Final Report

Three-dimensional packing problem

Sam Sweere, André Mertens, Lukas Breuer,
Roya Shahkouei, Diego Watanabe Ruiz, Ariadna Saladrigas Pernias

Master Research Project 2

Faculty of Science and Engineering
Dpt. of Data Science and Knowledge Engineering
Maastricht University
Netherlands

Abstract

The packing problem is one of the well known combinatorial optimization problems that comes in several variants. In this work, we consider the oriented 3D packing problem for parcels and pentominoes in which a set of oriented items is provided that must be packed into a container of a given size. To tackle this problem, an algorithm based on an existing tree search approach (CLTRS) was implemented to deal with parcel problems. For the pentomino problem, a variant of the well known Algorithm X (Dancing Links Algorithm) by Donald Knuth is presented. In addition to that, a new approach (CLTRSDLX algorithm) is introduced by combining the mentioned algorithms. The combined algorithm can solve large pentomino problems in a shorter time compared to other approaches in the literature.

Contents

Chapter 1

Introduction

1.1 Context

In the first half of the first year, DKE Bachelor students carry out a semester-long project. This alternates from year to year between coloring graphs and solving packing problems. The packing problem project consists of 3 phases. In the first phase, students have to implement an algorithm that checks if a given set of pentominoes covers a rectangle of a given size. Pentominoes are Tetris pieces but of size 5 instead of size 4. In figure 1.1 all pentominoes that can be created in 2D are shown. In the second phase of the project, the students should create a user interface that allows playing Tetris with the 12 possible pentominoes. In the third phase of the project, the students should implement an algorithm, that solves a special kind of 3-dimensional packing problem. Both the parcel problem and the pentomino problem assume that the cargo container has a fixed size and an infinite number of parcels/pentominoes are available to fill the cargo container. In this research project, we focused on the parcel and pentomino problem as stated in the third phase of the Bachelor project [1][2].

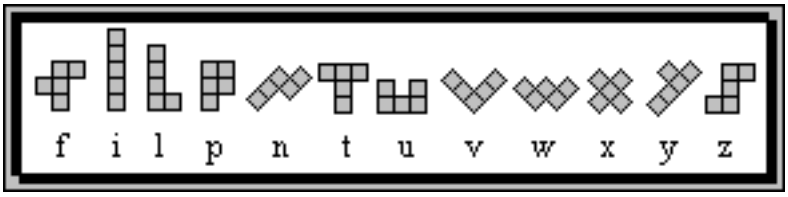


Figure 1.1: The 12 possible two-dimensional pentomino shapes one can make with 5 blocks. Adapted from [2].

1.2 Motivation

Until now, there is no possibility to automatically evaluate the algorithms developed by the bachelor students in the third phase. Therefore, several components should be developed in this Master Research Project to facilitate the evaluation. The most important deliverable in this project is a parcel and pentomino problem solver that provides a good approximation to the optimal value packaging. Contrary to the assumption for bachelor students, this problem solver should be able to solve problems with different container sizes, any subset of parcels/pentomino and a limited

number of parcels/pentominoes to fill the container. Based on this solver a test and tournament suite should be developed. These two should contain example problems that are solved by the solver and serve as test instances. These test instances should be used by students and/or examiners to see how well the students' code works and how well they perform against our benchmark (calculated value of the solver's packaging). Furthermore, the tournament suite should be able to run a (semi-)automated tournament where the code written by the students is played against each other and evaluated. This should help to evaluate the quality and efficiency of the code, but also to detect serious bugs in the implementation.

Furthermore, packing problems (knapsack problems) are known NP-hard problems that have been the subject of several investigations. Since packing problems can be specified arbitrarily, there are already different algorithmic approaches to generate good solutions for these specific problems in a short time. For example, there is already research work on the solution of polyomino packing problems. polyominoes are a superset of pentominoes. In addition to the 12 pentominoes used in this project, polyominoes also contain packings consisting of more or less than 5 pieces. Furthermore, polyominoes can occupy the complete 3-dimensional space. We consider only 2-dimensional pentominoes[3]. Starting from algorithms based on different publications for the solution of packing problems, we wanted to develop a solver that can quickly calculate an optimal or approximately optimal solution of packing and pentomino problems [1].

1.3 Literature Review

Three-dimensional packing problems have a range of different important applications particularly in logistic industries and in terms of cargo loading problems. Over the past years, there has been a large number of papers that use either optimization or heuristic algorithms to overcome design issues without a guarantee for optimal solutions. In this section, we review the literature focusing on the methodologies used by researchers to provide a brief insight into the problem of parcel packing and three-dimensional polyominoes packing.

Dutch mathematician Nicolaas Govert de Bruijn presented several approaches for packing congruent rectangular bricks of any given dimension into larger rectangular boxes, where there may be no empty spaces. De Bruijn's theorem suggests that a "harmonic brick" (one in which each side length is a multiple of the next smaller side length) can only be packed into a box whose dimensions are multiples of the brick's dimensions[4][5].

Among other optimizations, the Dancing Links algorithm was suggested by Donald Knuth. It uses a technique for reverting the operation of deleting a node from a circular doubly linked list. It is particularly useful for efficiently implementing backtracking algorithms, such as Algorithm X for the exact cover problem[5]. The Binary tree search method from Liu Shang [6] provides solutions for packing problems through applying guillotine constraints[7]. This means that a packing can be separated into its elements by performing a series of cuts parallel to the container boundaries while each cut separates the packing into two smaller sub-parts. How-

ever, due to the shape of the pentominoes, it is unlikely that a packing consisting of those fulfills this constraint. Several other heuristic approaches have been suggested such as the Most Constrained Hole (MCH) Algorithm where the algorithm simply chooses the hole that has the fewest number of fit possibilities as the next fill target[8]. Wissam F. Maarouf, Aziz M. Barbar, Michel J. Owayjan [9] proposed a heuristic algorithm called Peak Filling Slice Push (PFSP). This approach benefits from the slicing mechanism.

Up to now, the vast majority of the mentioned algorithms support the packing constraints such as support constraints and orientation constraints as well as other considerations to utilize the volume and optimize the packing. However, to use these algorithms we might have to come up with a way to either transform the problem or the algorithm such that they provide an achievable volume utilization for our problem.

1.4 Social impact

Our project is an analysis to verify which packing algorithm is the most reliable. It can be used as a guide to select the best algorithm depending on the environment or situation. From a pedagogical point of view, our project can help education staff when they have to evaluate a large number of student submissions. This allows them to save a lot of time and focus on other tasks such as investigation.

One of the key concerns for industries and governments is to find solutions for sustainable freight transportation, for example to reduce air-pollutant and Co_2 emission level in the logistic sector or simply save costs. Over the past years, many technologies have been developed to quantify and solve issues in logistics operations. Considering this, finding a solution that helps to minimize the space consumption in shipping containers, can help logistics companies to not only save packing costs but also reduce the quantities of various transportation modes. This can result in lower transportation and fewer emissions as a consequence.

Over the past decades, the world population is growing rapidly, which requires more surface areas, housing, and infrastructure. Knowing that soon we will run out of spaces to live, many construction companies and architectures are seeking to find solutions to enhance the mass-production of houses through optimizing the use of the surface area. Finding an algorithm that can optimize 3D packing problems can provide space-saving techniques for architectures to come up with creative planning processes and new design approaches.

1.5 Research questions

The research questions for our project are:

- Can we implement an algorithm that is able to calculate an optimal solution of packaging for parcel/pentomino problem for a fixed container in a limited time?
- Can we implement an algorithm that verifies if a packing is valid?
- Can we develop a generalized notation/format for parcel/pentomino problem that is as easy to understand by a human and easy to implement in code such that first year Bachelor students able to implement this?
- Can we create a (semi-) automated tournament suite that ranks the Bachelor students solvers by comparing the packings of their solvers with the packings of our developed solver for defined packing problems?

Chapter 2

Methodology

In the following, we will introduce our approach to implement an algorithm for the parcel and pentomino problem.

2.1 Parcel problem

We have been given the task to implement an algorithm that can find a solution for a parcel problem with a fixed container size. For years, researchers are trying to come up with the best solution for this problem, however, some results are more promising in comparison with others.

For the parcel problem (also known as the container loading problem) we decided to take the state-of-the-art container loading by tree search (CLTRS) algorithm [10] as a starting point. We implemented the algorithm almost exactly as described in the paper. In this section, we will give a short overview of how it works on a high level and will explain the specific modifications we applied to optimize it for our problem.

2.1.1 General Idea

Using tree search to solve the parcel problem sounds like a quite straight-forward method to pack a container. However, since there are so many ways one block can be placed inside a container the search space rapidly becomes too large to feasibly traverse it. Therefore we use some additional heuristic methods to reduce this search space. The paper *A Tree Search Algorithm for Solving the Container Loading Problem* [10] proposes a number of methods to do this. Their whole algorithm, called CLTRS (container loading by tree search) is able to achieve state-of-the-art results, outperforming all previous work. These heuristic methods consist of:

- Block Building (section 2.1.2)
- Residual Space Ordering (section 2.1.3)
- Basic Search Phase (section 2.1.6)
- Partition-Controlled Tree search (section 2.1.8)

2.1.2 Block Building

In order to fill a container, multiple parcels have to be placed. The set of different parcels is usually limited, especially in our problem. It is likely that for a good packing, a big part consists out of the same blocks stacked on top of each other. Instead of needing a big search to place all these parcels in a structured way, we could also use a simple algorithm to combine multiple parcels to create blocks. These blocks can then be used in the search. Since these blocks are larger than the individual parcels, the search space is reduced. We use a slightly different approach from the original CLTRS paper in the way we generate blocks. The block generation algorithm is:

- Take the initial parcels and generate rotated copies. Make sure to not duplicate parcels.
- Stack the starting and rotated parcels with themselves using all possible stacking orders. I.e. (1,1,1), (1,1,2), (1,2,1), (2,1,1), (1,2,2), (2,1,2), (2,2,1), (2,2,2), etc. Continue this process until we hit a user-defined limit or the generated blocks do not fit in the container anymore. Some examples are shown in figure 2.1.

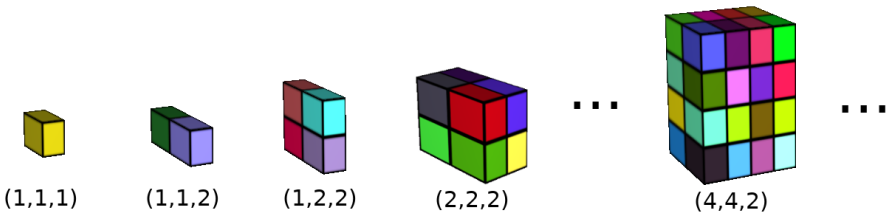


Figure 2.1: Blocks.

2.1.3 Residual Space

In order to limit the placement options for the parcels, we always place a parcel in the corner closest to the origin. After a block is placed, we usually have residual (empty) space left. We subdivide this residual space into new cuboid spaces, see figure 2.2. These residual spaces are then added to a residual space stack, which is used to track the residual spaces that still have to be packed. These residual spaces can then be considered as new containers, enabling a recursive way to solve the problem. There are, however, multiple ways to subdivide the residual space into cuboid spaces. Therefore, a heuristic algorithm is used. This algorithm is described in detail in [11]. The main idea is that we do not want to create two big residual spaces and one small residual space that cannot be filled anymore. Instead, we want to get residual spaces whose size is more evenly distributed so that they can be packed better. When the new residual spaces have been determined we can order them based on volume (minimal, medium and maximal, as seen in figure 2.2). We then put the residual spaces in order of minimal space, medium space and maximal space onto the stack. This way, if we find that the maximum or medium space cannot be packed

anymore, we can transfer some space to the minimal residual space where it might now be possible to place another parcel.

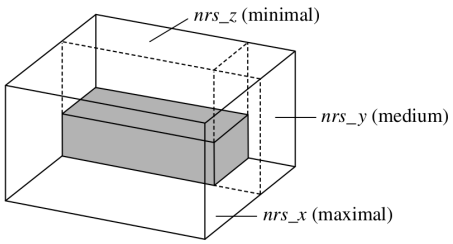


Figure 2.2: Residual space. Figure adapted from [10].

2.1.4 Placing A Block

We consider a certain (incomplete) packing with its corresponding residual space stack as a *state*. From this state, we pop the top residual space from the residual stack. We then choose a block to place inside it. This process is explained in section 2.1.6 and 2.1.8. The block is always placed in the corner closest to the origin. Once a block is placed, new residual spaces are calculated and added to the stack as previously described. This process is repeated until the residual space stack is empty, at which point no more blocks can be placed and we have a completed solution.

2.1.5 Chaining Searches

We want to use a search tree to find good block placements. However, it is usually infeasible to do a search from the empty starting container until a complete solution. The tree width can be very big, depending on the amount of blocks we have and very deep. Therefore, this is not a feasible way of solving this problem. To tackle this, we chain search phases. We first do a search up to a certain depth, get the best resulting (incomplete) state and start another search using that state as a root. We repeat this process until we have a completed solution. In order to get the best possible solution given a time frame, we start with a low *search effort*. This *search effort* determines how wide and deep we search until we select the best state to continue from. Every time a chained search is completed, we double this *search effort*, resulting in a search that takes approximately double the computation time. During all the searches we track the best-completed packing encountered thus far. This way we can stop the algorithm at any time and return the best-found solution.

2.1.6 Basic Search Phase

A basic search phase searches for the optimal block to place in a residual space using a tree search. The tree search works by:

- First get the top residual space from the stack.
- Get all the possible blocks that fit into this residual space.
- Sort these blocks in order of value density, which is the total value of the block divided by its volume. This way, we place the most valuable blocks first. If two blocks have the same value density we take the block with the biggest volume first. This is different from the original algorithm and specific for our problem.
- Take the first ns blocks from this list and place them as described previously. Save the resulting states to continue the tree search based on them.
- Continue this search until the desired depth is reached.

At this point, we have a lot of states with different packings. Due to usually being unable to continue the tree search until the whole container is packed, it often results in a lot of partially packed states. We cannot yet determine which of these incomplete states are better and should be picked to do further searches on. To solve this problem, we do a fast greedy completion on all these states and see which one gets the best greedy solution. This greedy solution can then be used to indicate how good the chosen block was. An example of a basic search can be seen in figure 2.3 at partition $\prod_1$ (upper left corner). In this example we see a basic search with a branching factor ns of 2 and a depth d of 3.

2.1.7 Greedy Completion

Greedy completion works by doing the same steps as described in the basic search. But in this case, we only calculate for one successor, i.e. we always place the block with the biggest value density and volume. We do this until the container is packed and we have a completed solution. Since this search has a branching factor of 1 it is really fast.

2.1.8 Partition-Controlled Search Phase

We can use basic searches to find a solution. However, it usually is unclear if it is better to search deeper or to search wider. In order to do a good search but without having to search both very deep and wide, which is computationally very expensive, we use a partition-controlled search. This works by chaining multiple basic search phases based on the *search effort* with all kinds of different partitions. An example of a few partition-controlled searches is shown in figure 2.3.

One such partition could be to do a basic search with a branching factor of 2 for depth 2, take the best-resulting state and do another basic search from this state but now with a higher branching factor of 8 and lower depth of 1, see figure 2.3 at partition $\prod_2$ (top right). Using partition-controlled searches allows us to search through

the most promising part of the search tree without having to compute it completely. For example, if we would want to search through a search tree of depth 3 and branching factor 8 we would have $8^3 = 512$ leaf states. On all of them, a greedy completion would have to be run to calculate complete solutions. Using a partition controlled search we only generate 56 complete solutions, this is 11% of the full search tree. In the end, we pick the state which has the best corresponding completed solution to continue the chained search on.

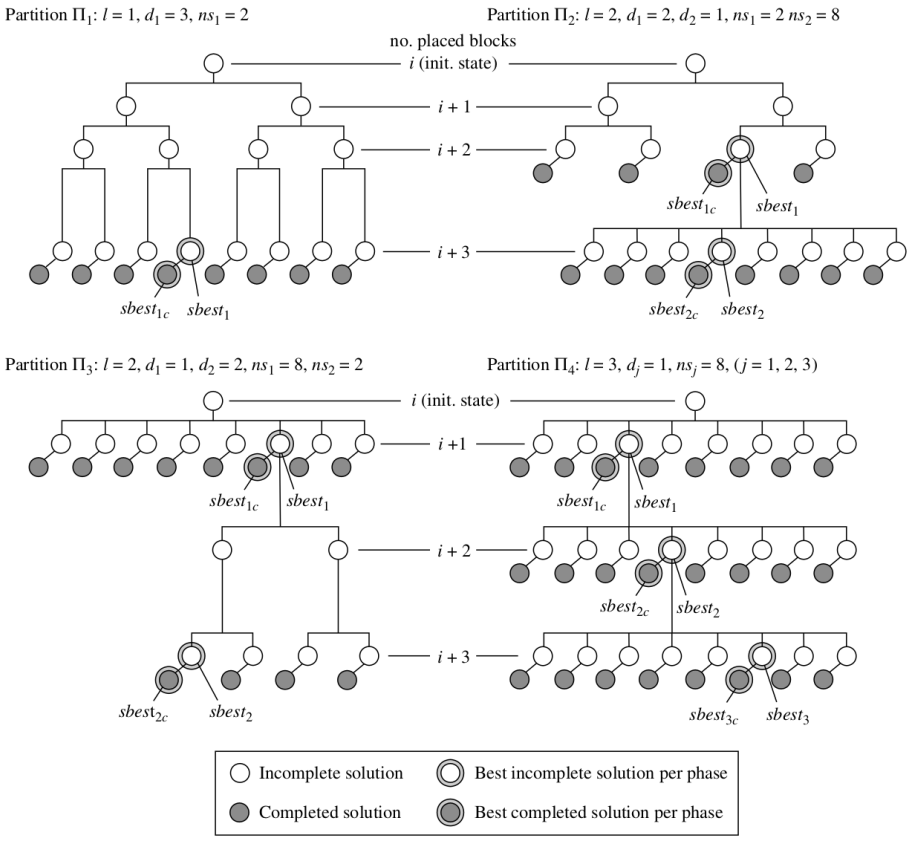


Figure 2.3: Partition controlled search tree. Figure adapted from [10].

2.2 Pentomino problem

Since the pentomino problem is very specific, there are no publications for this specific problem (2d pentominoes in a 3d container) yet. Also, the previously described CLTRS algorithm can only handle parcels in the shape of rectangular boxes, so a different algorithm had to be considered for the pentomino problem. Since the number of possible positions, rotations and shapes of the pentominoes within a container can become very large, a brute force approach looping through all the different options is not feasible for problems of a larger scale. Therefore, prototypical implementations of different algorithmic approaches were compared to decide which one to focus on and improve further. The two important quality measures of an algorithm are the total value of a packing and the time that the algorithm needs to provide high-value solutions. Initially, three different algorithms were implemented. The algorithm proposed by De Bruijn[12] and a simple greedy algorithm provided lower packing values for the same problems than our adapted version of Algorithm X by Donald Knuth[5], called DLX. The problem with De Bruijn is that it is hard to recognize a position that is impossible to fill. This weakness gets especially worse in our case due to the container is in 3 dimensions and we got larger pieces. Because of this, we decided to focus on improving the DLX instead of investigating the other algorithms much further. We also decided to implement an algorithm composed of the DLX and the aforementioned CLTRS algorithm to address the performance problems with large container sizes.

2.2.1 Pentomino Solving DLX

As previously indicated, this algorithm is based on Algorithm X, which makes use of the dancing links technique originally described by Donald Knuth[5]. This algorithm can solve exact cover problems with a backtracking approach. In an exact cover problem, you have a set of elements X and a certain number of subsets of X. A solution for this problem is a collection of those subsets in which each element of X is contained ("covered") exactly once. Such a problem can be described by a matrix consisting of 0s and 1s as in table 2.1. A solution would then be a subset of rows so that the resulting submatrix contains exactly one single 1 in each column.

Knowing this, Knuth's Algorithm X can be applied to our use case if we can convert a pentomino problem to an exact cover one. This is done by creating a matrix where each column represents one position in the provided container. For every possible placement of a pentomino, a row is created where all values are zero except for five ones. The columns of those one-values then indicate the spaces one pentomino uses within a container. Thus, assuming a pentomino consists of five 1x1x1 blocks, a container of the shape 5x5x5 is represented by a matrix with 125 columns. The number of rows depends on the different pentomino shapes you are allowed to place.

The algorithm consists of two different phases. In the initialization phase, the matrix is created based on the given container and pentomino pieces that you can use. The number of columns can easily be calculated because it equals the volume of the container. To create the rows, the algorithms iterates through all possible po-

	1	**2**	**3**	**4**	**5**
A	0	1	0	0	1
B	1	0	0	0	1
C	1	0	1	0	0
D	0	0	1	0	1
E	0	0	0	1	0

Table 2.1: Matrix representation of an arbitrary exact cover problem. Rows A, C and E form a solution.

sitions. Each position is used as an anchor position, and all the possible placement possibilities for all the pentominoes that include this anchor are computed. Each placement possibility is represented by five coordinates. The values in the columns associated with those coordinates are set to 1. The final step in the initialization phase is the removal of duplicate rows to increase the algorithms' performance.

The second phase of the algorithm is a recursive depth-first search that is continued until either an optimal packing is found, time runs out or all the possibilities are checked. Our variation of the algorithm comprises the following steps:

1. If the optimal solution is achieved or time ran out, return the best solution found.
2. Select the column with the least amount of 1s.
3. Remove all columns with only 0s.
4. For each row containing selected column:
    4.1. Add row to the solution.
    4.2. Remove all rows with "overlapping" 1-values.
    4.3. Remove rows contained in the selected row.
    4.4. Run algorithm recursively on a reduced matrix.

As you can see in step 2, you select the column with the least amount of 1s. In theory, you could select any arbitrary column, but choosing this approach the solution is likely to be found much earlier. This is because the algorithm basically tries to use the different shapes to fit them tightly into the existing gaps where only a few other pentominoes would fit into. This creates a dense packing overall.

The algorithm itself does not have to be changed a lot when allowing only a limited amount of pentominoes per shape, you just have to keep track of the remaining amounts for each tile. When adding a piece to the current packing, the amounts are decremented by 1. You increment them again when you backtrack during the algorithm and remove elements from the packing. As soon as a value reaches zero, all the rows corresponding to that specific pentomino can be ignored from that point onward until you backtrack to it again.

To increase the performance of the algorithm, we make use of the fact that the created matrix is sparse and contains a lot of zeros. For this reason, instead of keeping all the rows in memory all the time, it is sufficient to store the indices of the columns with a value of one. Since each row represents one pentomino placement, it always contains exactly five indices. An additional measure that is taken to reduce the size of the matrix is that, when iterating through the matrix to find the column with the lowest amount of 1s (step 2), each column where the number of ones is zero is removed from the matrix. This can be done because having not a single one in a column means that for all the possible sub solutions you will never find a pentomino that fits into this specific coordinate of the container. One example for this is a problem where only pentominoes of shape X are used which can never reach the corners of your container. It is important to remember which columns were removed though because you need to be able to add them again when the algorithm backtracks to this point.

The DLX algorithm can be used in two different ways: to find the densest packing or the highest value one. Maximizing the value is less efficient for this algorithm, especially when dealing with imbalanced problems where pentominoes of one shape have a much higher value than other ones. To heuristically increase the chances of finding better solutions in such cases, the rows of the initial matrix are sorted by the value of the corresponding pentomino in descending order. This way, if you have two pentominoes that fit into the current packing equally well, the algorithm will choose the one with a higher value first, which increases the chances of finding a high-value solution earlier. Even though this does not always find the optimal solution, it proved to work better than a random order.

2.2.2 Combination Algorithm CLTRSDLX

Since the DLX algorithm reaches its limits with larger container sizes and increasing amounts of different pentomino shapes, we decided to use the CLTRS algorithm in combination with it to be able to solve large pentomino problems in a shorter amount of time. The main idea of this combined algorithm is to first create small parcels of different sizes and try to fill this with pentominoes using DLX. We do this for all parcels dimensions that are unique under rotation until the time designated time for this phase runs out. The DLX algorithm is quite fast for smaller containers, within 30 seconds we usually reach parcel sizes of 5x5x5 with all the pentomino shapes involved. It can be that DLX cannot fill the parcels perfectly (which is sometimes unavoidable). After this phase is complete we run the CLTRS algorithm with as input the pentomino filled parcels. The CLTRS then packs the containers using these parcels as described in section 2.1. Finally, in the completed solution from the CLTRS algorithm all the parcels that are packed are transformed back to pentomino packing's. In the end, this results in a container packed with pentominoes. We call this combined method CLTRSDLX. Given a maximal run time we spend half the time on packing pentominoes into parcels using DLX and the other half on running the CLTRS algorithm.

Something we encountered when testing is that the DLX algorithm has trouble with heavily imbalanced problems. For example, if we have starting pentominoes of

shape L with value 1 and pentominoes of shape X with value 10 the DLX algorithm would still include a lot of L pentominoes in its solution while a volumetrically worse packing with only X pentominoes would result in a higher total value. To tackle this problem we sort the starting pentominoes on value. While filling the smaller parcels we first run the DLX with those pentomino shapes as input that have the highest value. Following that we include the shapes with the next highest value until all the shapes are included. For example, if we have starting pentominoes:

- Shape L with value 1
- Shape W with value 2
- Shape T with value 2
- Shape X with value 10

We would first run the DLX algorithm on a smaller parcel with only shape X, then we would add shapes W and T (since they have the same value), thus we now run the DLX with shapes X,T and W. Finally we would add the L shape and thus run the DLX with all the shapes. In this way we prioritize the packing of higher valued pentominoes, which usually results in packings with higher value densities.

2.3 Tournament suite

The tournament suite has been implemented to evaluate both pentomino and parcel problem for first-year DKE bachelor students. For this reason, problems with different difficulties were chosen to be evaluated, in order to find out how good the students' algorithms work on different kinds of problems. The detailed description of the tournament requirements and deliverable can be found in the appendix. A.1.

The tournament suite evaluates the students' algorithms with respect to achieved scores on the different assigned problems. The following formula is used to calculate students' tournament score for the submitted code:

$$total\_score = \sum_{p \in P} is\_valid(ALG(p)) \cdot p.weight \cdot \left(\frac{ALG(p).packing\_value}{p.master\_packing\_value} \right)^4$$

Where:

- P = parcel and pentomino problems
- ALG = students' algorithm
- $weight$ = the weight of a problem
- $master\_packing\_value$ = calculated packing value by our developed algorithm.

- is_valid = a function that checks the validity of the students solution by the correctness checker (see section A.3). If the students algorithm runs out of time the outcome is considered invalid. The method returns a 0 for invalid and a 1 for valid packing.

At first, the performance of the students' solver is calculated by comparing the packing value with the $p.master\_packing\_value$. This results in a relative performance to our algorithm. Next, we take the fourth squared power of this relative performance, this is such that solutions that are far from our score get penalized/rewarded more. For example, let us consider that a student gets a relative performance of 0.9 for 4 problems. His score will than be $4 \cdot 0.9^4 \approx 2.6$. Another student gets an excellent solution for three problems but fails for the fourth, he will get a score of $3 \cdot 0.99^4 + 1 \cdot 0^4 = 2.88$. This score is higher. This way we encourage good results even if the algorithm fails on some problems. The reason we take the fourth power (instead of the square or third) is that we expect that a lot of solutions will be very close to our solution. It is usually quite easy to fill most of the container, the small spaces that are left make it a difficult problem. But since these only form a small percentage of the whole container the value difference will likely be relatively small. Finally, this score is multiplied with the weight of the problem and summed to get the total score. This formula takes into account that some problems are more difficult than others and it will not penalize a students' code much if it works correctly for simpler problems, at the same time it fails with the hardest ones.

Chapter 3

Results

For all the CLTRS tests a maximum block count of 10000 was used and a minimum branching of 100. These values resulted in better results.

3.1 CLTRS performance tests

To test the average performance of the CLTRS algorithm, a container size of 33x5x8 was taken and three randomly sized parcels were added to the problem subset with a value equal to the volume of the respective parcel. A container of size 33x5x8 was chosen because we use this size as standard container size for the tournament problems (see section A.1). Using the solution computed within one minute, the proportion of the obtained packing value from the upper limit was calculated. In this case, the upper limit was calculated by the following formulas:

$$value\_density = \frac{parcel\_value}{parcel\_volume}$$

$$upper\_limit = container\_volume \cdot max\_value\_density \cdot \frac{container\_volume}{parcel\_volume}$$

This approach was repeated 50 times. In figure 3.1 the box plot diagram of the obtained values is displayed. It can be seen that the CLTRS algorithm can fill the container with three parcels of a random size to 90% of the upper limit on average. Thus, the optimal packing of a container of size 33x5x8 with three arbitrary parcels is approximated to 90% by the CLTRS algorithm.

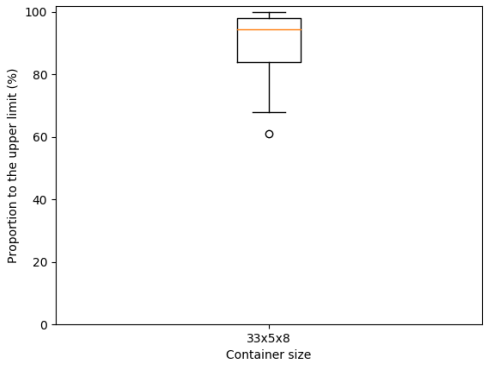


Figure 3.1: CLTRS performance tests

3.2 DLX and CLTRSDLX performance tests

To test the performance of the DLX and CLTRSDLX algorithm, pentomino problems were solved by increasing the container size linearly. We started from a cubic container with a size of 3x3x3 and increased all dimensions by 1 at each step up to a container size of 14x14x14. Since we only wanted to test the performance of the algorithms depending on the size of the container, we always used all pentomino shapes with a value of 1 in the problem subsets. So we wanted to avoid an effect of the presence or weighting of certain pentomino shapes on the performance. The time limit for the calculation of a solution was always set to 1 minute. After the solution was calculated, the proportion of the obtained packing value from the upper limit was calculated. In this case, the upper limit is the volume of the container divided by 5, because pentominoes consist of 5 atomic cubes and in the best case all coordinates of the container are filled with atomic cubes:

$$upper\_limit = \frac{containervolume}{5}$$

In figure 3.2 the proportion of the packing values to the upper limit is shown by increasing the container size. It can be seen that the DLX algorithm as well as the CLTRSDLX algorithm almost always reach 95%-100% up to a container size of 9x9x9. After that, the DLX algorithm shows a strong decrease to 0%, because the algorithm does not manage to do the internal calculations to create solutions with high packing values. The CLTRSDLX algorithm can also create solutions for larger containers that reach the upper limit with 99%-100%.

In figure 3.3 a packing created by the DLX algorithm and a packing created by the CLTRSDLX algorithm for a 9x9x9 container are shown for comparison. The pentomino arrangement of the DLX packing fills the container almost completely, but it looks very arbitrary and without any system behind it. In contrast, the packing of the CLTRSDLX algorithm is systematic. The container was divided into several smaller

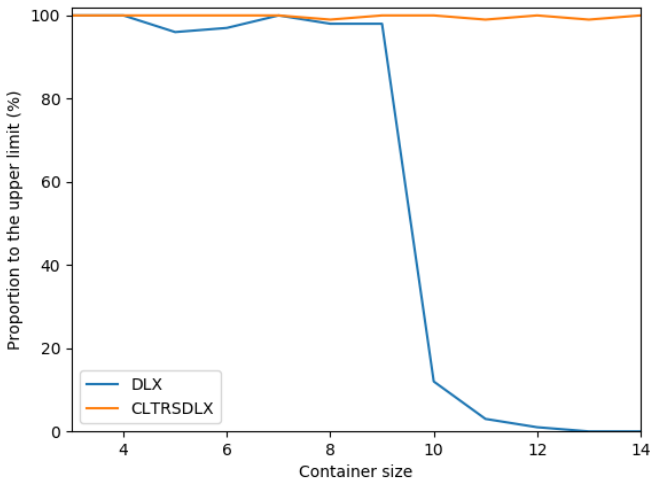
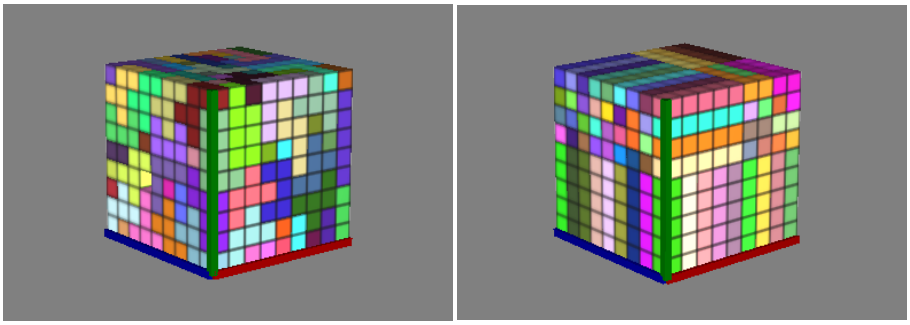


Figure 3.2: DLX and CLTRSDLX performance comaprison

sub-containers, which were then filled with pentominoes with Shape I.



(a) DLX (b) CLTRSDLX

Figure 3.3: Packing for a 9x9x9 container

Also, for a larger container with size 40x40x40, it was found out that the CLTRS-DLX algorithm uses exclusively pentominoes with shape I for the creation of solutions if all pentomino shapes with the same packing values are present in the subset of the problem (see figure 3.4a). Therefore, we wanted to test how robust the performance of the CLTRSDLX algorithm is when it no longer has its preferred pentomino shapes available. To do this, the first run of the CLTRSDLX algorithm provided all pentomino shapes with the same packing values and checked which pentominoes were used mostly for the solution. For the next run, the most used shape was removed and a new solution was calculated. This was done until only one pentomino

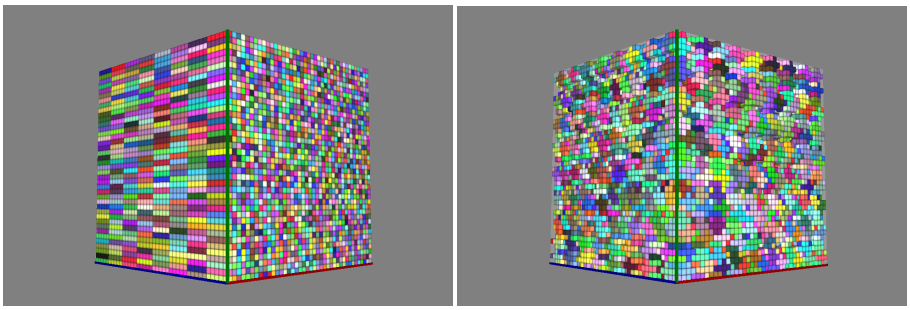
shape was left in the subset of the problem. A time limit of 1 minute was set for all runs.

Table 3.1 shows that if all shapes were present, only pentominoes with shape I were used to create the solution. But even after the exclusion of shape I only L, P and N shaped pentominoes were used to create new solutions. Only after omitting these shapes, the algorithm used combinations of different shapes. With the combinations of (T, U), (F, U) and (V, Y, F) the upper limit could still be reached to 100%. After that, the algorithm could no longer create solutions with 100%, but still 99% of the upper limit. For this purpose, combinations of (V,Z,F), (V,Z,W) and (Z,W,X) were used. Even with the last two remaining pentomino shapes Z and X the algorithm could still create a solution with 92% of the upper limit. Only when just using pentominoes with shape X only 63% of the upper limit could be reached. This is understandable though because as shown in figure 3.4b, X shapes can never be put together to fill a container completely. In contrast, pentominoes with I shape can be assembled to fill the complete container (see figure 3.4a).

Thus, we could show that the CLTRSDLX algorithm can react to the lack of certain shapes in the problem subset and search for alternative combinations of the existing shapes to reach the upper limit in the best possible way.

Solution(Shape:Amount)	**Packing value**	**Proportion to the upper limit (%)**
L:0, V:0, Z:0, W:0, Y:0, T:0, F:0, I:12800, P:0, U:0, X:0, N:0	12800	100
L:12800, V:0, Z:0, W:0, Y:0, T:0, F:0, P:0, U:0, X:0, N:0	12800	100
V:0, Z:0, W:0, Y:0, T:0, F:0, P:12800, U:0, X:0, N:0	12800	100
V:0, Z:0, W:0, Y:0, T:0, F:0, U:0, X:0, N:12800	12800	100
V:0, Z:0, W:0, Y:0, T:6400, F:0, U:6400, X:0	12800	100
V:0, Z:0, W:0, Y:0, F:3200, U:9600, X:0	12800	100
V:2560, Z:0, W:0, Y:7680, F:2560, X:0	12800	100
V:5825, Z:196, W:0, F:6688, X:0	12709	99
V:8410, Z:4278, W:18, X:0	12706	99
Z:4204, W:8507, X:28	12739	99
Z:9728, X:2048	11776	92
X:8143	8143	63

Table 3.1: CTLRSDLX packing for a container size of 40x40x40



(a) Only pentominoes with I shape (b) Only pentominoes with X shape

Figure 3.4: Packings of CLTRSDLX for a 40x40x40 container

3.3 Tournament suite

The students will be given specific conditions so their code adapts to the created tournament suite, including the input and output format. The code will be executed on a different parcel and pentomino problems. The problems differ from each other regarding parcel/pentomino subsets, values of each parcel/pentomino and container sizes. A detailed explanation of the rules and the description of the different problems can be found in the appendix section A.1. All processes from the creation of the output files by executing the code of the students, the evaluation of the outputs according to the presented formula up to the creation of the ranking have been automated by a batch script. This makes it possible to keep the tournament conditions identical for all groups and to avoid errors from manually testing the students' algorithms.

3.3.1 Performance of our algorithms on the tournament problems

The results that we obtain after running our CLTRS and CLTRSDLX algorithms on the tournament problem are shown in table 3.2 and table 3.3. We found that the DLX algorithm was too slow given the tournament time limitation.

				60 second runtime		120 seconds runtime	
Subset	**Value**	**Container**	**Upper Limit**	**Total Value**	**Upper Limit %**	**Total Value**	**Upper Limit %**
(2x2x4) (1x1x4)	5 1	33x5x8	412.5	394	**95.5**	394	95.5
(2x2x2) (2x3x5)	2 10	33x5x8	440	440	**100**	440	100
(2x2x4) (2x3x4) (3x3x3	3 4 5	33x5x8	247.5	238	**96.2**	238	96.2
(2x2x4) (2x3x2) (3x3x3)	16 24 27	33x5x8	2640	2592	**98.2**	2592	98.2
(2x2x4) (2x3x2) (3x3x3)	3 4 5	99x15x24	11880	11856	**99.8**	11856	99.8
(2x3x5) (3x3x7) (5x7x13)	7 15 115	37x23x17	3656.5	3588	**98.1**	3588	98.1
(4x2x2) (4x3x2) (3x3x3) (5x5x3)	1 2 3 4	17x11x27	561	517	**92.2**	517	92.2
(4x2x2) (4x3x2) (3x3x3) (5x5x3)	16 20 27 75	17x11x27	5049	5037	**99.8**	5037	99.8
(7x3x2) (13x7x1) (7x5x3) (17x11x5)	4 9 10 95	97x71x89	62277.6	62212	**99.9**	62212	99.9

Table 3.2: Performance of our CLTRS algorithm on the tournament parcel problems.

For the parcel problems (table 3.2), we find that the performance concerning the upper limit is never less than 92.2%. And in 6/9 problems this is even never less than 98.1%. It gets the same performance with the 60 seconds runtime compared to the 120 seconds runtime. Even in the case with bigger container sizes. This indicates that the heuristic methods of the CLTRS algorithm are sufficiently well such that a deeper search does not yield better performance. Even when the container sizes have bigger dimensions. In appendix A.5 a few visualizations of these solutions are shown.

			60 second runtime		120 seconds runtime	
Shape:Value	**Container**	**Upper Limit**	**Total Value**	**Upper Limit %**	**Total Value**	**Upper Limit %**
L:1	33x5x8	264	264	**100**	264	100
P:1	33x5x8	264	264	**100**	264	100
T:1	33x5x8	264	239	**90.5**	239	90.5
L:1, P:1, T:1	33x5x8	264	264	**100**	264	100
L:3, P:4, T:5	33x5x8	1320	1186	**89.8**	1186	89.8
L:1, P:1, T:10	33x5x8	2640	2100	**79.5**	2100	79.5
L:1, P:1, T:1	99x15x24	7128	7128	**100**	7128	100
F:1, W:1, Z:1	33x5x8	264	250	94.7	264	**100**
all shapes:1	33x5x8	264	264	**100**	264	100
F:3, I:1, L:1, P:1, N:2, T:2, U:1, V:2, W:3, X:2, Y:3, Z:3	33x5x8	792	787	**99.4**	787	99.4
F:3, I:1, L:1, P:1, N:2, T:2, U:1, V:2, W:3, X:2, Y:3, Z:3	99x15x24	21384	20588	96.3	20649	**96.6**

Table 3.3: Performance of our CLTRSDLX algorithm on the tournament pentomino problems.

When looking at the performance of our CLTRSDLX algorithm on the tournament problems (table 3.3), we see that the performance concerning the upper limit is never less than 79.5%. For 6/11 problems it is even able to find a perfect solution, this means that the value is the same as the upper limit. In contrast to the parcel problems we see that when all the pentomino types are used with different values the CLTRSDLX algorithm benefits from more computing time. In appendix A.6 a few visualizations of these solutions are shown.

Chapter 4

Discussion

Performance Difference Between our Algorithms
After running the tests, impressive results were obtained from the three algorithms. The CLTRS algorithm solves parcel problems in a short amount of time. Since the beginning, the Dancing Links algorithm (DLX) solved the pentominoes problem faster than other algorithms. During our tests, it performed as we expected. CLTRSDLX is the algorithm used to solve the pentominoes problem in big containers as can be seen in figure 3.4. It spends more time than the other two due to the need to handle more complex problems. It can be said that the bigger the container is and the more pieces it has, the more time is required to solve the problem.

Imperfect Block Generation CLTRS
The original CLTRS paper also included imperfect packing to the block generation. Since for our problems we expect that a lot of packing can be near perfect we did not include this functionality. Instead, we spend the computation time on generating bigger blocks such that the search time can be reduced.

Transposition Table CLTRS
For the CLTRS algorithm, we also implemented a transposition table using Zobrist hashing. The idea was to recognize a previously encountered state in the search and immediately return the previously found completed packing value. This would reduce the times we have to do greedy completion. However, we found that there were too many limitations of using a transposition table on our problem and it also barely increased the search speed. The implementation of the transposition table with Zobrist hashing is described in appendix A.4. The main limitations we encountered where:

- The possible states we can encounter while packing a container is so big that the transposition table has to become unfeasible large for bigger container sizes if we do not want to have too many transposition table collisions.
- The search speed did not increase significantly, signaling that the greedy completion is not the bottleneck of the CLTRS algorithm and/or we do not encounter similar states often enough during the search phase.

Because of these reasons we decided not to include the transposition table in the final CLTRS algorithm.

Partition Searches CLTRS
A lot of the partition sub searches with the same search depth do not yield better results. But sometimes it does. It might be better not to have so many partition searches on the same max search depth.

Offline Pentomino Packing CLTRSDLX
To reduce the runtime of the CLTRSDLX algorithm, the solutions for the sub problems that are currently calculated by the DLX could also be calculated in advance. Since the amount of possible pentomino shapes is quite limited and the container sizes for the sub problems do not have to be very large, this could provide strong improvements for the performance of the algorithm. One thing to keep in mind when implementing this approach is that it becomes a lot more complicated when taking imbalanced problems into account because different values can impact the resulting packings significantly. This could be addressed by precomputing solutions with all the different combinations of pentominoes and then you can select the ones that fit the distribution of the values the most.

Dynamically Providing Pentomino Shapes based on Value
As mentioned before, the DLX algorithm does not perform optimally on imbalanced pentomino problems, since the only measure that is taken to address this problem is sorting the rows of the matrix by the value of the corresponding pentominoes. While this on average seems to perform a bit better, it is just a small heuristic improvement. Within the CLTRSDLX algorithm we deal with that issue by adding the higher value pentominoes exclusively at first and only later on providing the complete set of shapes to the algorithm.

Visualizer Problems with Big Containers
Another problem is that the visualizer crashes for very big containers while our CLTRS/-CLTRSDLX solver is able to calculate them. This is probably because we load all the boxes and calculate textures for it, while we would only want to do this for the boxes we can actually see. Using a dynamic render would solve this problem.

Better Residual Space Merging
We could enhance the residual space algorithm more by when we cannot fill a minimal residual space we currently pop this from the stack and consider this space as empty in the final solution. Instead, we could save this space and add it to another list. When considering another residual space we could then check if this space could be enlarged by incorporating (a part of) one or more of the unfillable minimal spaces we saved.

More Extensive Tournament Results
With respect to the tournament, at the moment, the ranking is generated in a CSV format. However it can be improved later, using an interactive data visualization techniques. This make it easier to get an overview of the results and to compare the results of different participants.

Chapter 5

Conclusion

For the parcel problem we implemented an adapted version of the CLTRS [10] algorithm. The algorithm is able to fill containers, focusing on maximum value, given an arbitrary amount of input parcels with arbitrary values, within 60 seconds. For the pentomino problem we implemented the Dancing Links algorithm (DLX [5]) and adapted the algorithm to provide not only solutions for exact cover problems but also (imperfect) solutions that have high packing value. We found that the DLX algorithm finds good solutions for smaller container sizes. However, for bigger containers we found that the needed runtime and memory necessary to find a solution became too large. In order to solve pentomino problems with large container sizes, we propose a combination of the CLTRS and DLX algorithm, which we call CLTRSDLX. The combined algorithm generates parcels out of pentominoes using the DLX algorithm and uses those as input for the CLTRS algorithm which can deal with larger container sizes. We find that the CLTRSDLX algorithm is able to find good solutions for bigger container sizes while using a fraction of the runtime and memory that the DLX needed.

Furthermore, a packing visualizer and a correctness checker were developed within the project. These should enable Bachelor students to visualize and validate the packing computed by their algorithms. For this purpose, a format was developed that allows a parcel or pentomino problem to be written down unambiguously. Besides, the format is easy to read by humans and easy to process by simple string operations in the code. Finally, a tournament suite for the parcel and pentomino problems is proposed. This consists out of 9 different parcel problems and 11 different pentomino problems, chosen in such a way that they can test the quality and versatility of the students algorithm. Also a formula to calculate a tournament score based on the performance of the parcel and pentomino problems is proposed. We also developed code to run this tournament automatically.

We find that for the proposed parcel problems our CLTRS algorithm is able to generate a solution which is always within 92.2% of the upper limit within 60 seconds. For the proposed pentomino problems our CLTRSDLX algorithm is always within 79.5% of the upper limit within 60 seconds, and is able to get a perfect score for 6/11 problems within 120 seconds.

Bibliography

[1] S. Kelk, "Three-dimensional packing problem / pentominoes," *Master project description*, 2020.

[2] S. Kelk and P. Bonizzi, "Project 1-1: Pentominoes," *Bachelor project description*, 2019.

[3] K. Kimoto, H. Tsuji, Y. Murai, and S. Tokumasu, "A solution of three-dimensional polyomino packing problems," *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, pp. 3725–3730, 11 2007.

[4] N. G. de Bruijn, "Filling boxes with bricks," *The American Mathematical Monthly*, vol. 76, no. 1, pp. 37–40, 1969.

[5] D. Knuth, "Dancing links," *Millennial Perspectives in Computer Science*, 2000.

[6] S. Liu, W. Tan, Z. Xu, and X. Liu, "A tree search algorithm for the container loading problem," *Computers & Industrial Engineering*, vol. 75, 09 2014.

[7] R. R. Amossen and D. Pisinger, "Multi-dimensional bin packing problems with guillotine constraints," *Computers & operations research*, vol. 37, no. 11, pp. 1999–2006, 2010.

[8] M. Busche, "Solving polyomino and polycube puzzles."

[9] W. Maarouf, A. Barbar, and M. Owayjan, "A new heuristic algorithm for the 3d bin packing problem," *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*, pp. 342–345, 08 2008.

[10] T. Fanslau and A. Bortfeldt, "A tree search algorithm for solving the container loading problem," *INFORMS Journal on Computing*, vol. 22, no. 2, pp. 222–235, 2010.

[11] T. Fanslau and A. Bortfeldt, "A tree search algorithm for solving the container loading problem," 2008. Diskussionsbeitrag 426. Technical report, Fakultät für Wirtschaftswissenschaft, FernUniver- sität Hagen, Germany.

[12] N. G. de Bruijn, "Filling boxes with bricks," *The American Mathematical Monthly*, no. 76, p. 37–40, 1969.

Appendix A

Appendix

A.1 Tournament

A.1.1 Software

The students' code must be developed with Java version 11 to ensure the compatibility of our tournament environment with the students' code. Any updates or adjustments to this policy will require all components provided by us to be updated. To use our solution visualizer, Java FX (Framework for creating Java applications) must be installed additionally. Information on setting up a Java Project with JavaFX using IntelliJ IDEA can be found here: https://www.jetbrains.com/help/idea/javafx.html. The download of JavaFX is provided here: https://gluonhq.com/products/javafx/.

A.1.2 Deliverables

We expect the students to submit their source code and a .jar file of their code, which can be executed with the following command:

```
java -jar {jar file path} {input file path} {output file path}
```

To do this, the code must allow passing the path of the input and output files through console arguments. Furthermore the students' code must be able to read our input format (see section A.1.3) and write our output format (see section A.1.4). The name of the file indicates whether it is a parcel or pentomino problem for example parcel_sample_1.in or pentomino_sample_1.in.

A.1.3 Input format

In listing A.1 the input format of a parcel problem is displayed. The size of the container is specified in the first line. Since a container has an x-, y- and z-size, the line contains 3 integers separated by a comma. From line 2 on the definition of the parcel subset of the problem follows. For this purpose, 4 integers separated by a comma are noted in each line. The first 3 integers indicate the x-, y- and z-size of the parcel. The fourth integer defines the value of the parcel. For the tournament, an infinite amount of each parcel can always be assumed.

Listing A.1: Parcel input format

```
33,5,8
4,4,4,5
1,1,1,4
```

Listing A.2 shows the input format of a pentomino problem. As with the parcel problem, the container size is defined in the format "x-size, y-size,z-size" in the first line. The lines starting from line 2 contain the pentomino subset of the problem. Therefore each line consists of a pentomino shape and the value of the pentomino separated by a comma. Also for the pentomino problems, an infinite amount of parcels in the subset can be assumed.

Listing A.2: Pentomino input format

```
33,5,8
L,1
P,1
T,1
```

A.1.4 Output format

In listing A.3 the output format for a parcel problem is displayed. Each line notes the location of a parcel in the container. Therefore the first integer triple in brackets indicates the coordinate of the upper left corner of the parcel in the format "(x-coordinate,y-coordinate,z-coordinate)". The second integer triple in brackets indicates the x-, y- and z-size of the parcel. The two triplets are separated by a semicolon. With this notation, a parcel packing can be described unambiguously.

Listing A.3: Parcel output format

```
(0,0,0);(4,4,4)
(0,0,4);(4,4,4)
(0,4,0);(1,1,1)
(0,4,1);(1,1,1)
(0,4,2);(1,1,1)
(0,4,3);(1,1,1)
(0,4,4);(1,1,1)
```

Listing A.4 shows the the output format for a pentomino problem. The positions of all used pentominoes are noted per line. A line (pentomino) consists of 5 bracketed sections separated by semicolons. These sections stand for the 5 atomic cubes a pentomino consists of. Since each of these atomic cubes has an x-, y- and z-coordinate, each section consists of 3 integers separated by commas. Thus each pentomino packing can be noted clearly.

Listing A.4: Pentomino output format

```
(0,0,0);(0,1,0);(0,2,0);(0,3,0);(1,0,0)
(0,0,1);(0,1,1);(0,2,1);(0,3,1);(1,0,1)
(0,0,2);(0,1,2);(0,2,2);(0,3,2);(1,0,2)
(0,0,3);(0,1,3);(0,2,3);(0,3,3);(1,0,3)
(0,0,4);(0,1,4);(0,2,4);(0,3,4);(1,0,4)
(0,0,5);(0,1,5);(0,2,5);(0,3,5);(1,0,5)
(0,0,6);(0,1,6);(0,2,6);(0,3,6);(1,0,6)
```

A.1.5 Tournament problems

The following conditions apply to all tournament problems:

- The container size values are integers
- The standard container size is 33x5x8
- The parcel size values are integers
- The parcel and pentomino packing values are integers
- The pentominoes consist of 5 cubes of size 1x1x1
- There is an infinite amount of each parcel/pentomino of the subset

The tournament will consist out of 8 parcel (see table A.1) and 11 pentomino problems (see table A.2). The problems differ from each other on the parcel/pentomino subset, the value of each parcel/pentomino or the container size. Every problem also has an accompanying weight, this weight determines how much a problem counts in the score calculation, a higher weight indicates a more difficult problem. The algorithm will be run on a single thread, the time limit for every problem is 120 seconds.

Subset	Value	Container	Weight	Comment
(2x2x4) (1x1x4)	5 1	33x5x8	1	Perfect solution Easy to solve (trivial)
(2x2x2) (2x3x5)	2 10	33x5x8	1	Unlikely to be perfectly solvable. The bigger parcel has a higher value density, therefore the puzzle becomes how many of these to use before using the smaller ones.
(2x2x4) (2x3x4) (3x3x3)	3 4 5	33x5x8	2	Original problem (The same as in the current project description)
(2x2x4) (2x3x2) (3x3x3)	16 24 27	33x5x8	2	Original problem, but now the value density is the same for every parcel. Making the problem equivalent to trying to get maximum volume.
(2x2x4) (2x3x2) (3x3x3)	3 4 5	99x15x24	3	Original problem, but with a bigger container.
(2x3x5) (3x3x7) (5x7x13)	7 15 115	37x23x17	3	All the dimensions are prime numbers, also the container is bigger.
(4x2x2) (4x3x2) (3x3x3) (5x5x3)	1 2 3 4	17x11x27	3	Container has dimensions that do not subdivide well, perfect solution unlikely to be achievable.
(4x2x2) (4x3x2) (3x3x3) (5x5x3)	16 20 27 75	17x11x27	3	Container has dimensions that do not subdivide well, the values of the parcels are based on volume. All the value densities are the same.
(7x3x2) (13x7x1) (7x5x3) (17x11x5)	4 9 10 95	97x71x89	3	Ultimate problem, all dimensions are prime numbers, meaning that they do not subdivide well, the values of the parcels are based on volume such that bigger parcels have a higher value density, but since they do not stack nicely the problem becomes more complex.

Table A.1: Parcel problems

Shape:Value	Container	Weight	Comment
L:1	33x5x8	1	Easy to solve
P:1	33x5x8	1	Easy to solve
T:1	33x5x8	1	Easy to solve
L:1, P:1, T:1	33x5x8	2	Since the values are the same, this problem asks for the best volumetric solution.
L:3, P:4, T:5	33x5x8	2	The same as in the current project description.
L:1, P:1, T:10	33x5x8	2	Imbalanced problem, even a solution that does not fill the volume efficiently with only T's will be better than a tight packing where L and P are abundant. This will test how well the students optimize for the values of the pentominoes.
L:1, P:1, T:1	99x15x24	4	This tests how efficient the algorithm is.
F:1, W:1, Z:1	33x5x8	4	These shapes have the most variety and could therefore be considered the hardest.
all shapes:1	33x5x8	2	This tests whether the students have implemented all pentomino shapes to create solutions.
F:3, I:1, L:1, P:1, N:2, T:2, U:1, V:2, W:3, X:2, Y:3, Z:3	33x5x8	4	These values reward more complex shapes and give lower values to easy to pack shapes.
F:3, I:1, L:1, P:1, N:2, T:2, U:1, V:2, W:3, X:2, Y:3, Z:3	99x15x24	5	This can be considered the ultimate problem.

Table A.2: Pentomino problems

A.1.6 Tournament score calculation

The following formula is used to calculate students' tournament score for the submitted code:

$$total\_score = \sum_{p \in P} is\_valid(ALG(p)) \cdot p.weight \cdot \left(\frac{ALG(p).packing\_value}{p.master\_packing\_value} \right)^4$$

Where:

- P = parcel and pentomino problems
- ALG = students' algorithm
- $weight$ = the weight of a problem
- $master\_packing\_value$ = calculated packing value by our developed algorithm.
- is_valid = a function that checks the validity of the students solution by the correctness checker (see section A.3). If the students algorithm runs out of time the outcome is considered invalid. The method returns a 0 for invalid and a 1 for valid packings.

At first, the performance of the students' solver is calculated by comparing the packing value with the $p.master\_packing\_value$. This results in a relative performance to our algorithm. Next, we take the fourth squared power of this relative performance, this is such that solutions that are far from our score get penalized/rewarded more. For example, let us consider that a student gets a relative performance of 0.9 for 4 problems. His score will than be $4 \cdot 0.9^4 \approx 2.6$. Another student gets an excellent solution for three problems but fails for the fourth, he will get a score of $3 \cdot 0.99^4 + 1 \cdot 0^4 = 2.88$. This score is higher. This way we encourage good results even if the algorithm fails on some problems. The reason we take the fourth power (instead of the square or third) is that we expect that a lot of solutions will be very close to our solution. It is usually quite easy to fill most of the container, the small spaces that are left make it a difficult problem. But since these only form a small percentage of the whole container the value difference will likely be relatively small. Finally, this score is multiplied with the weight of the problem and summed to get the total score. This formula takes into account that some problems are more difficult than others and it will not penalize a students' code much if it works correctly for simpler problems same time it fails with the hardest ones.

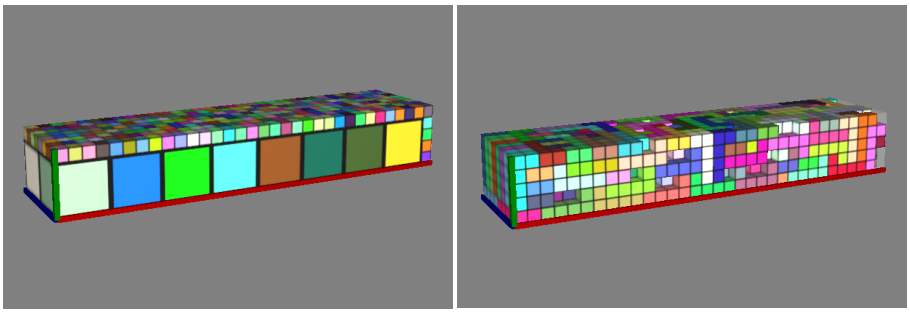
A.2 Visualizer

We provide the students with a visualizer as source code and jar file. The jar file can be executed with the following command:

fx=path to javafx-sdk-11.0.2\lib
j=Visualizer.jar
i=input file path
o=output file path

```
java --module-path {fx} --add-modules=javafx.controls -jar {j} {i} {o}
```

Therefore the input (see section A.1.3) and output files (see section A.1.4) have to be in the formats presented by us. In figure A.1 screenshots of the visualizer for a parcel (A.1a) and pentomino packing (A.1b) can be seen.



(a) Parcel packing (b) Pentomino packing

Figure A.1: Visualizer

A.3 Correctness checker

We provide students with a correctness checker that checks whether packages are valid. For this purpose the correctness checker tests:

- Are the input and output files in the correct format? (see A.1.3 and A.1.4)
- Are parcels/pentominoes placed outside the container?
- Does parcels/pentomnos overlap each other?
- Are the used parcels/pentomnoes included in the subset of the problem?

We provide the source code and a jar file of the correctness checker to the students. The jar file of the correctness checker can be executed with the following command:

```
java -jar CorrectnessChecker.jar {input file path} {output file path}
```

A.4 Transpotion Table and Zobrist Hashing

To potentially increase the search speed of the CLTRS algorithm we implemented a transition table (tt). This transposition table will be used in the greedy completion. The idea is that during the search we often use greedy completion on states which result in the same final solution. Instead of having the full greedy completion every time, we can save the final packing value for all the states we went through during the greedy completion in a transposition table (tt). For future work, we can immediately return the completed packing value for previously encountered state and had been saved in the tt. This can potentially save a lot of time by stopping greedy completion earlier.

Since we do not want to search through the tt every time that we get to a new state, we need a way to directly check if an entry for the state exists in the tt. And if so, to immediately get the saved information. To do this we reserve memory for the tt and use a hashing algorithm to get a hash key for a state. We can then use this hash key as a pointer in the tt.

For the hashing algorithm, we use Zobrist hashing. The nice property of Zobrist hashing is that when we place a new block and therefore get to a new state we only have to update the hash with this placement, instead of having to calculate the hash for every new state.

A.4.1 Transposition table implementation

In the transposition table, we store the primary hash of the state as well as the value of the greedy completion. The reason we only save the value instead of the whole state is to minimize the space needed for the transposition table.

Before starting the greedy completion we first calculate the hash of the current state. At every placement of a new block, first it is checked if this state is already in the tt. If so we get the value from the tt and stop the greedy completion. When placing a new block the hash is updated and passed on. If a collision occurs we always replace (replacement scheme new).

A.4.2 Zobrist hashing implementation

We use a random number of the Java Long type, which is 64-bits for the Zobrist hash table. This should be sufficiently large to have a low probability of collisions. A collision is when two different states have the same hash key.

We decided to use 3 bytes (24 bits) as the hash key. This is the maximum before we risk running out of memory for bigger containers.

In order to split the hash code (saved as Long) into the primary and secondary hash, we first convert it to a byte array using bit-shifting. Next, we use this byte array to take the primary and secondary hash, which are then converted back to Long's using bit-shifting. The reason we use bit-shifting is that it is done in $O(1)$ time complexity.

Generating the Zobrist random numbers

The used Zobrist random numbers are based on the current state. The state is defined by the order of how certain blocks are placed. The Zobrist hashing table, therefore, consists out of random numbers for every possible block within the maximum number of blocks that can be placed to fill the container.

A.4.3 Performance on the CLTRS algorithm

After implementing the transposition table with Zobrist hashing we found that it did not increase the performance as expected. The transportation table worked well but did not significantly improve the search speed. This could be because we might not encounter the same states often enough for the tt to be effective. We also found that for bigger containers the tt became very large and we had to reduce the number of bytes used for the hash key to 3. This resulted in more collisions. When a collision occurs we have to continue the greedy completion and cannot use the saved value. There is also always a risk of having a collision of the full hash keys, something we cannot detect and it can have big implications for the final solution. A promising state could be wrongly ignored because of the wrong value from the tt. Also, the tt increased memory usage and made the overall algorithm more complex. Because of these reasons we chose to not include the tt in the final version of the CLTRS algorithm.

A.5 Visualizations of CLTRS Solutions

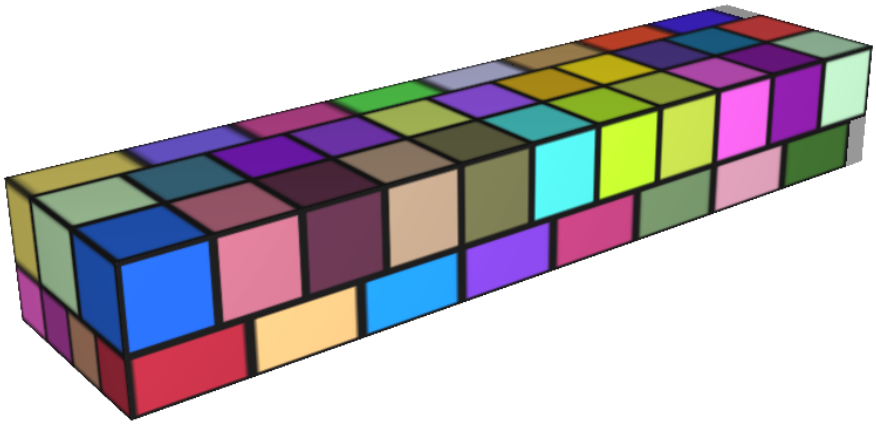


Figure A.2: CLTRS solution of a Parcel Problem with parcels (2x2x5):3, (2x3x4):4 and (3x3x3):5 with a container of size 33x5x8.

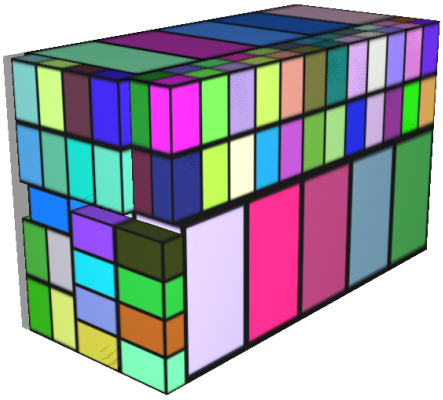


Figure A.3: CLTRS solution of a Parcel Problem with parcels (2x3x5):7, (3x3x7):15 and (5x7x13):115 with a container of size 37x23x17.

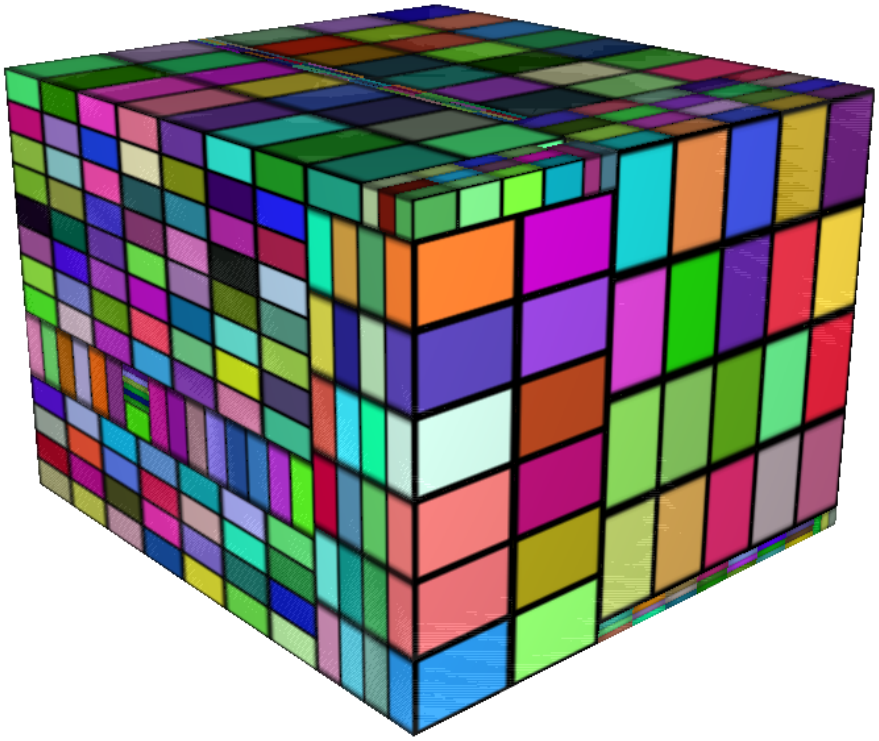


Figure A.4: CLTRS solution of a Parcel Problem with parcels (7x3x2):4, (13x7x1):9, (7x5x3):10 and (17x11x5):95 with a container of size 97x71x89.

A.6 Visualizations of CLTRSDLX Solutions

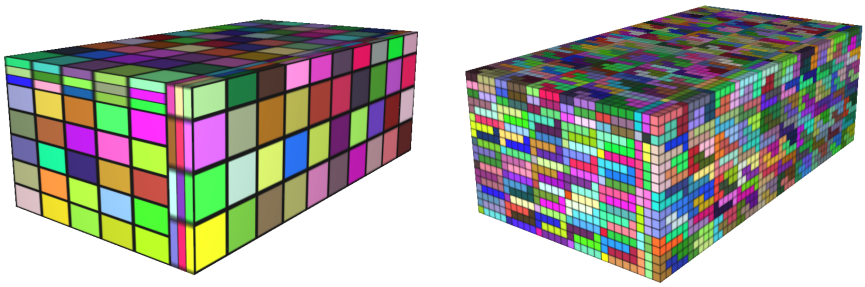


Figure A.5: Solution of the CLTRSDLX algorithm with the parcel solution (left) and the corresponding pentomino solution (right). All pentomino shapes where used with the save value with a container size of 25x9x14.

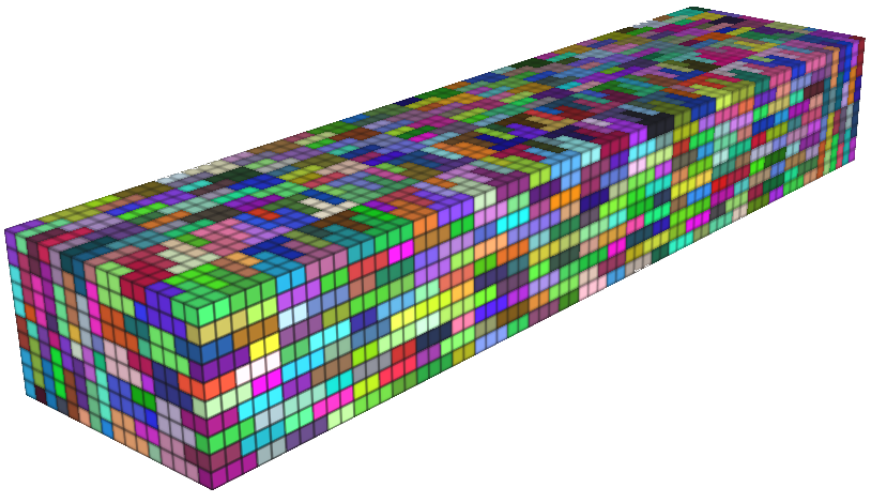


Figure A.6: CLTRSDLX solution of a Pentomino Problem with pentominoes L:3, P:4 and T:5 with a container of size 33x5x8.

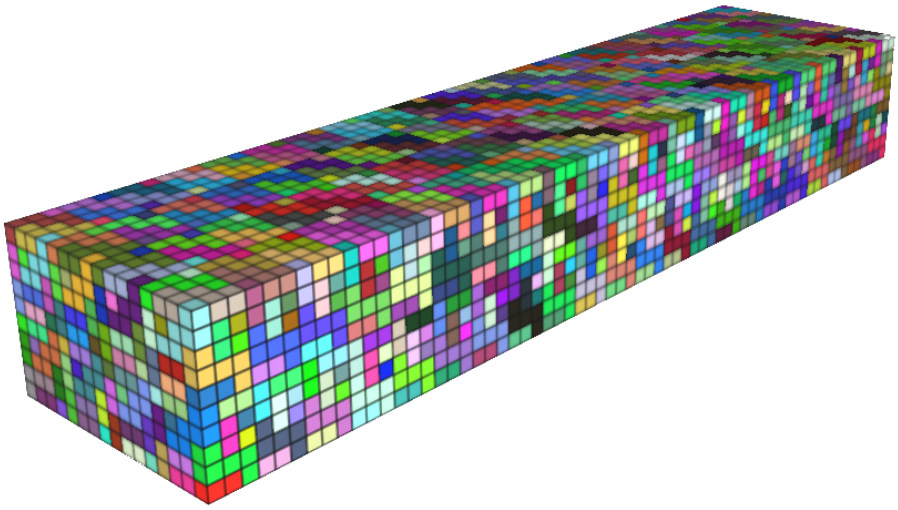


Figure A.7: CLTRSDLX solution of a Pentomino Problem with pentominoes F:1, W:1 and Z:1 with a container of size 33x5x8.

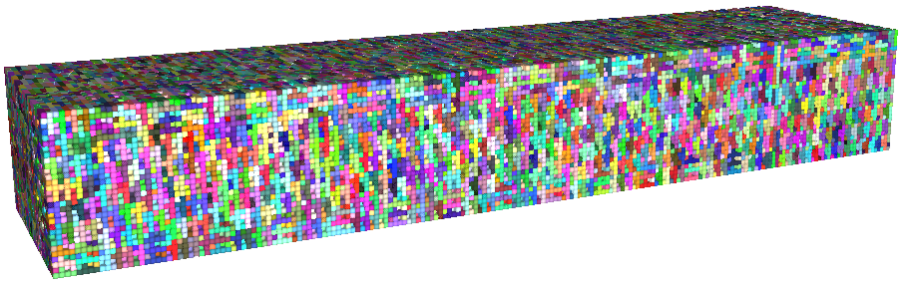


Figure A.8: CLTRSDLX solution of a Pentomino Problem with pentominoes F:3, I:1, L:1, P:1,N:2, T:2, U:1, V:2,W:3, X:2, Y:3 and Z:3 with a container of size 99x15x42.