True*.

  This summation cannot be directly expressed in the SMT syntax but there is a way around it. Z3 allows the use of an *If* function that has three parameters in it. The code looks like this:

  ```
  Sum([If(p_i, a_i, 0) for i in (0..n−1)]) ≥ min_fill_area
  ```

  The first parameter is the condition, that in this case is the presence boolean variable. If this condition is *True* then the second parameter is used in the summation (*Sum*) and if it is *False*, the third parameter is used. As a result, if an item is not present its area is taken as 0.

- The items need to be inside the boundaries of the frame.

  For each item i:
  $((x_i \leq frame\_width)$ and $(x_i \geq 0)$ and $(y_i \leq frame\_width)$ and $(y_i \geq 0))$

- If in a pair of items, both are present they must not overlap.

  For each pair of items i and j:
  $((p_i$ and $p_j) => ((x_i + w_i) \leq x_j)$ or $(x_i \geq (x_j + w_j))$ or $((y_i + h_i) \leq y_j)$ or $(y_i \geq (y_j + h_j)))$

4.1 Optimizing the coverage

Optimization of a variable is not directly implemented in the SMT solver. Instead a binary search is performed on the value of **min_fill_area**. Effectively the goal is to modify the coverage with each iteration of the binary search until the best one is reached.

Let's look at an example where the frame's width is 50 and its height is 100, which gives it an area of 5000.

The first value of min_fill_area is taken such that if the sum of the areas of all present images is greater than it, the coverage is higher than 50%. For a 50% coverage on 5000 area, the images' areas must sum to a total of 2500. This is the value that min_fill_area must have on the first step of the binary search.

If a solution is found then the next step is to increase the desired coverage to 75% (50% + 25%), where as if there is no solution the desired coverage drops to 25% (50% - 25%) The same calculation is made to determine the value of min_fill_area after the desired coverage is updated.

The amount with which the desired coverage changes each step is divided by two after each step. As shown above, it is at 25% at the start. Dividing this in two means that the second step in the search modifies the desired coverage with 12.5% and so on. A visual representation of this idea is shown in figure 4.1. This process repeats a few times until the point where further modification of min_fill_area is no longer beneficial.

After brief experimentation and consultation with the company's manager it was decided that a precision of more than 1% is unnecessary for this project, especially because in observed cases the solutions found at each stage usually have a higher coverage then the desired one. This renders every advancing tests redundant as it is highly likely that they will yield the exact same results, if successful.

Having the limit at 1% precision leaves the binary search with just 7 stages. Each stage being a call to the SMT solver with the only difference that the value of min_fill_area is updated with respect of the desired coverage in the stage. If, however, the 7 stages finish executing before the given timeout, the process can iterate more times for a guaranteed better precision.

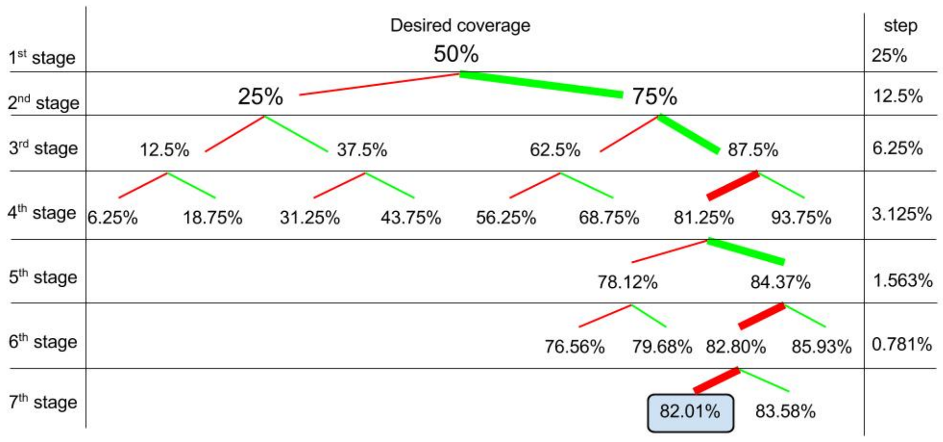

Figure 4.1: The progress of the desired coverage during the stages of the binary search. Green lines mean that a solution is found, red ones mean it isn't. The bold lines show the path taken to reach the final 82.01% in this example.

4.2 Additional SMT constraints

It is possible to narrow down the search space by adding more constraints, some of which are redundant.

4.2.1 Placing non-present items at a fixed position

Considering the presence variable, if an item is not present then there is no need for the solver to decide what its position is. This is obvious to a person, but it has not yet been stated in a form of a constraint. As the position itself does not matter, the non-present items can be placed arbitrarily on the sheet as the non-overlap constraint does not apply to them.

For each item i:
$((not\ p_i) => (x_i == 0 \text{ and } y_i == 0))$

Tests have proven that this additional constraint slightly improves performance depending on the particular set of input items.

4.2.2 Symmetry breaking

It often occurs that some items in the set have the same dimensions. While the SMT solver is doing its work it does not take into account that if these items swapped positions it would not change the outcome of the satisfiability check. This leads to the solver doing redundant tests which depending on the input item set may decrease the performance significantly. These duplicate tests can be avoided by adding a constraint that states: if two items have the same dimensions they can only have one possible position in respect to each other. This is a popular performance improvement method and it is called symmetry breaking [16].

Here is one way to implement it. If the width and height of $item_i$ and $item_j$ are the same it is implied that if their x's are equal then y_j has to be greater than y_i and if they aren't then x_j has to be greater than x_i

For every pair of items:
$(w_j == w_i \text{ and } h_j == h_i) => \text{If}(x_j == x_i) \text{ then } (y_j \geq y_i) \text{ else } (x_j > x_i)$

At first sight this seems overcomplicated but if anything is simplified or removed then some test cases may be falsely removed from the search space. For example, if one simply states that $(x_j > x_i)$ then it removes the test where the x's are equal. If it is $(x_j \geq x_i)$ then when the x's are equal the symmetry breaking is not working.

Tests prove that adding this constraint decreases the solver's performance not only in terms of speed but also in produced coverage, even though the opposite was expected. One reason for these unforeseen results may be the fact that this constraint is expressed in such a complicated way compared to the other formulas.

4.2.3 Finding suboptimal solutions

There are constraints that would further reduce the search space, but may lead to losing optimality. Such constraints may be beneficial to the method if the results still have satisfactory coverage and the run time is low enough. In other words, if the running time versus coverage trade-off is good enough, the addition is kept in the method.

Two such constraints are tested:

- Each item must have at least one side 'touching' another item's side.
- Each item must have one of its corners 'touching' another item's corner.

Both implementations of these constraints proved to only slow down the solver without improving the coverage of the found results.

4.3 Creating all nests for the problem

The above explanation provides a method that returns just one nest, whereas the full solution requires all items to be nested no matter in how many nests they must be placed.

This can still be done by just repeating the method enough times until all the items are processed. After the first items are nested, they are removed from the input item set. This way it is ensured that there will be no duplicate items in the resulting nests and a side benefit is that each consecutive call of the method deals with fewer items thus producing a result faster.

An obvious disadvantage of this process is that all the nests are created without "having the big picture" in mind. As the creation of each nest is independent of the others, the last nest often end up with a poor coverage that could be avoided if some items are swapped.

During the early stages of testing, this method gave unsatisfactory results which lead to a further improvement in the use of the SMT solver, described in the next chapter.

Chapter 5

SMT hybrid

The SMT only approach provides optimal solutions to the problem but it comes with a downside. For a high number of input items the problem becomes too complex to expect a result in a reasonable time. One of the goals of this project is to find a balance between speed and achieved coverage, which lead to this method

The SMT hybrid approach is exploiting the fact that SMT gives results in fractions of a second if the number of tested items is low. This allows several thousands of SMT calls to be made in less than a minute. A few ways of doing that were tested and the most successful one is described in this chapter. Even though it gives good results, it is clear that optimality is lost when it is being used.

The method consists of two main stages. The **first stage** tries to find strips of tightly fit items until every item is placed in a strip. A **strip** is defined as a sub-frame that has the width of the frame but its height is lower then the frame's. The height depends on the items tested in it.

The **second stage** aims to compress the resulting strips from the first stage. Depending on whether the printing is done on a sheet or on a roll, this process differs greatly. The exact details are given in a separate section for this method.

5.1 First stage

The process can be described with the pseudocode shown in Algorithm 1:

By using this algorithm, all the strips are created greedily, because the ones with a higher coverage are found earlier. Finding optimal strips one by one does not necessarily mean that when combined they give the optimal solution. This is the nature of all greedy approaches.

The backtracking algorithm, explained in the next chapter, faces the same issue, as seen in figure 6.3. The sheets shown in that example would be found as strips by the SMT hybrid algorithm thus it would fail to give the optimal solution in the same way.

Algorithm 1 SMT_hybrid_algorithm(base_required_coverage, max_items, step_down_percentage)

```
$required\_coverage \leftarrow base\_required\_coverage$
while $required\_coverage \geq 0$ do
  for all $n = 1$ to $max\_items$ do
    for all $combination$ in combinations($all\_items$, n) do
      Calculate required height of the strip, as $strip\_height$ by the following formula:
      $strip\_height = \frac{\text{the total area of all items in } combination}{\text{original frame's width} \times required\_coverage}$
      Generate a testing strip frame with height equal to $strip\_height$ and width equal to the original frame's width
      Use the basic SMT approach (from chapter 3) to test if the items in $combination$ can fit on the created strip frame as if it was a sheet.
      if The items fit then
        Save the strip.
        Remove $combination$ items from $all\_items$.
      end if
    end for
  end for
  $required\_coverage \leftarrow required\_coverage - step\_down\_percentage$
end while
Combine saved strips into larger strips with better coverage.
return The final set of strips
```

After extensive tests with the three parameters in SMT_hybrid_algorithm, namely base_required_coverage, max_items and step_down_percentage, these are the values that give best results:

```
base_required_coverage = 0.93
max_items = f(number of input items)
step_down_percentage = 0.07
```

If the base_required_coverage is taken too high then the chances of finding a strip with that coverage are low and most tests end up being a waste of time. On the other hand, if it is set too low then the tightly packed strips may be missed. This has been observed when there exists a possible strip with a high coverage and at least one strip with lower coverage, both of them being above that base_required_coverage, using roughly the same items, and the worse one is found first. This way the strip with the high coverage is removed from the possible testing combinations as some of its items are already in use.

By that explanation one can assume that there are too many conditions for this to happen frequently, but in practice it occurs surprisingly often.

The reasoning behind the value of *step_down_percentage* is basically the same.

The only parameter whose value has a visible relationship with the input, more precisely with the number of input items, is *max_items*. The main reason for this is the fact that the total number of tested combinations is solely dependant on the number of items.

If, for example, a problem instance has 50 items, the combinations of 2 items from them are 1225 in total, but the combinations of 6 items goes up to 15890700. It is unacceptable to spend too much time on one set of combinations. The other issue is that the basic SMT approach's running time is positively correlated to the numbers ot input items. As a result, an increase in the number of items of a problem instance there is an increase not only in the number of tests for each set of combinations but also their running time, which overall increases the method's execution exponentially.

This leads to the *max_items* parameter being a function of the number of input items instead of it being static as the other two. The exact implementation of the function was designed after a series of experiments. Due to the time limit of 40 seconds for a nest, set at the start of the project, the testing of a single set of combinations of *max_items* items should not take more than roughly 5 seconds. This way the method would have enough time to explore different combinations of items.

The results could be expressed as states shown in table 5.1.

Number of input items	max_items
160+	3
90-159	4
50-89	5
40-49	6
0-39	7

Table 5.1: The value of max_items as a function of the number of input items

The actual **code implementation** of this method has several ways of breaking out of the three loops to improve speed.

The method used to test if the items fit in the strip is the Basic SMT approach described in chapter 3.

In a perfect situation, every combination of items should be tested if it can create a strip with a high coverage. With item numbers high enough, this becomes impossible as the amount of combinations multiplied by the time an SMT call takes results in unacceptable running times of the method. This is solved by **limiting** how many combinations can be tested for each subset. The exact limit on the combinations is a function of the number of input items.

To explain this method in details, let's look at an example:

Assume the frame is a sheet and not all items can fit in it.

At the start of the algorithm the combinations of 2 items are tested one by one. A test consists of preparing the strip's size and checking if the items can fit in it. The size is determined by the coverage percentage that is being tested (*required_coverage*), which is high at the start and gets lowered after all tests with that coverage have finished.

Example: the combined area of the two tested items is 93 and the width of the frame is 20. If the currently tested coverage is 93% the area of the strip must be 100. This would give the stripe a height of 5 (as 5*20=100).

If the items fit in the strip, it is saved and the fitted items are removed from the available items (*all_items*). After all combinations of 2 items are checked, the method proceeds to test combinations of 3 items, and so on.

The maximum number of items in a tested strip depends on the initial amount of items, but it is no higher than 7 because then the SMT call no longer takes a fraction of a second.

The *required_coverage* starts at 93% and the above process is repeated while the it drops by 7% every time it is restarted.

This process is repeated as many times as needed until all items are placed in a strip, thus ending the **first stage**.

5.2 Second stage for a Sheet problem

After all the strips are formed what is left is to place them in as little number of sheets as possible. This problem is exactly a *bin packing problem* [9]. The heights of the strips are used as input whereas the height of the sheet is the capacity of the bin, which turns it into a one dimensional problem.

The way this is solved is not explained in detail in this paper partly because it is not in the scope of the research and partly because there is already a Python library [7] that solves it.

The library provides a few methods representing the most popular ways of solving the bin packing problem. Instead of relying on just one, all of them are called and the best result is used to create the sheets. Figure 5.1 shows how three strips that were chosen to fit in a frame create one nest.

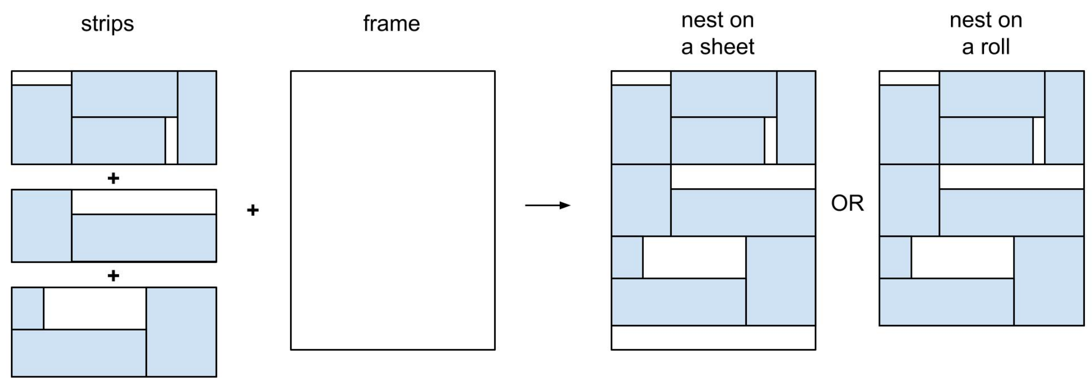

Figure 5.1: Strip packing, Stage 1, creation of one nest

5.3 Second stage for a Roll problem

Since the length of the frame is variable when working with a roll there is no strict need to solve a bin packing problem to decide which strips need to be combined to form the least number of nests. Having each strip as its own nest results in the same overall material use as if the strips were into larger nests.

The reason to actually group multiple strips into nests is that during production it is less tedious to handle less printing jobs. A perfect scenario is to place all strips in one long nest, but due to technical reasons there is a limit on the nest's length.

The same process can be applied for the Roll problem as it is for the Sheet problem, using the frame's maximum allowed length as the bin's capacity. The bin packing library is used to provides the best way to distribute strips along different nests. The only difference comes when creating the nest itself. As shown in figure 5.1, the length of the nest is determined by the place of the last items in it instead of the frame's specifications.

Chapter 6

Backtracking algorithm

When talking about optimization problems the most intuitive approach for a human is to test all possible options and see which one works best.

By testing every image on every available position it is guaranteed that all options are checked and the best one can be selected as a result. Such an algorithm can be implemented using a backtracking notion.

With an increase of the number of input items, the the time it would take to do that grows exponentially, therefore some limits are introduced to guarantee an acceptable solution in limited time. One increase in performance is gained by limiting this method to creating only one nest, similarly to the Best Coverage SMT approach (chapter 4), where if not all items can fit in the nest, a subset of them is chosen and placed in it instead. If the solution requires multiple nest, that process is repeated that many times. An even greater increase in performance is gained by greatly limiting available positions at which an item can be placed inside the sheet, which is described in section 6.1.

Detailed description and justification of how the full solution, of creating all necessary nests, works can be found in section 6.2.

The basic idea behind this method is that one of the items is placed in the bottom right corner of the frame, then another one is placed next to it, and then another one until no more items can be placed in the frame. When this state is reached the algorithm goes back one step and changes the last placed item with another one and continues testing. This goes on until all items are checked for the last position, then another step back is taken and the process is repeated.

Due to the high complexity of this method it is unrealistic to expect the software to go through all combinations within the desired time, especially for higher number of input items. Thus the optimal cases may not be reached in time.

One advantage of the method is that at any moment it could be stopped and the best found result is available. Increasing the time this method is allowed to work increases the chances of finding a better solution.

The main element of the method's implementation is a recursive function:

```
func backtrack(item, fitted_frame, pos)
```

It has three parameters, namely *item, fitted_frame* and *pos.* The function tests if the item (*item*) can be placed on position (*pos*) in a frame that may already contain some items (*fitted_frame*) placed there by previous calls of the function. The parameter *pos* defines a position with its coordinates as (x, y). If the item does not overlap with the others in the *fitted_frame* and stays within the boundaries of the frame it is placed there and the function precedes with the recursion.

The initial call of the function tests if the first item can fit in an empty frame if placed in its bottom left corner:

```
backtrack(<first item>, <empty frame>, <(0, 0)>)
```

Two global variables need to be initialised before starting the recursion. They hold the nest with the best coverage (*best_nest*) and the value of that coverage (*best_coverage*).

```
best_coverage ← 0
best_nest ← <empty frame>
```

The backtrack function can be described with the following pseudocode:

Algorithm 2 backtrack(item, fitted_frame, pos)

```
Try to place item in fitted_frame at position pos
if item fits then
  Keep item at position pos, adding it to fitted_frame
  if fitted_frame's coverage > best_coverage then
    best_nest ←fitted_frame
    best_coverage ← fitted_frame's coverage
  end if
  for all Available positions in fitted_frame as unused_pos do
    for all Other items as new_item do
      backtrack(new_item, fitted_frame, unused_pos)
    end for
  end for
end if
```

Next paragraph gives details on what is considered an *available position.*

6.1 Limiting the available positions

Let's define **test point** as a point in the frame (described by its (x, y) position) which is used to place items in the frame such that a corner of the item has the same coordinates as the test point.

By observing some manually fitted nests it becomes obvious that most items are usually placed with a corner 'touching' another item's corner. More precisely the item's bottom left corner touches either the bottom right or upper left corner of the other item. An example is shown on figure 6.1a.

Upon further investigation it became clear that this is not enough to find optimal solutions. As seen in figure 6.2, the upper left item would not be placed in the nest as no test point exists which would lead to the item being tested at that exact position.

In order to be able to cover problems, such as the one shown in figure 6.2 more test points need to be added per item. After some tests it was found that if a new test point is placed on the upper right corner of an item and the other items connects to it by their lower left corners instead of right, that would widen the possible nests that the method can find. Example is shown on figure 6.1b.

Although optimal solutions may still be missed, further increase in test points would only cause a worse exponential blow up.

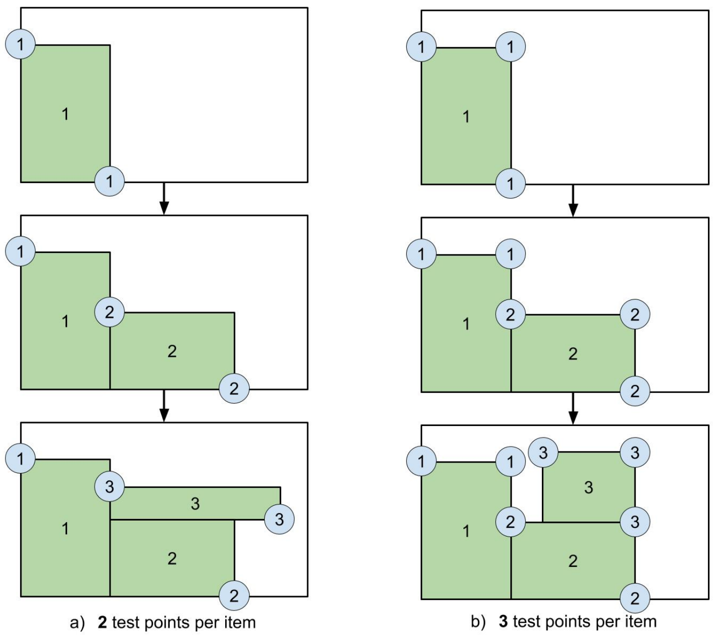

Figure 6.1: Three items with their corresponding test points.

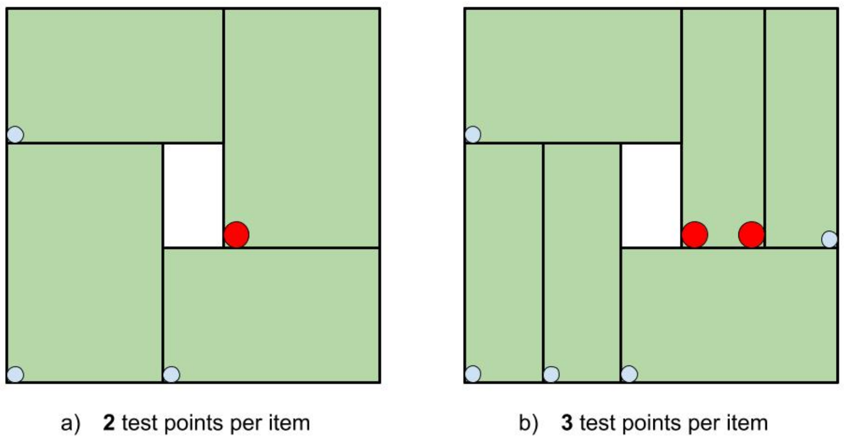

Figure 6.2: Examples of optimal solutions that can not be found by using 2 and 3 test points per item. The circles show which test point was used for the item. Red circles show the lack of one.

6.2 Creating all nests for the problem

In theory it is possible to build a backtracking algorithm that directly optimizes the overall coverage. Instead of stopping when an item cannot fit the frame, a new frame needs to be created and the recursion can continue as described. In practice though that proves to be too slow of a process and the results ended up being unsatisfactory.

Creating just one nest is done in a more appropriate time. After having one nest, the items it contains can be removed from the input of the problem and the same method can be run again with the new reduced set of items. This repeats until there are no more items to be nested.

6.3 Sorting the images first

There are ways of pushing good tests near the start of the algorithm in such a way that in a limited time a solution with an acceptable coverage is found even though it might not be optimal.

The simplest modification that helps with that is to sort the images by area before iterating over them. This way the method will start placing the largest items first.

In many cases the algorithm is unable to backtrack to the first items it placed due to the time limitation. This means that all the combinations that the algorithm would test have the biggest few items in the frame, thus the resulting nest will have them too. Even though this can be described as an inconvenience, it proved to be helpful after all.

Often not all items can be placed in one frame, but they all need to be placed somewhere eventually. In practice it is visible that in most cases the intuitive way to fit the items is to put the biggest one first and then find which other items can fit around it. Even if the best possible coverage can be achieved by placing only small items while skipping the largest item, it must eventually be placed in another nest. And if there are a few big items that were ignored in previous nests due to better coverage achieved by small items, the last few nests may end up having a low coverage. In extreme cases the smaller items could perfectly fit next to the larger ones, decreasing the overall number of nests, decreasing the overall waste. Such a case can be seen on figure 6.3.

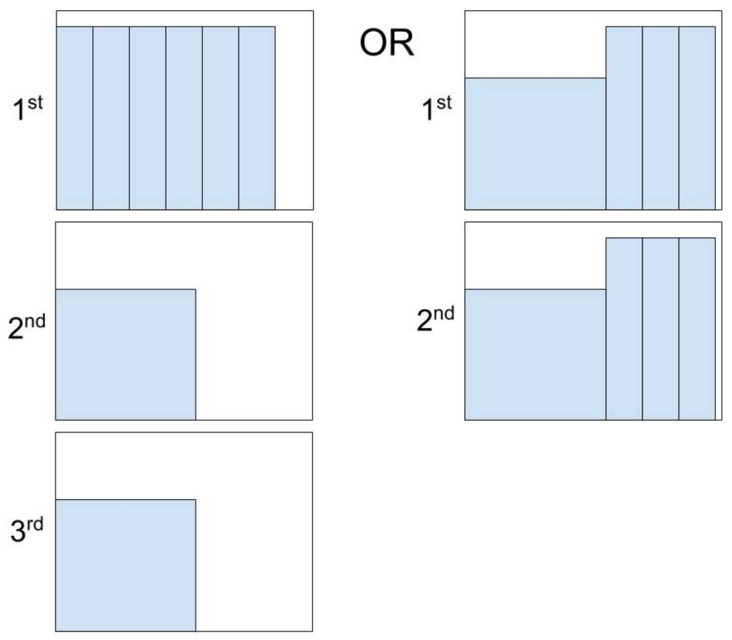

Figure 6.3: Example of overall improvement if large files are fitted in the first nests.

6.4 Randomizing image order

As mentioned earlier, due to time limitations the algorithm often does not reach a point where different starting items are tested. Even though ordering the items by size does help, testing a more diverse set of item arrangements gives a better chance at hitting an optimal or close to optimal solution.

If the item order is randomized once and then the algorithm runs in the exact same way there will be no improvement, as the starting items will not change again. This is fixed if the algorithm runs multiple times, randomizing the order before each run.

As the time limit still needs to be taken into account, each new algorithm run has a lowered timeout, the sum of all being equal to the complete timeout.

There is one inherent downside that comes with this method: using random values is unreliable. As the chances of this algorithm to find an optimal solution get higher, so do the chances of hitting only weak tests.

Testing the outcomes of this method are also unreliable. Results from benchmarking the randomized order vary significantly.

This is not a desired effect in practice as the person using this may never know if another run of the algorithm would provide a better solution. If multiple tests are done on the same problem with this method, the overall time spent on it increases too, which is another undesired effect.

6.5 Grouping items with the same size

In practice it is often the case that each image needs to be printed a number of times. For our problem each instance of the image is a different item that needs to be placed in a frame.

Here is an example: *img1* needs to be printed 10 times, *img2* and *img3* - 5 times each. A total of 20 items. Assuming that all three images have different dimensions then the 20 items can be viewed as 3 multisets each having a frequency equal to the number of items it contains. If *img2* and *img3* both had the same dimensions, the multisets would go down to 2 in this case.

How is this important for the backtracking algorithm? When iterating over all the items in the backtrack function (2) there will be a great deal of redundancy for the above example. If however, the iteration goes over the **multisets** instead of **individual items**, the number of executed tests is greatly decreased.

Of course, if all items have completely different dimensions this gives no benefits but it also has no disadvantages. Therefore this addition to the method is useful.

6.6 Differences if working on a roll

For this backtracking approach to work on problem instances that use rolled up material only one modification needs to be made. The method is run as if solving a Sheet problem and after the nests are created, the excess material at the end of each nest (if any) is trimmed. An example can be seen in figure 5.1, where the nest on a roll is a trimmed down version of the one for a sheet.

Chapter 7

Genetic algorithm

As already stated, an optimal solution to the rectangle packing problem can not be given in a timely manner, unless the problem instance consists of very few items, therefore approximate solutions are looked into. Genetic algorithms are a popular option in situations like this. Due to their random nature, genetic algorithms can never guarantee an optimal solution for a problem, but they often find a good solution if one exists.

An attractive quality of such an approach is that at any given time the process can be stopped and the best solution found so far is returned. As it is for the backtracking algorithm, the more time the method has, the better solutions it can provide.

There are three main elements that need to be decided on when implementing a genetic algorithm: the chromosome encoding, the fitness function and the way new generations are created. The effectiveness of the implementation greatly depends on these choices. A change in any of the three elements leads to different results, thus they need to be chosen in the most beneficial way for the specific problem at hand.

7.1 Overview of genetic algorithms

Genetic algorithms are designed to mimic evolutionary processes like 'survival of the fittest'. It is an iterative process in which each iteration represents a generation of individuals, each defined by a chromosome. Every individual in that generation is evaluated to find the best ones, labelled 'fittest', and they get to carry on their genes in the next iteration. The first generation consists of randomly generated individuals but every succeeding generation is created based on the fittest individuals from the previous one.

As in nature, each generation undergoes mutation to avoid the 'species' stagnating in a local extremum. If a mutation introduces a feature that improves the fitness of an individual it will subsequently be transferred to next generations.

7.2 The developed genetic algorithm

To be able to apply this approach for the rectangle packing problem, the chromosome structure must be chosen in such a way that it can be used to generate a nest. In the process of creating the nest, its coverage can be calculated and used as the fitness of the chromosome, thus the fitness function must be able to do that. The algorithm passed through several stages of development before the final characteristics of the method were decided on. The following is the description of the final version while the details on the decisions in the development process are given in the sections after that.

The chromosome is a string of elements (genes) each of which consists of a number and a letter *t* or *f* (as in *true* or *false*). The number in the gene represents a specific item from the inputs and the letter shows if the item is rotated or not, true (t) if it is or false (f) if it isn't. The input items are given a number, from 1 to n, n being the number of input items.

The string itself is an ordered list in which the numbers in each gene are unique. This is uncommon for genetic algorithms as most popular methods of generating the new chromosomes of each generation cannot be used and it is generally uncommon to have the requirement that genes must have unique values.

The order of the items in the chromosome is important as it is later used in the fitness function to create the nest. It is used to show the order of appearance of each item in the nest from left to right and bottom to top. This can be seen in figure 7.1 where two different chromosomes are shown plus the nests they would lead to.

An example of a chromosome, the first one from the figure 7.1, is [4t, 5t, 6f, 1f, 3f, 2f]. It has 6 genes whose numerical part is one of the numbers from 1 to 6 because the input items are 6 in total. First in the list is '4t', this means that the item at the furthest bottom left corner of the nest is item 4 and it is rotated by 90° because of the 't'. The second item in the list is '5t'. The algorithm that places the items by their order is explained in detail in section 7.6, but the general idea is that each next item is placed to the right of the previous ones or above it if there is no available position to the right. In this example, the 5^{th} item is placed to the right of the 4^{th} and it is also rotated as it has t in the gene. The third element in the order is item 6 and it is a gene with f so the item is placed with its original, unswapped dimensions, to the right or above the previous item. And the process continues for all genes in the chromosome in order.

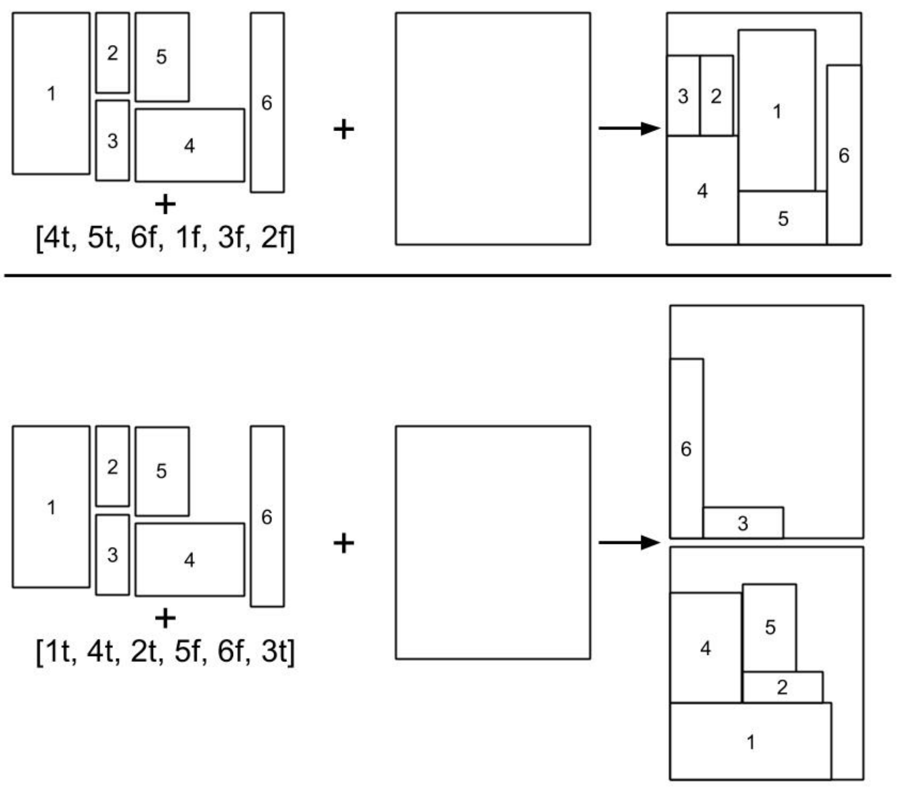

Figure 7.1: Nests created by the 3tp_ordered function with two different chromosomes.

By generating the nest, the fitness function can calculate the achieved coverage, which is used to determine which chromosomes are the fittest.

Using the nature analogy, two fit individuals (chromosomes) that are used to create a new one are referred to as **parents** and the new one is their **child**. The following is an in-depth description of the process of child chromosomes creation:

1. Two chromosomes are chosen at random from the fittest ones. In this example they are:

   ```
   parent 1 (p1) : [1t, 2t, 3t, 4t, 5t]
   parent 2 (p2) : [3t, 1t, 5t, 2t, 4t]
   ```

2. Two slicing points (or crossover points) are chosen at random. The genes between them will be used from p1.

3. In this example they are between 1t/2t and between 3t/4t, shown by '|' below:

   ```
   [1t | 2t, 3t | 4t, 5t]
   ```

4. The part from p1 that will be used keeping its order is:

   ```
   [2t, 3t]
   ```

5. Remove those numbers from p2 as repetitions are not allowed

   ```
   [XX, 1t, 5t, XX, 4t]
   ```

6. The genes that are left to be used from p2 are the following:

   ```
   [1t, 5t, 4t]
   ```

7. Create an empty child chromosome of the same length as the parents and place the part of p1 ([2t, 3t]) at a random position keeping the original order:

   ```
   [__, __, 2t, 3t, __]
   ```

8. Fill in the blanks in the child with the genes kept from p2 ([1t, 5t, 4t]), again keeping the order

   ```
   [1t, 5t,   ,   , 4t]
   +
   [__, __, 2t, 3t, __]
   =
   [1t, 5t, 2t, 3t, 4t] -> resulting child
   ```

The example above has '*t*' in all genes for ease of explanation and because this process does not modify the rotation of objects. It only modifies the order of the genes. The orientation is only changed by **mutation**.

Each individual in the newly created generation has a chance to **mutate**. The process is explained fully in section 7.5, but the general idea is that when a chromosome mutates, one or more of its genes changes. Either some genes swap positions or a gene changes its rotation value (from t to f and vice versa).

That being said, the next sections are a description of how the decisions that lead to this final look of the algorithm were made.

7.3 Choice of chromosome encoding

The natural way of describing a chromosome is by a sequence, with its elements being binary, integers or characters. How that sequence is interpreted afterwards depends on the fitness function of choice.

During the literature study two main ideas for the chromosome encoding representations were found: order of appearance [6] and splicing tree [15]. The order is a sequence of integers, each integer referencing a different item while the splicing tree chromosome is given as a sequence of characters that describe the tree itself. The second method has one drawback - there are optimal nests that cannot be created using its chromosomes. A way of improving that was not found during this project, thus no further work using the tree representation has been done.

On the other hand, the description by order of appearance, from left to right and bottom to top, must be able to provide a way of finding optimal solutions to the problem. The issue is: how to translate that sequence into an actual nest. There are papers that describe the use of one of the pre-existing direct approaches, such as First-Fit or Floor-Ceiling. These approaches usually sort the items before placing them in the nests but to be able to use them in the genetic algorithm that sorting is skipped and the order described in the chromosome is used instead.

Therefore, a sequence giving the order of the items is used for the genetic algorithm implementation, with a slight modification : the genes not only carry the item number but also if the item is rotated or not.

For example: a chromosome [3f, 1t, 2f] :

From left to right and bottom to top, the first seen item is $item_3$, second is $item_1$ and last is $item_2$. The letter after the number represents if the item has to be rotated: True (t) or False (f). In this case, $item_3$ is not rotated while $item_1$ is.

The additional information about the rotation of the item is later used in the fitness function described in section 7.6.

The size of the search-space created using that chromosome encoding grows exponentially with the size of the input items. The exact size is shown in the following formula:

$$\boxed{\texttt{number of valid chromosomes} = \texttt{n! * } 2^n\texttt{, where n is the number of input items}}$$

7.4 Creating new generations

Each new generation consists of the 'offspring' of the previous one's fittest individuals, in the hope that their beneficial traits will be passed on.

In most studied instances of problems that can be solved by genetic algorithms utilising a sequence representation of the chromosome, the creation of the offspring is done by a simple crossover function.

The simplest way of generating an offspring is to choose two parenting chromosomes then a random point in them and coping everything before this point from the first parent and then coping everything after the crossover point from the other parent. Here is an example:

Chromosome of Parent 1	[a, b, a, c, d, d, c]
Chromosome of Parent 2	[a, a, a, b, b, b, d]
Possible Child 1	[a, b, a, c, d ‖ b, d]
Possible Child 2	[a, b, a ‖ b, b, b, d]
Possible Child 3	[a ‖ a, a, b, b, b, d]

Table 7.1: Simple example of a crossover between two parents. The different children shown have used a different crossover point, shown as the symbol '‖' in them.

Such a method is impossible to apply in the rectangle packing problem as the sequence chosen represents the order of items, thus the elements of the chromosome need to be unique.

A modification needs to be made in order to work with unique genes. The genes from the first parent can still be kept in the same order and position but, to prevent repetition of genes, the ones used from parent 1 need to be removed from parent 2. What remains can then be added to parent 1's part and thus a child with all unique genes is created. Although only the first part directly inherits the traits of a parent, the second part still preserves a certain order from the other parent.

Chromosome of Parent 1	[a, b, c, d, e]
Chromosome of Parent 2	[c, a, e, b, d]
Possible Child	[a, b ‖ c, e, d]

Table 7.2: Simple example of a crossover for parents with unique elements in the chromosome. The crossover point is shown as the symbol '‖' in the child.

As the beneficial trait in a parent may be represented by a cluster of items, it could be at any position in the chromosome, not just at the start or end. To be able to extract such traits from a parent two crossover points need to be selected and the part in between them can be used.

For example, in the chromosome [a, b, c, d, e] let's say that only *c* and *d* work together nicely. If only one crossover point is used, to be able to preserve that trait in next generations if this is a parent 1 chromosome then all 4 starting genes need to be passed on to the child. Instead, two points can be used to cut only the [c, d] part.

Chromosome of Parent 1	[a, b, c, d, e]
Chromosome of Parent 2	[c, a, e, b, d]
Possible Child	[a‖ c, d‖ e, b]

Table 7.3: Simple example of a crossover for parents with unique elements in the chromosome using **two** crossover points, shown as the symbol '‖' in the child.

Furthermore the position of the [c, d] part does not necessarily need to be at the same position in the child as it was in the parent.

The different crossover methods explained in this section have been tested, proving that the final chosen crossover function gives an improvement in performance over the rest. Detailed data can be found in section 9.6

A summary of the **final crossover function**:
Two chromosomes are randomly selected as parent 1 and parent 2. Two random points are chosen in parent 1 and the genes in between those points are placed consecutively at a random position in the child. The selected genes from parent 1 are removed from parent 2 and what is left is then used to fill the rest of the child's chromosome, keeping the order of the genes of parent 2 the same.

An example can be seen in table 7.3.

An in-depth explanation can be found at the end of section 7.2.

7.5 Mutation of the new generation

When using the fittest individuals to create the succeeding generations there is the tendency that the population will go in one direction and not change much after a number of iterations.

A popular way of avoiding such a stagnation is to introduce mutation of the offspring. Given the chosen chromosome encoding there are limited possibilities of mutation. Here is a list of them:

1. Randomly chose a gene and change its rotation.
2. Switching the positions of two randomly chosen genes.
3. Switching the positions of two neighbouring genes, again at random.

Even though mutation 2, switching two random genes, may result in switching two neighbouring genes effectively making it the same as mutation 3, both mutations are necessary. The reason is that both play a different role in evolution. Mutation 3 usually results in similar nests because it swaps items close to each other thus providing a sort of 'fine tuning'. On the other hand, mutation 2 has a higher chance of greatly altering the nest's appearance, thus its purpose is to help the population escape a local extremum, if it is in one.

7.6 Choice of fitness function

A fitness function's main role is to determine the fitness of a given chromosome and in this problem the most suitable metric for that is the achieved coverage. In order to calculate that coverage the chromosome needs to be 'translated' into nests and while section 7.3 covers the chromosome's encoding, this section explain exactly how that 'translation' is done.

The implementation of the Genetic Algorithm started with the use of FF and FC (chapter 2) as fitness functions, witch is not a new concept. The goal was to improve that.

Test results showed that FC gives slightly better results than FF but it is more than 10 times slower on average. The better coverage is expected as FC is an improvement over FF but the decrease in speed is not. Further inspection of the two algorithms proved how little is needed to slow the whole method.

FC basically runs FF first and then goes over all items, starting from the smallest, and iteratively checks if it would fit in any other level's ceiling. Each test if an item fits on a level is done by iterating over all the items in it and checking if there is any overlapping. This results in a substantial increase of computation which is only visible if done a great number of times, as a single call of both functions only takes a fraction of a second.

For example, if a genetic algorithm, for a problem with 150 items, is ran having 100 individuals as its population and executing in 300 generations, the fitness function will be called (150*100*300) 4500000 times. Because of the time limitation a fitness function must be as quick as possible while still being able to build nest that are close to optimal, if not optimal.

After advances while working on the backtracking algorithm one such function became available. It utilizes the "test points" idea, described in section 6.1.

The pseudocode for that fitness function, called 3tp_ordered can be found in Algorithm 3 below. An important aspect of it is that whenever a test point is used to test an item, it is being removed from the whole list of test points, effectively using every test point only once during the function. This is what makes this function so quick, there are no opportunities for prolonged loop executions.

Figure 7.1 shows the nests that this function would create, given two different example chromosomes with the same input items and frame size.

Algorithm 3 3tp_ordered(chromosome, images)

```
$nest$ ← empty nest.
$nests$ ← empty list
$test\_points$ ← $[(0, 0),]$
for all $gene$ in chromosome do
   $image$ ← images[$gene.number$]
   if $gene.rotated$ then
      rotate $image$
   end if
   $require\_new\_nest$ ← True
   $test\_points$ is sorted by (x, y)
   for all $tp$ in $test\_points$ do
      remove $tp$ from $test\_points$
      if $image$ fits in $nest$ at point $tp$ then
         place $image$ in $nest$ there
         $require\_new\_nest$ ← False
         Add the 3 new test points created by placing $image$ to $test\_points$
      end if
   end for
   if $require\_new\_nest$ then
      append $nest$ to $nests$
      $nest$ ← empty nest.
      place $image$ in $nest$ at position (0, 0)
      Add the 3 new test points created by placing $image$ to $test\_points$
   end if
end for
append $nest$ to $nests$
calculate achieved coverage as $nests\_coverage$
return $nests$, $nests\_coverage$
```

7.7 Choice of population size

Parallel to making decisions on the chromosome's encoding and the fitness function, the size of the population is experimented with. However, a test specifically designed to determine the most appropriate size was done at a later stage of the method's development. The exact results of that test can be found in section 9.5.

An initial value of 30 was arbitrarily chosen for the first tests with the algorithm and it increased to 100 after the tests.

7.8 Difference for sheet and roll problems

The fitness function (from Algorithm 3) regards roll problems as sheet ones, using the maximum height of the roll, to be the height of the sheet. The difference comes at the end when the coverage is calculated. Then if a roll problem is considered the excess length of the sheet nests is removed to fit the actual problem.

A greater difference comes when deciding which individuals are the fittest:

- for a roll problem the population is sorted by the coverage each individual results in.
- for a sheet problem the sorting is done using the number of created nests as a primary key and the coverage that would have been achieved if it was a roll problem, as a secondary key.

The reason why this is done for the sheet problem is that, even though the main goal of solving it is to minimize the number of nests created, it does not provide a fine-grained distinction between which individual is performing better. To get the sheet nests that were best fitted, they could be considered as rolls just to get a number that represents a secondary criteria for the sorting.

7.9 Greedy creation of nests (one nest at a time)

Intuitively, if the approach takes the arrangement of all nests in mind it must give the best solution to the problem, even if the individual nests may look suboptimal. And that approach was presenting good results until one of the tests included more instances with large number of input items. In that test the Floor Ceiling method was added just to prove how much better the new GA approach is.

Even though on average GA performed best, the test showed a major flaw. At high number of items Floor Ceiling gave better results with some cases having more than 10% more coverage than any other method.

After examining the nests created by both methods for these cases, it was apparent that the greedy approach used in Floor Ceiling worked well, while the nests created by GA were inefficient, with low coverage.

On the other hand, GA gives close to optimal solutions for cases with smaller number of items, so the problem must come from the increased complexity at high number of inputs. To combine the advantages of both methods a greedy GA method is created.

The 3tp_ordered algorithm (3) is edited to create only one nest instead of all of them, which is a simplification to the function that even makes it faster. That way when a chromosome is evaluated by it, the nest coverage that is returned is only based on the first nest that would have been created. Which results in the Genetic Algorithm to return one nest with a very good coverage, disregarding the files that are not included in it.

To create the second nest, the items used in the first are removed from the original input items and the process is repeated, creating another well optimized nest. This is done until there are no more images to be nested.

Solving the problem by such a greedy approach leads to issues such as the one described in figure 6.3, where creating the best nests first gives a worse result. That issue gets less and less relevant with the increase in nests created, because the bad coverage in the last nests is compensated by the other better nests.

7.9.1 Fitting in the given time limit

The Genetic Algorithm can be given a maximum time it can run and if the algorithm is not finished by the end of that time it just returns the best found solution so far. In the greedy approach using GA this can not be directly used as it uses numerous calls of the Genetic Algorithm and if not all of them are finished in time the final solution is incomplete, thus invalid. So it must be ensured that each GA call is limited in such a way that the remaining calls still have some time to produce good nests.

The method described in algorithm 4 has a pessimistic approach on predicting how many nests will be created to make sure the time limit is met while all nests are created by the GA. If, however, at the end of the time there are still un-nested items the process is finished by using Floor Ceiling because of its fast execution.

The more important aspect of this algorithm is the **dynamic** prediction of the expected nests. With each iteration, that number is recalculated by dividing the combined area of all items that are left by the area that is expected to be covered by images in the next frames. As experiments have shown that when creating multiple nests the coverages are usually similar, the calculation can use the coverage of the last nest as prediction.

Often this leads to the starting nests receiving a higher time limit than the last ones but that is not a problem as the last ones are working with few input items and generally require less time to complete.

Assume that the function time_left() used in the pseudocode below gives how much time is left until the time limit is reached and the function GA_one_nest_only(*items*, *separate_timeout*) creates one nest using GA and returns it.

Algorithm 4 Greedy_GA(*items*)

```
last_coverage = 80%
nests ← empty list
while items is not empty do
  if time_left() ≤ 0 then
    break
  end if
  expected_nests = $\left\lceil \frac{\text{combined areas of } items}{\text{frame's area} * last\_coverage} \right\rceil$
  separate_timeout = time_left()/expected_nests
  nest = GA_one_nest_only(items, separate_timeout)
  remove the newly nested items from items
  add nest to nests
  last_coverage = nest's coverage
end while
if items is not empty then
  complete nests using Floor Ceiling
end if
return nests
```

7.10 Disadvantage

Even though the Genetic Algorithm may provide satisfactory solutions to the problem quicker then most methods, testing and gathering results for it is difficult. There are instances of the problem for which this genetic algorithm implementation provides a coverage of 83% once and if it is tested again, provide 91% coverage. That particular instance has 120 items in it and a person will never be able to achieve even 80% coverage in the 20 seconds it took this method. Still, a fluctuation of this range is undesired as it lacks consistency, which is one of the initial goals of this project.

To be able to evaluate the method against the other more consistent ones a number of tests are performed, taking the average as the result. Even then, it is taken into account that the average is still an estimation.

Chapter 8

Particle Swarm Optimization

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Eberhart and Kennedy in 1995, inspired by social behaviour of bird flocking or fish schooling.

Similarly to the Genetic Algorithm discussed above, the Particle Swarm Optimization is a type of evolutionary algorithm. Therefore, it uses chromosomes that describe a possible solution which is then constructed and evaluated by a fitness function. The difference lies in the way the search-space is explored. While the Genetic Algorithm has a population that evolves with each generation and the individuals in it represent a point in that search-space, in PSO the population consists of particles that do not 'evolve' but rather 'move' in the search-space. Each particle's chromosome represents its location and by moving that location changes to a neighbouring one.

The success of PSO is best presented with a one or two dimensional search-space as anything above two dimensions is difficult to display on screen or paper. Figure 8.1 shows a graph representation of an example one dimensional search-space where the line shows its fitness.

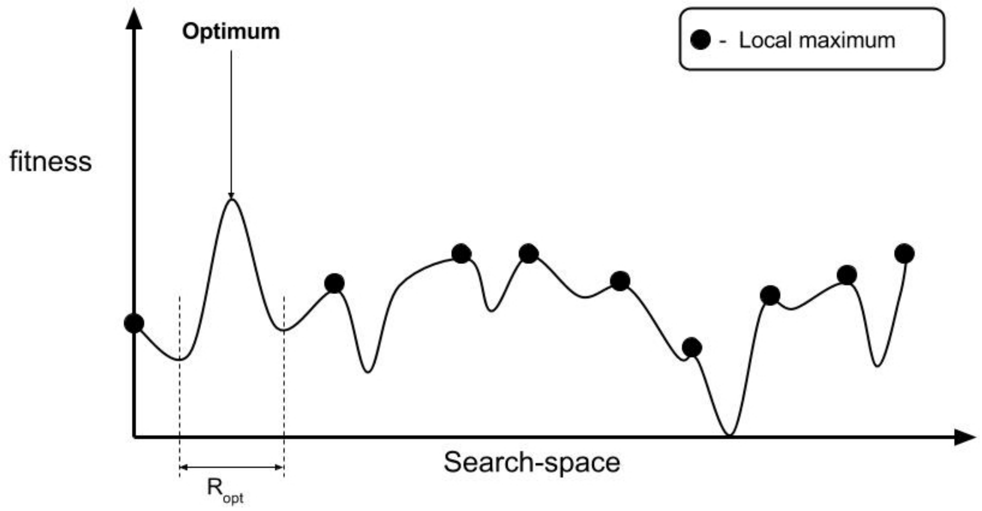

Figure 8.1: One-dimensional search space that can be traversed using PSO.

The initialization of the method spreads particles randomly in the search-space. The iteration after that aims to move them to a location with a better fitness. As this example is one dimensional, a particle can only move in two directions, left or right, and the one that gives a better fitness is chosen to be its next location. In a way, this makes the particles climb in search of a local maximum and when they find one, they settle and report it.

The particles are initially spread without knowing what fitness distribution looks like in the search-space, otherwise that would defeat the purpose of searching, so there is a chance that no particle is placed near the global optimum. For this example, in figure 8.1,if a particle is placed in the range R_{opt} then it would start climbing that 'hill' and the optimal solution for that problem would be found. Otherwise, one of the other local maximums would be the result.

8.1 Adapting it to the rectangle packing problem

During the implementation of the Genetic Algorithm, the chromosome structure and the fitness function proved to give close to optimal results for this problem. Even though that doesn't necessarily mean that they will give good results for PSO, they are still good candidates. And because this method was studied less extensively during the project no better candidates were found.

The more important aspect of PSO is deciding how to traverse the search-space. A particle is generally allowed to move only to its neighbouring positions, but what defines that neighbourhood for this particular problem is open for interpretation. Of course, it heavily depends on the chromosome encoding as it is used to determine the search-space itself.

A neighbouring location can be defined as a one that is one 'step' away and in this case that 'step' can be one change in the chromosome. The smallest changes are described in section 7.5 because they are used as mutations to the population.

Example:

Lets say a particle is at location $[2t, 1t, 3t]$.

It's neighbours would be all locations that result from all possible gene swaps and rotation changes.

neighbours:

- $[1t, 2t, 3t], [2t, 3t, 1t], [3t, 1t, 2t]$ from gene swaps
- $[2f, 1t, 3t], [2t, 1f, 3t], [2t, 1t, 3f]$ from rotation changes

They are all tested to see which one is the fittest and the particle moves can move there. This process repeats with each iteration until all neighbours have a lower coverage or have been previously visited, thus finding a local maximum and settling.

8.2 Particle Communication

Having the particles 'communicate' with each other is a promising idea, but implementing it proved that it has some limitations. There are two popular ways of creating that communication.

On one hand, the whole search-space can be mapped and each 'location', described by a unique chromosome, can have a boolean variable associated with it that states if a particle has passed through it or not. That is infeasible because it would require an exponential amount of memory to create that map.

On the other hand, a history can be kept that stores which 'locations' in the search-space were visited. This way the memory requirement is lowered, but it comes with its own disadvantages. As a particle should not go where another has already been, to avoid redundancy, each test of a particle's neighbour has to first be looked up in the history to see if it is available. With time that history can become so large that the process of looking up items from it becomes the bottle-neck of the method.

The only way of removing that inconvenience is know at which point communication no longer gives benefits and stop using it then. The particles may duplicate some of their paths but they would be searching, instead of spending most of the time on the communication. Absence of that history does lead to a lowered performance of PSO.

8.3 Search-Space Disadvantage

The more particles are deployed in the search space, the more local maximums are found, the higher the chances of finding one that is close to the optimum. Due to the nature of the problem and, consequently, the chromosome structure, the search-space's size is exponential. Using the formula from 7.3, here are some examples:

- for 3 items the size is 48
- for 5 items the size is 3840
- for 10 items the size is 3715891200
- for 100 items the size is 1.18305 e^{188}

The number of local maximums increases with the increase of the size too, so in order to continue providing good results the algorithm needs to use more particles. As it can not use an exponential number of particles, the results worsen with increase in size and that can be seen in the overall results obtained in the final benchmarks seen in table 9.1.

Chapter 9

Results and Discussion

Due to the nature of the problem, after every change in a method, no matter how insignificant it may seem, a benchmark test should be done to determine weather the change has lead to an improvement in the given method. The inputs that are used at the start of the project are different from the ones that are used in the final evaluation. The benchmark inputs developed through the project had to be adjusted due to unaccounted variations of the problem. Furthermore, some experiments target very specific test cases, thus a set of inputs different from the general one were used.

- It must not be assumed that the test results from the different tables shown in this chapter are based on the same inputs. This means that the results from two separate tables may not be compared meaningfully unless otherwise specified in their description.

9.1 Final results, Benchmark from practice

A benchmark set of inputs is used in order to chose a method that generally has the best performance. That set consists of 120 different problems that had to be solved in the company and they are chosen to represent the working conditions observed over the course of the project. The main property that dictated the chosen inputs is the number of items they have:

- 42 cases (35% of all) with 2 to 10 items
- 20 cases (16.67% of all) with 11 to 20 items
- 14 cases (11.67% of all) with 21 to 30 items
- 10 cases (8.33% of all) with 31 to 45 items
- 6 cases (5% of all) with 46 to 70 items
- 28 cases (23.33% of all) with 71 to 280 items

To make the results more meaningful the following changes are made.

First, the distribution of the problems, depending on the number of input items they have, is adjusted. In the original set of samples provided by the company, more than 75% of the problems have less than 11 items. Even though this benchmark was intended to represent the reality in practice, test showed that most methods produce good results with that little input items. If there is hardly any difference in the methods for 75% of the test cases then the overall average would have little meaning and the in depth examination of the tests would leave less samples that give valuable information. This being said, the problems with less than 11 items are not useless as there are occasional instances that get different solutions from different methods. So instead of not including any of them in the final benchmark, their numbers were decreased to only amount to 35% of all tests. As a result, the numbers of test cases in the other categories are increased, most noticeably in the last one, with the largest numbers of input items.

Second, even though originally about half of the inputs are done using sheets, rolls produce more accurate results, thus all problems were treated as roll problems. The difference in results between the two problems is mainly the granularity. For example, if two methods give 3 sheet nests as results to a problem, considering them as equal, but if the same inputs were set on roll the results can prove that one method gives a better solution than the other. That was also observed in the initial tests with these 120 inputs as the sheet problems either gave big result differences or none at all, which lead to that improvement in testing.

After the the final adjustments on the benchmark inputs, all relevant methods are run on them. The results can be seen in table 9.1.

Tested Method	**Average Time (s)**	**Average Coverage (%)**
Genetic Algorithm (one nest at a time)	32.9657	84.9156
Genetic Algorithm (all nests at once)	53.3880	83.2479
SMT Hybrid (all nests at once)	83.0958	81.8224
Backtracking	95.1452	80.3591
SMT Hybrid (one nest at a time)	189.4323	76.5153
Floor Ceiling	0.0406	76.4821
Particle Swarm Optimization	179.3735	63.6756
Using the best of all methods	-	86.1259
Final combination of methods	35.0357	86.0061

Table 9.1: Final results from all implemented methods. Based on the final benchmark tests of 120 problem instances from practice.

The method that presented best average coverage is the **Genetic Algorithm (one nest at a time)**, described in section 7.9. In other words, if only one method needs to be used to solve the company's nesting problems, that is the method of choice. Even the Average time it took is lower than most.

During the process of development, the Genetic Algorithm generally started to show the best results after the fitness function was changed appropriately and the crossover function was improved.

Next is the **Genetic Algorithm (all nests at once)**, described extensively in section 7, which is the more pure form of the Genetic Algorithm as it considers all items while creating the final solution. Due to the complexity of the problem at cases with high input numbers, it fails to provide an overall better average coverage.

Following closely, is the **SMT hybrid** method (section 5), more precisely the version of it that generates all tightly fit strips and then arranges them in nests. Its counterpart, one nest at a time, is fourth in the list and uses twice the time in average. This fact indicates that considering the problem as a whole gives an advantage while using SMT contrary to what the GA methods show.

On the other hand there is the greedy **Backtracking algorithm** (section 6) that also uses the same idea of generating one nest at a time, because dealing with all nests simultaneously is extremely slow and regularly produces worse results than what Floor Ceiling does. Thus the second variation of the Backtracking algorithm is not even included in the final consideration.

The **Floor Ceiling method** (section 2) is an outdated method that used to be a standard years ago and is used as a comparison to the others. The only advantage it still has over the rest of the methods is that it can handle tens of problem cases in a second, making it, on average, a 1000 times faster than the Genetic Algorithm.

The **Particle Swarm Optimization** (section 8) showed promising signs at the start of the research but even though it got improved over time it is unable to give satisfactory results in the end. The fact that it shares most of the ideas with the Genetic Algorithm, namely the same fitness function and chromosome structure, shows how important the evolution of the chromosomes really is. The strategy of traversing the search space using this method proved to be inefficient for this type of problems.

Next in table 9.1 is the **"Using the best of all methods"** which is a theoretical best case scenario for these test cases, if all methods were run and the best result was chosen at the end.

Lastly, the result obtained by running the **Final combination of methods**, which is implemented in the final application of this project (more details in chapter 10). As the name suggests, it benefits from all viable methods to find a better solution than if using just one. The result proves that to be an improvement, even though the theoretical best is not reached. The inability to reach those results comes from the time constraint, because in the final combination all the methods share the same time limit instead of each having it separately.

A summary of the observations on a finer scale can be seen in figure 9.1. This information is crucial in the final decision on how to use the different methods in cooperation to provide the best solutions. The frequencies in each group adds up to more than 100% because a number of methods can find the same coverage on a test, which happens more frequently with smaller number of items.

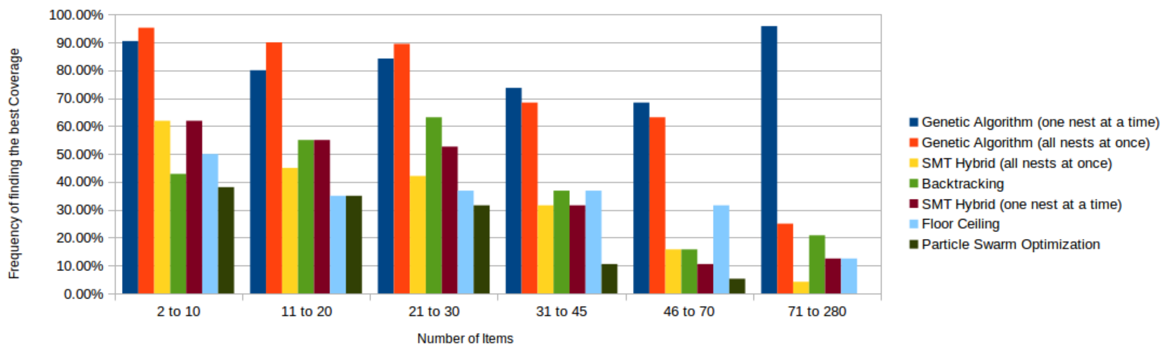

Figure 9.1: The frequency with which a method has found the best solution in the benchmark from practice.

Detailed results for each test case can be seen in appendix A. By examining the results case by case it can be seen that the Genetic Algorithm finds the best solutions 97 times, while both SMT hybrid methods and the Backtracking algorithm find them around 52 times (52,54 and 51 more precisely).

There are 11 cases where the Backtracking algorithm provides the best solution, making it a good candidate to be used in parallel with the leading method. However, it will not be used as the improvement that is gained is at a great cost of time.

9.2 Improvement in the company

Near the end of this research, one of the workers was asked to complete some tasks two different ways: using the software designed in this project and the way they used to work before it. Only five tasks were recorded because it disturbed the work flow and the worker stated the results will repeat themselves, with small variances.

The process, used before this project, was aided by a commercial software that helps order the images as well. Its main disadvantage is that that software was created for general use. This creates overhead due to the need of setting the options manually with the provided interface. Additionally, each image is selected and added to the nest manually, thus further slowing the process.

After the images are picked and everything is set, the nest is arranged within a second, even for cases with a higher image count. The only methods, found during this research, to accomplish that are the direct approaches. However, they aim for a quick solution regardless if it close to optimal.

To have a completely manual reference on the tested problems, solutions were found by just using pen, paper and a calculator to verify that there is no overlapping. The process is slow and cumbersome especially for cases with more than 8 items in them. It was similarly to the idea of backtracking with smarter choices of combinations, which most of the time lead to the same results found by this project's software. Having a small number of items gives the opportunity to check all possible arrangements, thus finding the best one easily, but slowly. With a higher item count the best arrangement becomes easier to miss, as seen in the results, shown in table 9.2.

	Commercial Software		Proposed use of GA and SMT		Completely Manual	
№ items	time (s)	coverage (%)	time (s)	coverage (%)	time (s)	coverage (%)
4	25	**75.7**	**6**	**75.7**	120	**75.7**
5	25	78.6	**10**	**80.3**	120	**80.3**
8	28	89.5	**17**	**93.9**	420	89.5
10	30	87.1	**15**	**92.0**	800	**92.0**
14	**36**	83.7	39	**87.5**	920	86.4
Average	28	82.94	**17.4**	**85.7**	492	84.78

Table 9.2: Time and coverage results comparing a previously used commercial software in BigImpact, the currently used software designed during this project and solving the problem manually without any software help

On average, using the newly developed software gives a coverage improvement of 2.76%, while making the process faster for problems with a lower item count. Even though it is slower for some inputs it is still acceptable by the company's requirements.

Another noticeable improvement over the new system is that the new process is more automated than it used to be that reduces the possibility of human errors.

9.3 Benchmark with random inputs only

Working with test cases observed in practice is proof that these methods are an improvement for the specific company. To determine if these findings hold for general rectangle packing problems a different set of tests is required. These tests are generated pseudo-randomly, in other words, all random values have lower and upper limits.

The frame's width is between 1000 and 4000, its height is between 1000 and 10000. Each item's width is between 20 and the frame's width, while the height is between 20 and frame's height. The number of input item ranges from 2 to 200. Instances where one item takes up a whole frame are not practical for the research, therefore, items' dimensions are chosen at random with a Gaussian distribution.

For the averages to be useful the number of tests must be large enough. In this case 300 problems were generated and later examined to determine what they would look like if only a part of them is used. For the generated tests even using the first 100 test cases the same conclusions can be made regarding the tested algorithms, which conceivably suggest that the 120 instances from practice are sufficient.

Having so many different problems in this benchmark has one drawback, the time it takes to run all algorithms. After having the results shown in table 9.1 some methods were removed from these tests, so in the end only the first few methods are tested to determine if they show the same quality and Floor Ceiling is included because of its low time of execution and good baseline opportunities.

Tested Method	Average Time (s)	Average Coverage (%)
Genetic Algorithm (one nest at a time)	89.85138	84.47314
Genetic Algorithm (all nests at once)	79.85138	77.01155
SMT Hybrid (all nests at once)	113.09587	65.52693
Backtracking	202.16912	71.34855
Floor Ceiling	0.03794	76.20065
Using the best of all methods	-	85.03442
Final combination of methods	54.56384	84.87922

Table 9.3: Final results, based on benchmark tests of 300 random problem instances

9.4 Basic SMT compared to Genetic Algorithm

In order to test the efficiency of the Genetic Algorithm and its ability to find close to optimal solutions it is compared with the Basic SMT method.

As stated before, the Basic SMT method finds close to optimal, if not optimal, solutions for problems with sufficiently small number of input items. This number can go up to 30 but tests demonstrate that after 10 items the chances of this method experiencing timeouts for some of the calls to the SMT solver, increase gradually. Therefore, only test cases with up to 10 items are used in the following test.

Similarly to the benchmark with random inputs only (section 9.3), to get more accurate results an input set of 300 test cases is generated and the results, using both methods, are recorded. The general statistics from the test can be seen in table 9.4, where a the difference in coverage is calculated by subtracting GA coverage from SMT coverage.

From all 300 tests, in 57 cases SMT gives better coverage, while in the rest 243 cases they both give the same result. The same test was done three times, to reduce the effects of the randomness in the Genetic Algorithm, and the results were consistent each time with small variances. The Basic SMT method provides better coverage on average and even though it is by less than half a percent, this advantage was apparent in all three test runs.

It is important to note that the Basic SMT method not only outperforms GA for cases with under 10 items in terms of average coverage, it also does it around 6 times faster.

As a result, when given an input of up to 10 items that have a combined area less than the frame's area, it is always beneficial to first try that method and if any of the SMT calls timed-out, then use GA and return the best result of both. Even though in the majority of cases such a timeout does not cause a decrease in coverage, it is a sign that it could have happened. Furthermore the arrangement found by SMT can be added to the starting population of the GA so that it accelerates the evolution in it.

The extreme differences where GA provides a much worse solution than the SMT can be used to find issues with the fitness function that is responsible for arranging the items in the nest. During the first run of this test such a defect was found in one instance and fixing it gave a global improvement in efficiency of the Genetic Algorithm. On the other hand a large difference, like the one seen in table 9.4, does not necessarily mean there is something wrong in the algorithm. This exact case is shown in figure 9.2. An initial check shows that the SMT nest cannot be formed by any chromosome in the Genetic algorithm, but if the nest is mirrored it becomes possible, thus this relatively low coverage is caused by the algorithm missing the best arrangement. Further evidence comes, after a few runs of GA over that problem instance because it eventually finds that optimal nest.

№ items	GA coverage (%)	GA time (s)	SMT coverage (%)	SMT time (s)		Difference in coverage(%)
2	82.1014	0.0023	82.1014	0.0758		0
...	...	...	...	...		
7	79.1595	6.4999	80.0157	0.1378		0.8562
7	86.2291	7.0046	86.2291	0.4017		0
7	88.5801	4.4595	88.5801	0.1475		0
...	...	...	...	...		
9	81.4435	12.0537	88.3563	3.8024		6.9128
10	90.3779	16.2836	90.3779	4.4510		0
10	87.8439	17.7451	87.8439	2.4552		0
...	...	...	...	...		
Averages:	**78.6867**	**6.4261**	**79.0793**	**1.1049**	**average**	**0.3925**
					maximum	**6.9128**

Table 9.4: Statistics on the 300 test cases with a few examples included.

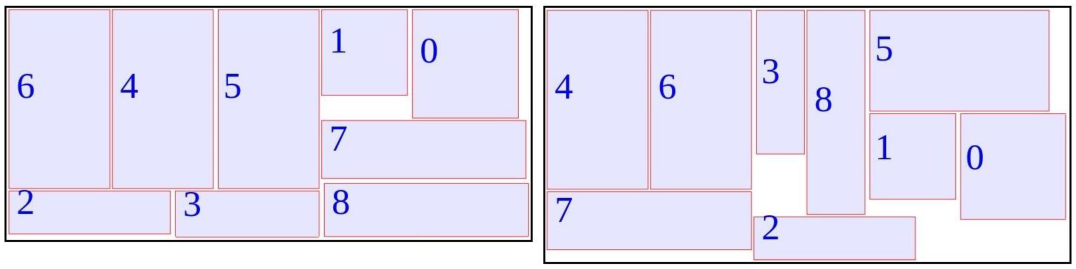

Figure 9.2: Nests created by Basic SMT (left) and Genetic Algorithm (right) for the case with highest difference in coverage shown in table 9.4.

9.5 Population for GA

One aspect of the Genetic Algorithm that contributes to its efficiency is the size of the population. Table 9.5 shows the results of a test in which the only variable is the population size. The Genetic Algorithm was run over 80 cases from practice that was used as a benchmark at that moment, but because the differences turned out to be comparable to the fluctuations of GA's results, each test was conducted 4 times and an average was used for the comparison. One obvious change is the increase of time, with increase of population size, which is acceptable as long as it provides an increase of resulting coverage.

Having a larger population with the same number of generations passed means that more possible chromosomes are tested which gives a higher chance for a better solution. This is apparent in the results too but at one point that benefit is not large enough to compensate for the increase of run time.

Finally, the population size of 100 is chosen, because around that point the coverage starts to increase with lower rates rapidly.

The rare cases when an input problem has less than 4 items, all possible chromosomes are less than 100 which breaks the rule that all chromosomes in the population need to be unique. Having n as the number of input items, the actual population size is based on formula 7.3 as follows:

```
population size = min($n! * 2^n$, 100)
```

Population Size	**Time (s)**	**Coverage (%)**
20	9.819	82.630
30	12.264	83.011
40	12.471	83.341
50	13.942	83.589
60	15.211	83.763
80	16.337	83.924
100	**18.160**	**84.097**
140	20.775	84.110
200	23.160	83.168

Table 9.5: Results from having different population size. Each result here is an average of 4 runs on 80 cases from practice.

9.6 Crossover functions for GA

After the chromosome structure and the fitness function for the Genetic Algorithm (section 7) were finalized, a series of tests were done to determine which crossover technique works best with them.

The tests include two with different techniques and one without using crossover at all, relying only on mutation for the evolution. These tests were done before the final benchmarking thus used different input cases and the results differ from the one used in the final comparison.

All test results with the averages can be seen in table 9.6. Not using any crossover technique yields the worst results, while having the crossover use two points for slicing the chromosome, gives the best performance. This proves that the reasoning in section 7.4, that explains the different ways of creating new generations, is correct.

	Test №	Average Coverage (%)	Time (s)
Using two crossover points	1	72.34697	86.04934
	2	72.04435	85.57286
	3	72.44652	85.52286
	4	73.14536	32.63143
	5	71.92474	85.9099
	6	73.58826	33.34453
	Average	**72.5827**	**68.1718**
Using one crossover point	1	71.42859	91.57120
	2	69.98009	142.22980
	3	70.41646	141.9696
	4	72.16561	90.41039
	5	71.99898	90.44193
	6	71.79684	90.40818
	Average	**71.29775**	**107.83857**
No crossover technique	1	69.32332	28.16234
	2	68.90096	81.73140
	3	68.69671	81.26450
	4	69.61067	27.94164
	5	68.28536	81.00666
	6	69.37543	27.77267
	Average	**69.03207**	**54.64655**

Table 9.6: Comparison between different crossover techniques for the Genetic Algorithm. Each test is done over 80 cases from practice and the averages are presented.

Chapter 10

Combining Methods

After obtaining the results for the separate methods, shown in the previous chapter, a few general patterns are observed:

- For inputs with less than 15 items Basic SMT has the highest chance to find the best solution.
- For inputs with less than 50 items both variations of the Genetic Algorithm are the preferred methods.
- For higher number of items it is the Genetic Algorithm (one nest at a time).
- If Basic SMT is unable to produce a nest then the images cannot fit in one frame and another method must be used.
- Floor Ceiling is quick enough, that it can be used in every case before the other methods.

These ideas are implemented in the final application for the company and the exact way it is done is shown as a flow chart in figure 10.1. One thing that is not shown directly in it is how the time is shared among the methods. At the start of it the users, in this case the workers, specify how much time they give the application to find a solution. Let's call that time *work_time*.

Previous tests show that Basic SMT executes fairly quickly, thus it is given only a part of the *work_time*. The exact amount of time it would need is heavily dependent on the type of inputs the software is expecting to work with and how much time do workers usually allow it to work. Tests show that the maximum time the Basic SMT executes for was 7.942 seconds while the time given is most often 40 seconds, so allowing this method to work for 20% of *work_time* must be enough.

Floor Ceiling executes for part of a second and is done first, or after the 20% of *work_time*, thus it does not need a time constraint.

If Genetic Algorithm (all nests at once) needs to be used, it is given half of the remaining time. Depending on weather there has already been a Basic SMT execution that failed or not, that time is either 40% or 50% of *work_time*, respectively. Again, only a part of the time is used as an execution of the Genetic Algorithm follows and it needs to execute too.

The time given to the last method is not calculated as a percentage of *work_time*, but instead a calculation, of what time is remaining, is used.

All these percentages above can be adjusted to find the best combination but testing showed that using values that are even +/- 10% than those show less change in performance than the variance caused by Genetic Algorithm's randomness.

In the flow chart (figure 10.1) there are two parameters, namely *smt_threshold* and *GA_threshold*. To fit the specific needs of the company they were chosen as 15 and 50, respectively. Different conditions might require these parameters to change.

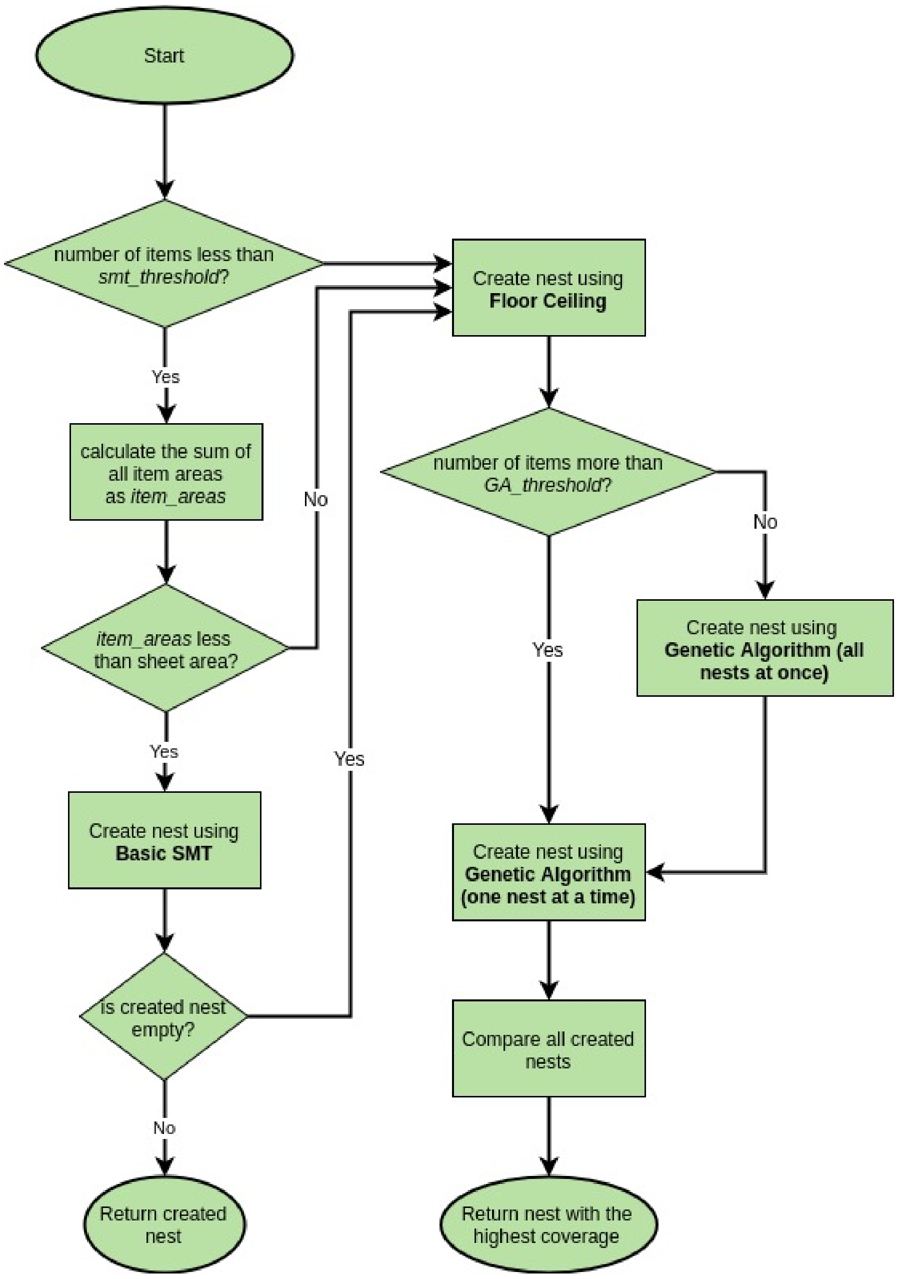
Start
number of items less than *smt_threshold*?
Yes
calculate the sum of all item areas as *item_areas*
item_areas less than sheet area?
Yes
Create nest using **Basic SMT**
is created nest empty?
No
Return created nest
No
Yes
Create nest using **Floor Ceiling**
number of items more than *GA_threshold*?
No
Create nest using **Genetic Algorithm (all nests at once)**
Yes
Create nest using **Genetic Algorithm (one nest at a time)**
Compare all created nests
Return nest with the highest coverage

Figure 10.1: Flowchart showing when the different methods are used for the final application.

The results of this method can be seen in table 9.3 and 9.1 where it is referred to as **Final combination of methods**. Its performance is comparable to the what could theoretically be achieved if all methods were run parallel and the best results were used. The reason why that maximum isn't reached is that only four of the methods are included in this final one.

Chapter 11

Conclusions

The goal of the project is to find a method that gives the best coverage to the rectangle packing problems in a limited time. To properly compare the effectiveness of different methods, they need to be implemented and tested with the same problem instances.

In addition to some known approaches, a new one that makes use of an SMT solver is introduced. It resulted in three different methods, of which one is being used in the final application.

Conclusions for each method are made separately:

- Floor Ceiling algorithm (FC) - An intuitive direct algorithm of arranging the items. Not optimal but has the benefit of executing much quicker than any other tested method.
- Basic SMT approach - If it finds a solution it is either optimal or close to optimal but it succeeds only with problems that result in a single nest solution and the input items are less than 25.
- Best coverage SMT approach - A failed experiment. Even in its final form, this method is too slow to provide satisfactory results.
- SMT Hybrid approach - It can solve all given problems with good achieved coverage. With high number of input items its execution time becomes long but still within limits
- Backtracking algorithm - This method also gives good results but they quickly deteriorate with an increase in the number of input items. It is an improvement over a direct greedy approach.
- Genetic Algorithm (GA) - The method that gave best results if used solely. Benefiting from a non-optimal but quick fitness function that allows for quicker testing of the population.
- Particle swarm optimization (PSO) - Another experiment that yielded unsatisfactory results, especially to more complex input problems.

The results proved that depending on the input, different methods give the best results and in order to build a general solver, a tool was designed to utilizes some of them. The methods that are included in that tool are Basic SMT, FC and GA, while the others are absent either because of their long execution time or because their results are unsatisfactory.

The research process during the thesis was going in parallel with the development of the tool used in BigImpact [3] that integrated the researched methods. It proved to be useful after the introduction of the SMT hybrid approach and continued to improved with the development of the rest of the methods.

A test, comparing this tool against the method previously used in the company, shows an average increase in material coverage of 2.76%. In 2016, the company used 770 000 m^2 of material to print on, with an average cost of 2.5 euros per m^2. Assuming the average coverage was 82.94%, suggested from the test in section 9.2, the increase of coverage to 85.7% means that only 745 200 m^2 would have been sufficient. That translates to saving **24 800** m^2 of material which would cost roughly **62 000 euros**.

Future work

The implemented methods are not the only ones solving the problem, hence a different set of methods may be better used in the final application.

There are also more options of using the SMT solver with other algorithms that could not be included in this research. It is possible that a better one exists.

Even the method that yielded best results, the Genetic Algorithm, has unexplored varieties that may lead to it having a better performance.

Bibliography

[1] D. Vigo A. Lodi, S. Martello. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing 11 (4) (1999) 345-357*, 1999. 1, 2

[2] J. O. Berkey and P. Y. Wang. Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society*, 38(5):423–429, May 1987. 2

[3] BigImpact. Large scale printing company. `http://www.bigimpact.com/`. 49

[4] Bernard Chazelle. The bottom-left bin-packing -heuristic: An efficient implementation. *IEEE Transactions on computers vol c-32. no. 8 August*, 1983. 2

[5] D.S. Johnson F.K.R. Chung, M.R. Garey. On packing two-dimensional bins. *SIAM J. Algebraic Discrete Methods, 3:66–76*, 1982. 2

[6] Peng Yuan Haiming Liu, Jiong Zhou Xinsheng Wu. Optimization algorithm for rectangle packing problem based on varied-factor genetic algorithm and lowest front-line strategy. *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014. 28

[7] Troy Harrison. 1D Bin Packing Problem python library. `https://github.com/ibigpapa/bin_packing_problem/`, 2016. 16

[8] G. Galalnbos J. B. G. Frenk. Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem. *Computing 39, 201 - 217*, 1987. 2

[9] Richard E. Korf. A new algorithm for optimal bin packing. *AAAI-02 Proceedings*, 2002. 16

[10] Wong L and Lee L.S. Heuristic placement routines for two-dimensional bin packing problem. 5, 04 2009. 2

[11] Andrea Lodi. Algorithms for two-dimensional bin packing and assignment problems. Master's thesis, UNIVERSITA DEGLI STUDI DI BOLOGNA, 19961999. 2

[12] Matthias Heizmann, Giles Reger, and Tjark Weber. SMT-COMP 2017. `http://smtcomp.sourceforge.net/2017/`, 2017. 7

[13] ARC project. Survey on two-dimensional packing. `http://cgi.csc.liv.ac.uk/~epa/surveyhtml.html`, 2006. ixix, ixix, 5, 6

[14] Microsoft Research. Z3Prover z3. `https://github.com/Z3Prover/z3`, 2017. 2

[15] Jomg-Tzong Homg Shim-Miin Hwang, Cheng-Yan Kao. On solving rectangle bin packing problems using genetic algorithms, 1994. 28

[16] Toby Walsh. *General Symmetry Breaking Constraints*, pages 650–664. 2006. 12

Appendix A

method	Maximum	Genetic Algorithm	SMT Hybrid	Backtrack	SMT Hybrid	Floor Ceiling	Particle Swarm	basic SMT
№ items			(all at once)		(one at a time)		Optimization	
2	65.0964	65.0964	65.0964	65.0964	65.0964	65.0964	65.0964	65.0964
2	91.0916	91.0916	91.0916	91.0916	91.0916	91.0916	91.0916	91.0916
2	66.2977	66.2977	66.2977	66.2977	66.2977	66.2977	66.2977	66.2977
2	85.0108	85.0108	85.0108	85.0108	85.0108	85.0108	85.0108	85.0108
2	96.1660	96.1660	96.1660	96.1660	96.1660	96.1660	96.1660	96.1660
2	42.1654	42.1654	42.1654	42.1654	42.1654	42.1654	42.1654	42.1654
3	28.8921	28.8921	28.8921	28.8921	28.8921	28.8921	28.8921	28.8921
3	96.9738	96.9738	96.9738	96.9738	96.9738	96.9738	96.9738	96.9738
3	96.9738	96.9738	96.9738	96.9738	96.9738	96.9738	96.9738	96.9738
3	64.6860	64.6860	64.6860	43.3114	64.6860	64.6860	64.6860	64.6860
3	85.8668	85.8668	85.8668	85.8668	85.8668	85.8668	85.8668	85.8668
3	89.6433	89.6433	89.6433	89.6199	89.6433	89.6433	89.2352	89.6433
4	71.2219	71.2219	63.8332	70.9894	63.8332	63.8332	63.8332	71.2219
4	89.5327	89.5327	89.5327	89.5327	89.5327	89.5327	89.5327	89.5327
4	68.4423	68.4423	68.4423	48.2234	68.4423	48.2403	48.2403	68.4423
4	91.4161	91.4161	91.4161	91.3984	91.4161	76.5638	76.5638	91.4161
4	87.8009	87.8009	79.5711	87.7301	79.5711	78.2837	49.5836	87.8009
5	82.8878	82.8878	82.8878	82.7916	82.8878	70.5899	52.2066	82.8878
5	78.2914	78.2914	74.0044	59.8342	74.0044	60.5039	59.8342	78.2914
5	55.5965	55.5965	55.5965	44.1843	55.5965	44.2127	44.2127	55.5965
5	91.9244	91.9244	91.9244	91.9244	91.9244	52.1394	91.9244	91.9244
5	81.3088	81.3088	55.6902	80.3264	55.6902	55.6902	41.8959	81.3088
6	78.3586	78.3586	78.3586	78.3586	78.3586	78.3586	78.3586	78.3586
6	78.3586	78.3586	78.3586	78.3586	78.3586	78.3586	78.3586	78.3586
6	88.7644	88.7644	88.7644	88.7279	88.7644	88.7644	54.3730	88.7644
6	94.0513	94.0513	94.0513	94.0513	94.0513	94.0513	94.0513	94.0513
6	97.4500	97.4500	97.4500	97.4500	97.4500	97.4500	97.4500	97.4500
7	75.7556	75.7556	75.7556	75.7316	75.7556	75.7556	75.7556	75.7556
7	91.7624	90.3930	65.6470	90.3670	65.6470	70.9322	42.6563	91.7624
7	80.6468	80.6468	80.6468	60.1931	80.6468	58.2742	60.1931	80.6468
7	91.5291	91.5291	91.5291	63.0092	91.5291	68.1041	68.1041	91.5291
7	91.4272	91.4272	71.8271	87.2799	71.8271	84.6261	37.6181	90.6653
8	90.0853	90.0853	90.0853	89.6604	60.9231	90.0853	44.1021	90.0853
8	93.6649	93.6649	92.3169	81.8202	92.3169	79.2778	52.9641	93.0824
8	83.5305	83.5305	83.5305	83.5305	75.2711	75.2711	54.9501	0.0000
8	93.9270	93.9270	90.4970	88.5645	72.4970	65.5689	44.1570	93.9270
9	73.8859	73.8859	71.8373	73.8859	71.8373	73.8859	46.0292	0.0000
9	90.6799	90.3248	84.0316	79.3450	76.0316	71.9967	41.3314	90.6799
9	92.7501	92.7501	87.8877	80.5126	79.8877	82.0149	45.5270	91.4734
9	90.6388	90.6388	84.3639	77.1067	72.3639	69.0027	72.3639	90.6388
9	91.5699	91.5699	87.2216	79.1853	73.2216	77.1832	44.4063	91.5699
9	84.0772	84.0772	83.1283	84.0062	83.1283	83.1283	76.2352	0.0000
10	96.1056	96.1056	96.1056	78.4588	96.1056	77.2147	60.8827	96.1056
10	85.7339	85.7339	76.6377	72.9036	76.6377	76.6377	53.9888	85.7339
12	88.3738	88.3738	88.3738	88.3738	88.3738	88.3738	88.3738	88.3738
12	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502
12	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502
12	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502	93.0502
13	93.6644	93.6644	89.1310	83.8431	93.6644	79.1475	58.4997	0.0000
13	93.6644	93.6644	89.1310	83.8431	93.6644	79.1475	58.4997	0.0000
13	85.0360	85.0360	80.7032	85.0360	80.7032	64.5454	44.9302	0.0000
13	90.4082	90.4082	87.8341	84.1580	87.8341	77.3375	51.4406	84.5053
14	91.5558	90.2926	91.5558	90.2926	85.2358	85.2358	58.7048	0.0000

Table A.1: All the coverage data from the Benchmark on production problems. Part 1

method	Maximum	Genetic Algorithm	SMT Hybrid	Backtrack	SMT Hybrid	Floor Ceiling	Particle Swarm	basic SMT
№ items			(all at once)		(one at a time)		Optimization	
14	76.8995	76.8995	71.0288	76.8995	73.1684	52.3509	50.5988	0.0000
14	91.7820	91.7820	89.3720	91.7820	89.3720	55.7675	48.1202	91.7820
15	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433
15	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433
15	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433	92.4433
18	90.8333	90.8333	90.8333	90.4611	66.8864	66.8864	60.6387	90.8333
18	92.2896	89.1870	92.2896	81.7083	92.2896	81.3683	47.4883	0.0000
18	79.1014	79.1014	79.1014	79.1014	65.4621	65.4621	61.2288	79.1014
18	93.4080	93.4080	90.3120	76.6915	76.6915	66.9108	33.1199	89.8211
20	92.4537	92.4537	92.4537	92.4537	92.4537	92.4537	92.4537	92.4537
20	53.9405	53.9405	53.9405	53.9405	53.9405	53.8462	53.9405	53.9405
21	84.9661	84.9661	77.7584	84.3571	70.1568	69.5104	46.8084	0.0000
25	93.7481	93.7481	86.7824	87.3886	86.2241	82.0996	72.0996	0.0000
25	91.4396	89.9215	88.6563	89.1455	91.4396	82.0609	48.0646	0.0000
25	66.0920	66.0920	66.0920	66.0920	66.0920	66.0920	66.0920	0.0000
25	64.9108	64.9108	64.9108	64.9108	64.9108	64.9108	43.2739	0.0000
26	77.5714	77.5714	77.5714	77.5714	77.5714	58.1786	38.7857	0.0000
29	76.4772	76.4772	74.9466	68.8749	74.9466	73.6765	52.2232	73.4996
29	86.1047	86.1047	84.4582	86.1047	81.0226	71.9929	49.4196	0.0000
30	90.9686	90.9686	84.9501	84.9501	62.1126	90.9686	63.6522	0.0000
30	90.9686	90.9686	90.9686	84.9501	62.1126	90.9686	63.6522	0.0000
30	55.2330	55.2330	55.2330	55.2330	55.2330	55.2330	55.2330	55.2330
30	86.5044	86.5044	84.7402	58.2934	74.7402	72.0588	58.2934	69.2035
31	95.7430	94.8345	89.5130	95.7430	89.4639	89.7097	60.5085	0.0000
32	83.4438	83.4438	81.0377	79.0377	76.0377	79.7930	76.0377	0.0000
32	89.0748	89.0748	85.5432	89.0748	66.7838	66.7838	60.3228	0.0000
33	93.4878	93.4878	70.4422	86.8401	70.4422	82.2242	67.3742	0.0000
33	82.5800	82.5800	82.5800	82.5800	82.5800	82.5800	82.5800	0.0000
34	93.0576	93.0576	90.8269	80.0174	81.9635	76.4920	67.0872	0.0000
35	43.7924	43.7924	43.7924	43.7924	43.7924	43.7924	43.7924	0.0000
36	89.7451	88.1200	89.7451	85.5534	89.7451	86.3693	66.9220	0.0000
40	87.7137	86.1659	84.8538	87.7137	80.9037	77.0378	48.6005	0.0000
46	76.4011	76.4011	73.7127	58.7127	62.5523	57.0420	50.9033	0.0000
49	74.6792	74.6792	72.4589	68.1651	65.2364	60.4909	51.4877	0.0000
55	96.8096	96.8096	77.8161	82.4058	76.6365	80.0567	71.4197	0.0000
61	94.4969	94.4969	89.1988	84.5551	87.1082	76.2871	50.8945	0.0000
62	83.6888	77.8377	75.8072	78.8641	58.9195	83.6888	58.9195	0.0000
65	63.1646	63.1646	63.1646	63.1646	63.1646	63.1646	42.1097	0.0000
76	98.5625	98.5625	79.2321	82.6407	78.3435	81.7614	72.1632	0.0000
81	79.7494	79.7494	79.7494	79.7494	79.7494	79.7494	39.8747	0.0000
82	95.8382	95.8382	81.6407	84.2453	67.6251	78.1493	54.4410	0.0000
87	78.2483	69.5540	68.1528	78.2483	62.5986	69.5540	52.1655	0.0000
90	90.6271	83.5009	87.4413	90.6271	89.2943	88.1542	61.0993	0.0000
95	93.8406	93.8406	88.5145	83.9707	70.8304	78.6370	56.8347	0.0000
97	95.6047	95.6047	88.2233	84.9249	60.7855	79.3457	74.2402	0.0000
99	77.3769	67.7048	75.7649	77.3769	77.3769	77.3769	54.1638	0.0000
101	89.3788	89.3788	88.7655	84.2213	68.0093	79.9929	58.3715	0.0000
101	76.3907	69.4461	74.2687	76.3907	76.3907	76.3907	50.9271	0.0000
107	89.7256	82.6301	89.6890	89.3691	87.7601	89.7256	61.8299	0.0000
107	89.5614	89.5614	86.0404	88.9010	74.3284	79.5645	59.1378	0.0000
115	91.1350	87.9508	86.8013	91.1350	76.6237	82.2062	59.5041	0.0000
118	96.9157	96.9157	90.5024	83.3588	60.5416	80.5558	74.3229	0.0000
123	91.2463	88.4717	86.8936	91.2463	77.7049	82.4247	61.3490	0.0000
124	89.7825	81.8204	89.1912	85.8769	86.2940	89.7825	63.7531	0.0000
131	91.3405	88.4594	88.1859	91.3405	78.2170	83.0856	61.7925	0.0000
139	91.4214	87.6665	90.8409	91.4214	81.1258	84.5970	63.5514	0.0000
142	95.5727	95.5727	93.8254	85.0892	60.3643	80.3437	74.1336	0.0000
145	85.7816	84.7508	83.8876	85.1671	75.8905	85.7816	63.2291	0.0000
151	89.7922	87.8902	87.2959	89.7922	77.8808	85.9014	63.0071	0.0000
158	89.2238	85.5205	88.4415	89.2238	84.5295	85.4093	63.9694	0.0000
166	90.8074	90.8074	86.0754	84.6172	62.4154	80.5807	74.1885	0.0000
186	86.7617	86.7617	83.1069	84.7641	64.8329	80.7837	74.6204	0.0000
206	89.7761	89.7761	85.8169	84.8752	66.0207	80.6277	75.6086	0.0000
226	85.3022	85.3022	81.8632	84.3409	64.6683	80.7900	75.1340	0.0000
246	85.3037	85.3037	81.0980	85.0961	60.0698	80.9260	76.1659	0.0000
266	85.3544	85.3544	82.7600	84.8895	69.7882	81.0415	75.4778	0.0000
278	85.4195	84.0920	82.7317	85.4195	69.2928	81.6499	74.8050	0.0000
averages:	85.0052839967	83.2479160697	81.8224628963	80.35910975	76.515309032	76.4821793948	63.6756329077	40.1929367481

Table A.2: All the coverage data from the Benchmark on production problems. Part 2