Algorithms for the 3D bounded knapsack problem with polyominoes

Group 22

Dr. M.C. Popa

Lars Quaedvlieg, Sander op den Camp, Daniël van der Velde, Martin Aviles, Ngoc Hoang Trieu

Maastricht University

Department of Data Science and Knowledge Engineering

January 21, 2020

Word Count: 7164

Page Count: 25

1 Abstract

This paper introduces multiple different approaches to optimize a 3D bounded knapsack problem with polyominoes. This variation of the knapsack problem is particularly applicable for many companies aiming to pack and ship their products all over the world.

The problem was addressed in four different approaches. Their performances were compared with multiple constraints, such as time limit and values variation. The first approach is a greedy algorithm which prioritises the highest valued parcels first. Secondly, a genetic algorithm was implemented for this optimization problem. Finally, algorithm X with dancing links, well known to be particularly efficient with exact cover problems, and algorithm X+, which is a variation of algorithm X, adapted to make it compatible with partial covers solutions. A graphical user interface was implemented using JavaFx (a Java package for GUI programming) in order to visualize the results.

The experiments concluded that algorithm X+ performed best, obtaining an average of 97% of the theoretical maximum score on the test cases. The genetic algorithm established an average of 85% and the greedy algorithm, a score of 80%.

2 Introduction

The problem attempted to solve was a three-dimensional variation of the 'Knapsack Problem'. In computer science, a *bounded knapsack problem* is a combinatorial optimization problem where the objective is to fill a limited volume, the container, with many types of items, the parcels, having different values and shapes, while maximizing the combined value of all items in the space.

The three-dimensional bounded knapsack problem has potential in real life packaging applications. For instance, it could serve as a means of facilitating high-priority shipping, as packages with higher priority could be assigned a higher value. A new way of looking at this problem is explored, namely viewing the container as a grid, with fixed sizes for the parcels that fit into this grid, to find potentially better solutions than the ones that already exist.

There already exist algorithms (*Pisinger, David*, 2002)(*Yadav, Veenu, and Shikha Singh*)(*Kolhe, Pushkar, and Henrik Christensen*) that are able to optimize the value for rectangular boxes and non-convex shapes alike. However, to our knowledge, no algorithms are specifically optimized for three-dimensional polyominoes.

Considerations:

1. Is there a way to completely fill the container with items such that there is no empty space left?
2. When each item is given a value, what is the highest achievable value for a filled container given a set number of items?

Six different polyomino items were used. Three rectangular shapes (in meters): A (1x1x2), B (1x1.5x2), C (1.5x1.5x1.5) and three non-rectangular shapes: L,P and T (Figure 1) that are built up of 0.5x0.5x0.5 blocks.

The container size was set at 16.5x4x2.5, to represent the size of a real life container.

For the rest of this report every piece will be regarded as a polyomino built up of 0.5x0.5x0.5 cubes, which will be referenced as being 1 block (so A would be of size 2x2x4). The container will therefore be regarded as a 33x8x5 grid of these cubes.

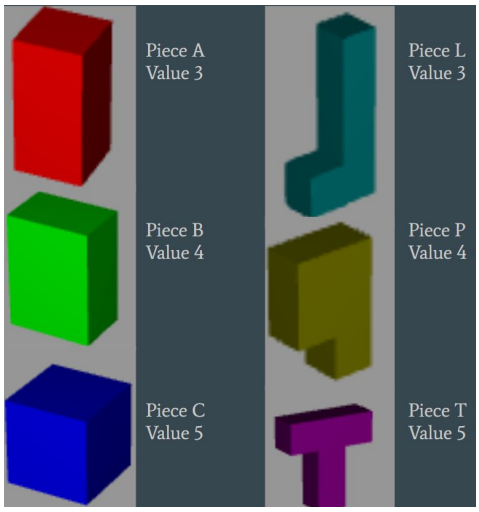


Figure 1; All pieces

In order to answer these questions with the given items and their respective sizes, four algorithms will be explored:

1. **Sticky greedy algorithm**

This algorithm was introduced due to its highly intuitive approach: attempting to fit the highest density value first, and close to each other. While this was a fairly good algorithm in itself, it also served as an excellent benchmark for the other algorithms and their performance.

2. Genetic fitting algorithm

This algorithm relies on different criteria, known as performance measure, in order to choose where to place the parcels. The relative importance of each criteria is represented with weights which can be trained with a genetic algorithm to optimize the decision making.

3. Algorithm X with Dancing links

Algorithm X (Knuth, Donald) was used to compute gap-free solutions. When combined with Dancing Links, it is well known for its high performance in recursive problems. Hence editing it to work for the three-dimensional bounded knapsack problem seemed like the perfect idea. Furthermore, grouping the basic pieces into bigger segments, before fitting those in the container, for better efficiency was another path that was explored.

4. Algorithm X+

Algorithm X was modified to work for partial covers as well, allowing solution with empty spaces in the container without sacrificing the efficiency of the dancing links.

This paper consists of brief descriptions of the algorithms and how they operate (Section 3), the java implementation (Section 4), together with the procedure of all the experiments conducted (Section 5). This also includes the constraints, the hardware and finally the results. In the section about the conclusions and discussion (Section 6 and Section 7), further elaboration of the project and applications can be found. Additionally, there is a reference list with all of the used literature (Section 8). Finally you can find the list of appendices (Section 9).

3 Algorithms

During the production process, most attention was focussed on 4 separate algorithms including two variations of algorithm X, a ‘greedy’ algorithm and a genetic algorithm.

3.1 Sticky greedy

A greedy algorithm (Annu) is a quick and relatively easy way of computing a decent solution to the problem. The standard greedy approach has been altered with a performance measure to make the solutions of the algorithm more feasible.

The Sticky greedy algorithm works collaboratively with three main algorithms: An ordering algorithm, a subspace algorithm and a placement algorithm. In this section, all these three algorithms will be discussed and put together to form the main algorithm.

By placing the pieces with the highest values, it still resulted in a relatively high score despite the amount of gaps left inside the box. This algorithm only aims to optimise the total score by mainly focusing on the value of the parcels. In practice, this algorithm would give at least 50% of the best score achievable. Although more often than not, it results in a higher percentile score. Simultaneously, it serves as a viable starting algorithm to use to relativise the scores from more accurate algorithms.

3.1.1 Ordering algorithm

The search algorithm determines the order of the pieces to be placed by prioritising pieces with higher values densities. These are calculated using $value\ density = parcel\ value / parcel\ volume$. Since these values and volumes are the same for each parcel type, the algorithm will calculate its corresponding value density and order them correctly.

```
1)  OrderParcels(amount of each parcel type, value of each parcel type, volume of each parcel type)
2)      VDP = [value for each parcel type / volume of each parcel type] (value density of each parcel type)
3)      parcelOrder = []
4)      if at least one VDP is different
5)          for parcelType in parcel Types do:
6)              find max (VDP) and min (VDP)
7)          middle (VDP) is the remaining one
8)      else order max middle min with order given
9)      for parcel type in ordered list
10)         instantiate amount of parcel type
11)     return list (max middle min)
```

3.1.2 Subspace algorithm

The subspace algorithm creates a new subspace from the current state of the container in which the placement algorithm will calculate all possibilities to fit the next piece it gets from the list that was produced in the selection algorithm.

```
1)  GenerateSubSpace(container, nextPieceToPlace)
2)      newWidth = container.width + nextPieceToPlace.width
3)      newHeight = container.height + nextPieceToPlace.height
4)      newDepth = container.depth + nextPieceToPlace.depth
5)      newsubSpace = copy of container with dims newWidth, newHeight, newDepth
6)      return newsubSpace
```

3.1.3 Placement algorithm

The placement algorithm brute forces through all the possible locations of a parcel within the subspace given, and saves each of those states in a list. Finally, it selects the best state based on a simple heuristic that is referred to as “adhesion”. Adhesion measures how much the piece that is placed in the state is touching other placed pieces or the sides of the container, in order to minimize the empty spaces in the container. It repeats this process, updating and continuing with the best state found until there are either no more parcels or there is no more space in the box. At this point, the final solution has been obtained, which can then be returned.

```
1)  FindSolution()
2)      orderToPlace = OrderParcels(amount of each parcel type, value of each parcel type, volume of each parcel type)
3)      container = empty container with corresponding width, height and depth
4)      while ( pieceToPlace = orderToPlace.next() exists and container.spaceLeft()):
5)          solutionsForPiece = []
6)          segmentToFill = GenerateSubSpace(container, pieceToPlace)
7)          for all permutations of pieceToPlace:
8)              for (point in segmentToFill)
9)                  if (can place piece) add to solutionsForPiece
10)         pick state from solutionsForPiece with highest adhesion and set it equal to container
11)     return container
```

3.2 Genetic fitting algorithm

3.2.1 Placement algorithm

The idea of implementing a genetic algorithm for bin packing (Falkenauer) emerged first in a previous work, where the objective was to create a bot capable of playing a Tetris-like game. The bot is able to compute every possible move, including placement and rotation, and attribute to each single situation an appropriate score. From there, the configuration with the highest score is chosen. This algorithm follows the same idea, and is merely adaptation from a two-dimensional environment to a three-dimensional one. It starts by filling one end of the cargo space, and slowly advances towards the other end.

The scores are based on four heuristics: adhesion, number of holes, depth reached, number of segments completed and actual value given to the parcels.

The **adhesion** is a score that measures how much each piece is in contact with another, or with the wall of the container. The idea behind it lies in the fact that the more the parcels are touching each other, the better the space configuration.

The **number of holes** is a representation of the volume of unoccupied space. In practice, one hole is counted as a volume of space measuring a quarter of a cubic meter not occupied by a parcel. It calculates this score up to the point where the further piece is placed. This weight is expected to be negative, as fewer holes means that the space is more optimized.

The **depth reached** is obtained by computing the distance between the first end of the cargo and the furthest point where the further parcel is placed. It also represents the idea of space optimization within the configuration.

The **number of segments completed** is obtained by adding all the different segments that were generated and filled entirely together. It is the final aspect of the space representation within the score evaluation.

Last but not least, the **actual values** of the parcels that are encoded by the user are also taken into consideration. This is the value representation for the score evaluation function. However, if the weight is too high, it would mimic the greedy algorithm.

1) while (number of parcels>0)
2) compute the working space of the cargo
3) compute different combinations of parcel that makes up 10 moves ahead
4) try all possible placement and rotation and compute the score
5) chose the state with the best score and update the working space and the parcels numbers

3.2.2 Training

The relative importance of each score can be modified by attributing a unique weight to the performance measures. A good way of optimizing these weights is to use a genetic algorithm.

In order to train a genetic algorithm, one must select a few components mimicking biological evolution. First, there must be a fitness measurement in order to score the different agents. In this case, the fitness is represented by the the final score that the agent gets after filling the container with parcels.

In this paper, elitist selection (Thierens, Dirk, and David Goldberg) is used as a selection method. This means keeping the best scoring parents of the previous generation when calculating the next one. Furthermore, a random crossover is used, selecting either one of the two genes of a parent for each gene. The mutation rate of a gene to be changed was set to 30%.

In the experiments section of this paper, more will be elaborated on this.

1) Create pool of N agents with random weights between 0 and 1
2) for m = 0 to NUM_GENERATIONS:
3) for each agent:
4) score the agent with the container value it gets
5) create new pool with the agents using elitist selection (in here mutation is also applied)
6) return the results

3.3 Algorithm X

	1	2	3	4	5	6	7
B	1	0	0	1	0	0	0
D	0	0	1	1	1	1	0
F	0	1	0	0	0	1	1

Figure 3; Example of an exact cover problem (https://media.geeksforgeeks.org/wp-content/uploads/ProbMatrix12.jpg)

Algorithm X is known to work well for finding solutions for many exact cover problems as it is a recursive, non-deterministic, depth-first, backtracking algorithm that has a built-in way of pruning out bad solutions and trying the best solutions first. Figure 2 shows the Algorithm X representation of an exact cover problem.

In our case of the translated problem, the columns represent all the different positions in the grid. A three-dimensional field can easily be translated to this representation by adding a column for each position in 3D-space. This is meant to represent the rule that at each cell in the grid there can only be one parcel at once, preventing overlapping placement.

Furthermore, each row in the grid represents a position of one permutation of one parcel inside the grid. For each row there is a zero in the corresponding column if the permutation of that row is not placed in that cell, and a one if it is.

The application of Dancing-Links algorithm in an earlier related project, fitting pentomino-shaped objects in a two-dimensional grid, showed astounding results. Which is why it led us to believe that it could, in theory, work to find a way to fill the container with A,B and C or L, P and T pieces in a time-efficient manner.

However, the first attempt to implement this algorithm for A, B and C pieces was far from being conclusive: the algorithm ran for multiple days and no viable solution resulted from this. This was with pruning. Every position of a parcel that is one block away from the edge of the container is not added to the linked list, as no piece, at least for the A, B and C pieces, would be able to fit in that one-wide space. This reduces the size of the linked list dramatically, namely by ~40% of the previous size. Unfortunately the algorithm was unsuccessful in finding a perfect solution for the problem.

Conclusively, there most likely exists no exact cover solution for this parcel combination, even though not all possibilities were checked. Given the efficiency of the algorithm, multiple billions of possible configurations within that time frame were explored. Furthermore, running the algorithm for the

parcels L, P and T, many solutions were found instantaneously. Therefore, it is safe to assume that no solution exists for the combinations of A, B and C parcels with a container of that size.

3.3.1 Segmentation of pieces

In an effort to reduce the search space for the A, B and C pieces, the pieces were put together into bigger so called segments, thereby (theoretically) making the problem smaller. In this way, it would be possible to go through all the possibilities with algorithm X. Figure 3 represents one of many possible segments.

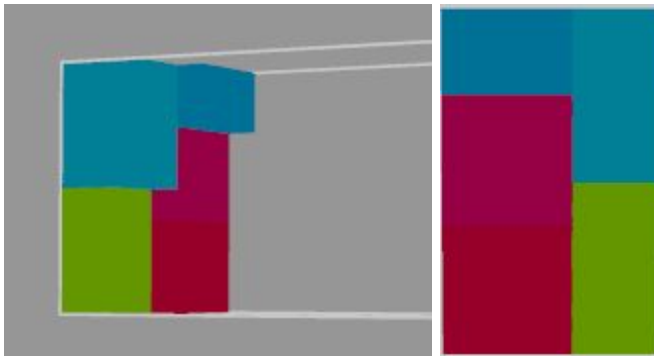


Figure 3; Example of a 3D segment (left) and a 2D segment (right)

In Figure 3, one two-dimensional representation is shown on the right, and on the left, one of the many three dimensional results of its 2D counterpart. The segments are based on a 2D representation of the container having the dimension of its width and height. In there all spaces are completely filled by 2D representations of the pieces, meaning that the length and width (but disregarding the depth for now) has to be completely filled.

All possible slices are then converted back into 3D to obtain all segments that have at least one layer (1 by 4 by 2.5 area) completely filled, but varying depths. An exact cover can then be made by fitting these pieces together into the container.

However this experimentation concluded into a mild failure, the previous step resulting in a tremendous amount of segments, which was over a million after taking out duplicates. Therefore the computational power required was still way too high and could therefore not find the solution.

3.4 Algorithm X+

Algorithm X normally works as such: it finds a column such that the amount of possible pieces placeable in that column are minimum, or in practical the least amount of rows, and iterates through all those rows until there are no more rows left and then backtracks. This process is repeated until every path is explored, and gives a solution if all the columns are filled, ie: the container is completely filled.

However, for this problem, the goal is not necessarily to obtain an exact cover, as there might not be any. Therefore, instead of always returning the column with the least rows, this adapted algorithm skips those with zero rows and leaves them as empty spaces.

To reduce the increased branching factor caused by ignoring these empty spaces, the algorithm prunes each partial solution by looking at the score per block placed to determine if a solution can be better than the current best one.

$$\frac{s}{vP + vE} > \left(1 + \frac{vP + vE}{vT \cdot S}\right) \cdot \frac{S}{vT}$$

s = current score of partial solution
S = current best score found by algorithm
vP = volume of all pieces currently in solution
vE = total amount of 'empty' columns in current solution
vT = total volume of the container

This formula determines if a branch (partial solution) will continue being searched. When the first piece is placed, it only needs to have the same score per block as the current best solution, but as more pieces are added, it needs to be slightly better than the previous one, until the last piece, where the score needs to be exactly 1 better than the current one. This will prune away every solution that isn't better than the previous one, though it can also remove some solutions that could be better, but have a lower score at the start.(exact 7/heuristic 4) (see pseudo code on next page)

Another way of pruning was only allowing the same or less empty spaces for each solution. After implementing this, it ended up pruning away too many possible good solutions that it wasn't faster than not using it. So it was removed for the final product.

The initial linked list pruning for A, B and C pentominoes that was used for the exact cover (Section 3.2) could also be used for this algorithm. If a piece is placed one block away from the wall of the container, no other piece will be able to fit in between them. Even when allowing these empty spaces, each solution that has a piece one away from the wall could be generated without these positions by placing these pieces right against the wall, so no possibilities for solutions are lost. (X+ 6)

To increase the speed of the algorithm, seemingly counterintuitive, multiple searches are run simultaneously. This allows for multithreading and increases the probability that a good solution is found right from the start. This is significant due to the fact that X+ is still a depth first algorithm and it takes a long time to get all the way back to the start, to replace the pieces there. If not for the multiple searches, the algorithm could start with a bad combination of pieces. It could then take a long time to find a good solution it could have gotten immediately with one of the other searches. By having all of these searches look at and change the same best score, they will all get pruned with the same score for the average score per block. Each search is done using a different orientation for the cargo space (33x8x5, 33x5x8, etc). This makes it so different orientations of the first piece are used and the algorithm goes through the space in a different way. (X+ 1)

Lastly, when initialising the linked list, the items are added starting from the one with the highest score per block. Because algorithm X goes through the pieces from first to last added, this change effectively makes X+ a greedy algorithm. (X+ 4)

All of these Optimizations and prunings can be used for the exact cover version of algorithm X as well and could perform better in certain cases, because of the added pruning for the empty spaces. This version of X+ will also be tested and compared to the ‘normal’ X+. (exact)

3.4.1 Pseudo code

X+:

```
1)  for each orientation of the container:
2)      create new instance of algorithm
3)      initialise column nodes of linked list
4)      sort input items by highest value/block
5)      for each position of each input:
6)          if not ABC or if not 1 away from edge:
7)              add input to linked list
8)      run DLXexact/DLXheuristic
```

Exact:

```
1)  if time not up:
2)      if found a solution:
3)          global best score = score
4)      else:
5)          select column node
6)          for every row:
7)              if piece can still be placed and if (formula mentioned before) is true:
8)                  continue search
```

Heuristic:

```
1)  if time not up:
2)      select column node*
3)      for every row:
4)          if piece can still be placed and if (formula mentioned before) is true:
5)              continue search
```

Select column node:

```
1)  for every column node:
2)      if amount of rows < minimum:
3)          if amount of rows != o:
4)              if at least one piece in the column can still be placed:
5)                  min = current column
6)          else:
7)              empty spaces += 1
8)  if column found:
9)      return column
10) else:
11)     if score > global best score:
12)         global best score = score
13)     return any column
```

4 Java Implementation

4.1 Unified Modelling Language Diagram

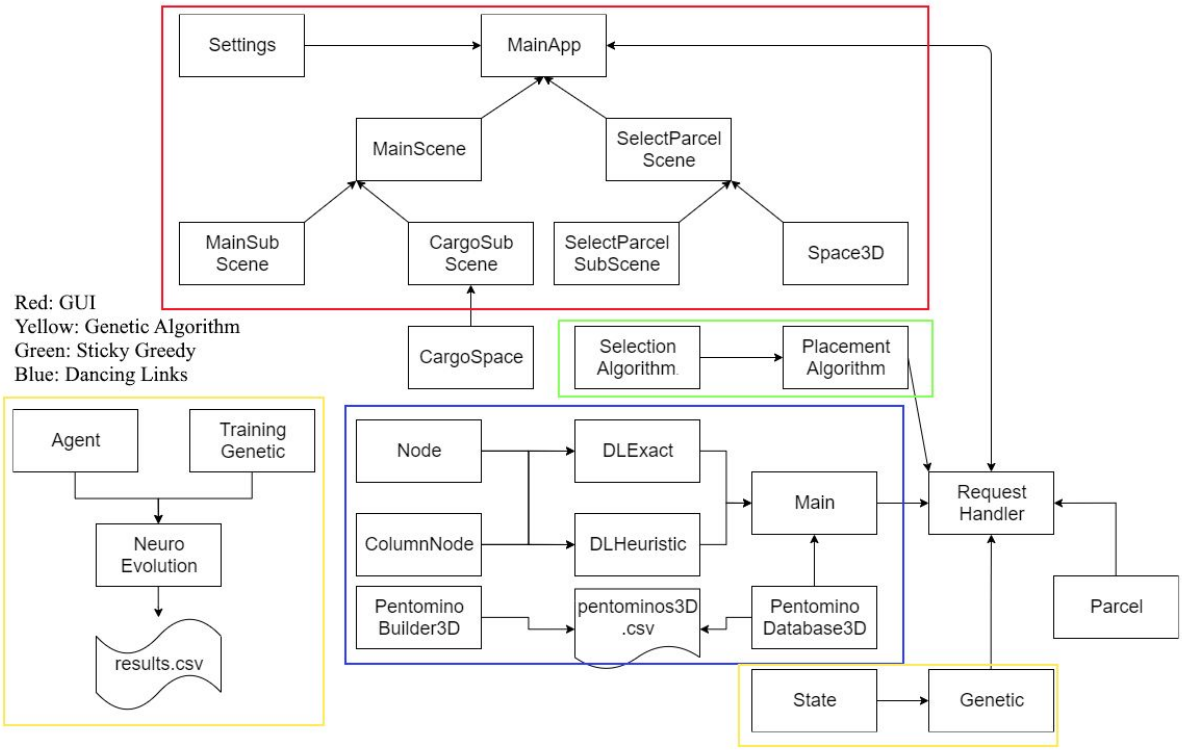


Figure 4.0 UML-diagram code

Settings: Contains the settings for the container and the UI (e.g. sizes, paths to gifs,...).
MainApp: The Stage for all of the JavaFX application. It's the main point of connection for all the UI elements.
MainScene: The Scene that will show when you start the application.
SelectedParcelScene: The Scene that will be shown when the "Select Parcels" button is pressed.
SelectedParcelSubScene: Contains the buttons and textfields for selecting the parcels.
Space3D: This class represents the subscene containing a grid with 3d objects in it (the actual x,y,z grid)
MainSubScene: Represents the selection/button/images part of the main menu
CargoSubScene: This class represents the subscene containing the cargo space with 3d objects in it
CargoSpace: Represents the cargo space to fill the parcels in
SelectionAlgorithm: Algorithm used to select the order in which to place the list of parcels
PlacementAlgorithm: Represents the algorithm that places the parcels in the container based on input data from the selection algorithm

RequestHandler: Represents a request to the application to solve the problem using a certain algorithm. Main connection between the UI and the algorithms
Parcel: Represents an abstract notion of a parcel
Genetic: The file that contains the Genetic Algorithm
State: The state of the container. It has a corresponding score that is given by the weights of the genetic algorithm and the earlier described heuristics
Main: Represents the connection between the data and the Dancing Links algorithms. Talks directly to the Request Handler.
PentominoDatabase3D: This class takes care of reading all pentominoes and their permutations from a CSV
pentominos3d.csv: The csv-file containing the 3D-representation of the parcels
PentominoBuilder3D: Builds pentominos3d.csv
DLExact: Represents a dancing links implementation of algorithm X
DLHeuristic: Represents a dancing links implementation of algorithm X made compatible for holes
Node: Represents a single node in a 4-way linked list. A node has a left-, right-, up-, downwards connection and has a column node (the node of the column it is in).
ColumnNode: Represents a single column node in a 4-way linked list. A node has a left-, right-, up-, downwards connection and has a column node (the node of the column it is in).
Agent: This class represents an agent within a certain environment
TrainingGenetic: Runs the application without the GUI using the weights from the agents, calculating and returning their score
NeuroEvolution: This class represents the genetic algorithm strategy. It takes a pool of agents and with a score function it can run
Results.csv: The results of the trained genetic algorithm

4.2 Visualisation

4.2.1 JavaFx setup/download

It can be a little bit tricky to setup JavaFx at first. The way JavaFx is used in our group is using the Integrated development environment (IDE) IntelliJ with a downloaded JavaFx software development kit (SDK). To use these pieces of software you need to download them both first from the internet, you can find IntelliJ on and you can get JavaFx from the official oracle website.

After installing both programmes, start up IntelliJ. Now you will have to make sure that IntelliJ can use JavaFx. In the “project” that you want to run your JavaFx program, add a pathway to the downloaded SDK path. Now go to project structure again and select “lib”, you will now have to add the path of the JavaFx SDK package in the library. Now you can run the program.

Alternatively, the application can also be run from the command line. This also requires Java and a compatible version of JavaFX (in this project, version 13.0.1 was used). When in the command line, first head to the directory *"src"*. Then, execute the following commands in correct order:

1) *javac --module-path "Path\To\JavaFX\javafx-sdk-13.0.1\lib" --add-modules javafx.controls,javafx.media UI/MainApp.java*
2) *java --module-path "Path\To\JavaFX\javafx-sdk-13.0.1\lib" --add-modules javafx.controls,javafx.media UI.MainApp*.

If everything goes correctly, the GUI of the application should pop up.

4.2.2 Functionalities and user guide

See appendix for a user manual with corresponding functionalities

4.3 Design choices

A minimalistic design was used to create the interface. This is because there are plenty of choices to be made within the application, such as which algorithm you want to use, if you want to get an exact cover of the container and how many boxes and which types of boxes you want to use. Therefore the design is made to be minimalistic, efficient, clear and fast. There are also some funny easter eggs in, ready for exploration.

5 Experiment

In order to measure the performance of the algorithms, one of the parameters taken into account is the total value of the container. Alternate means of assessing efficiency is based on the execution speed of the algorithms. The effectiveness of the pruning strategies were also tested

5.1 Benchmarking Settings

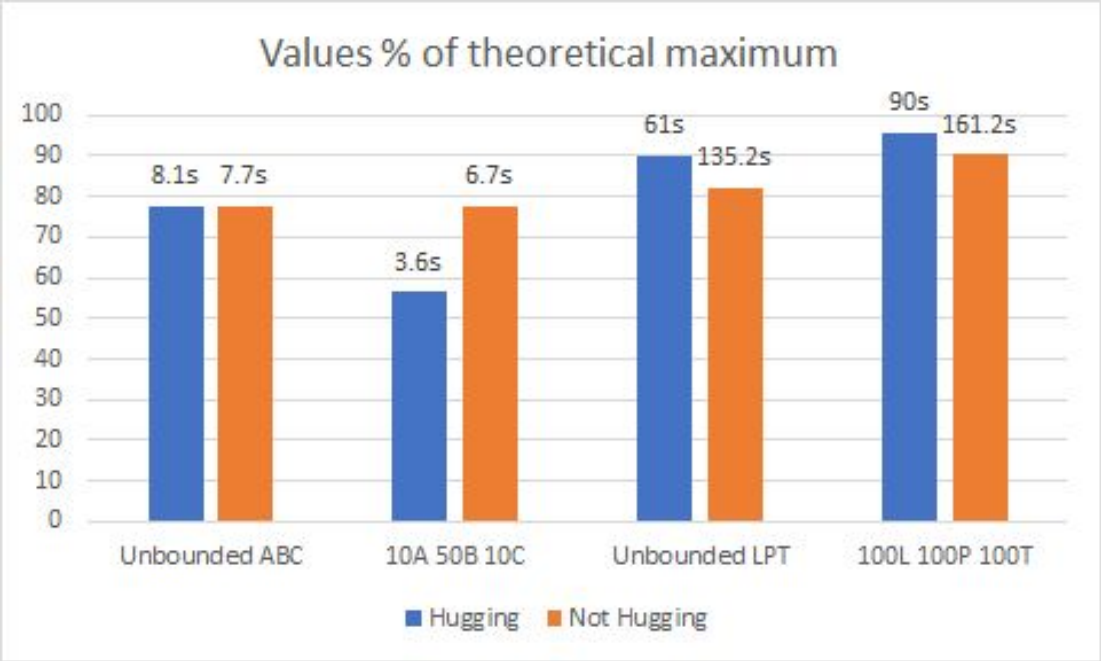
The experiments were performed using a desktop with a stock Ryzen 5 3600 CPU running with 16GB of RAM. All algorithms were limited to a single core by setting the process affinity in the task manager to only allow it to be scheduled on a single thread.

5.2 Results

All results were obtained with values 3,4 and 5 for the A,B and C and the L,P and T pieces respectively. While all the algorithms will work for different values for the pieces, this was not tested. Scores are shown as the percentage of the theoretical maximum score that can be achieved. The maximum score was calculated by choosing the combinations of pieces with the highest combined score that, by just looking at their total volume, could still be placed inside the container.

Four different methods of testing the algorithm were chosen to assess their efficiency. The first one, unbounded ABC runs the algorithm with 999 A, B and C parcels, so it can fit all of one type in a container. The second test, 10A, 50B, 10C forces the algorithm to make decisions regarding which parcel to place. The same concept goes for the LPT-parcels. The respective maximum theoretical values for each task are 247.5, 227, 1320, 1092. This will be used as a relative measurement in the tests. As the first two algorithms yield only one solution and stop searching afterwards, the time execution is more relevant than comparing the result within a certain time frame. However, as X+ goes through many solutions, multiple time limits were chosen to show how the scores evolve over time and if there is a score increase at all. 30 Seconds was chosen as the first time, as this would be the time that the algorithm would probably be run in a real life situation.

2.1 Greedy Algorithm results

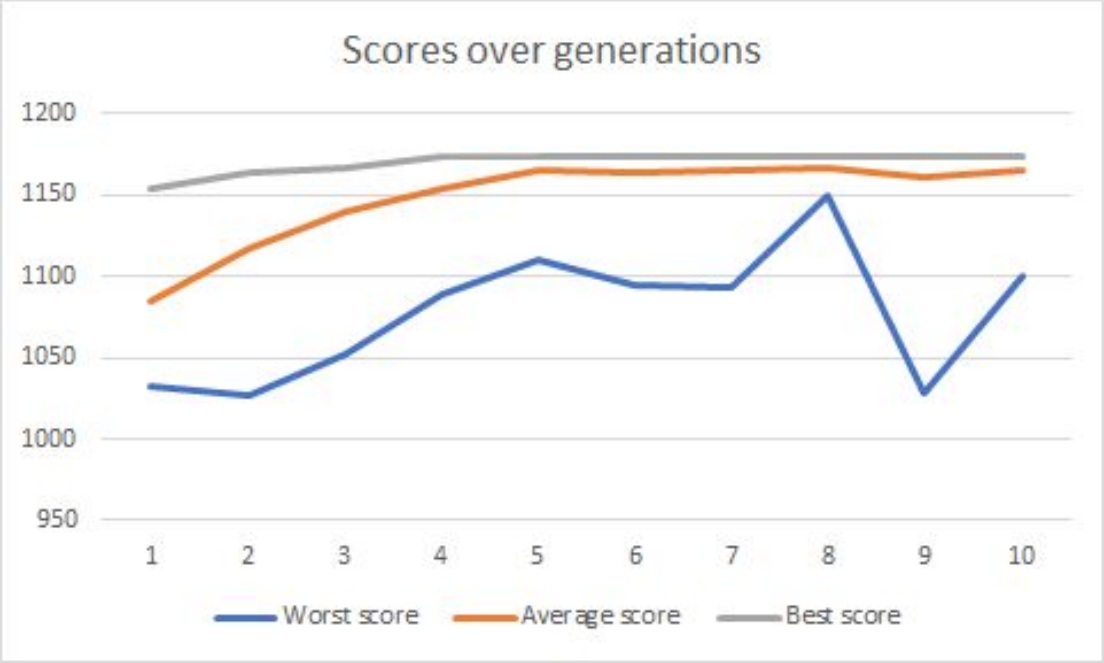


(1.1)

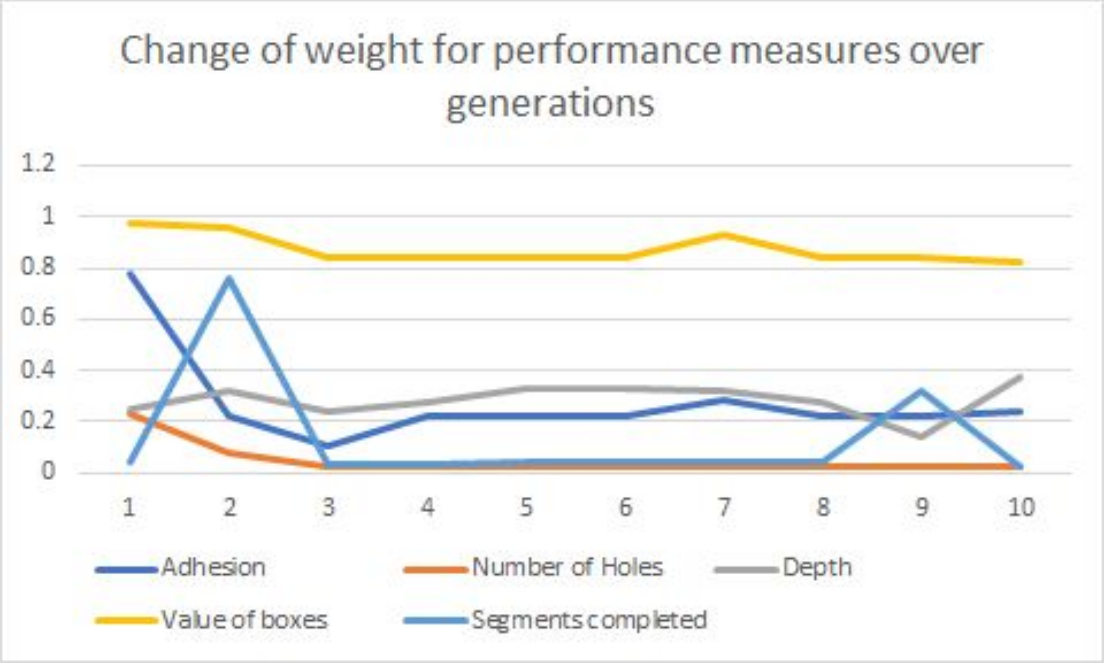
The figures (1.1) displays the results for the greedy algorithm, with a limited and an unlimited amount of parcels. The blue bars show the result of the actual greedy algorithm. For comparison, the orange bar represents the score without the sticky algorithm. The score is displayed in percentage of the theoretical maximum for each situation. On top of each bar, the execution time is shown in seconds.

For the A, B and C parcels, when unlimited, the score is 192, which is about 77.5% of the theoretical maximum; while the limited amount of parcels yields a result of 128, which is 56% of the maximum. As for the L, P and T parcels, if unbound, the score is 1189, or 90%; and when limited, the score is 1047 or 95%.

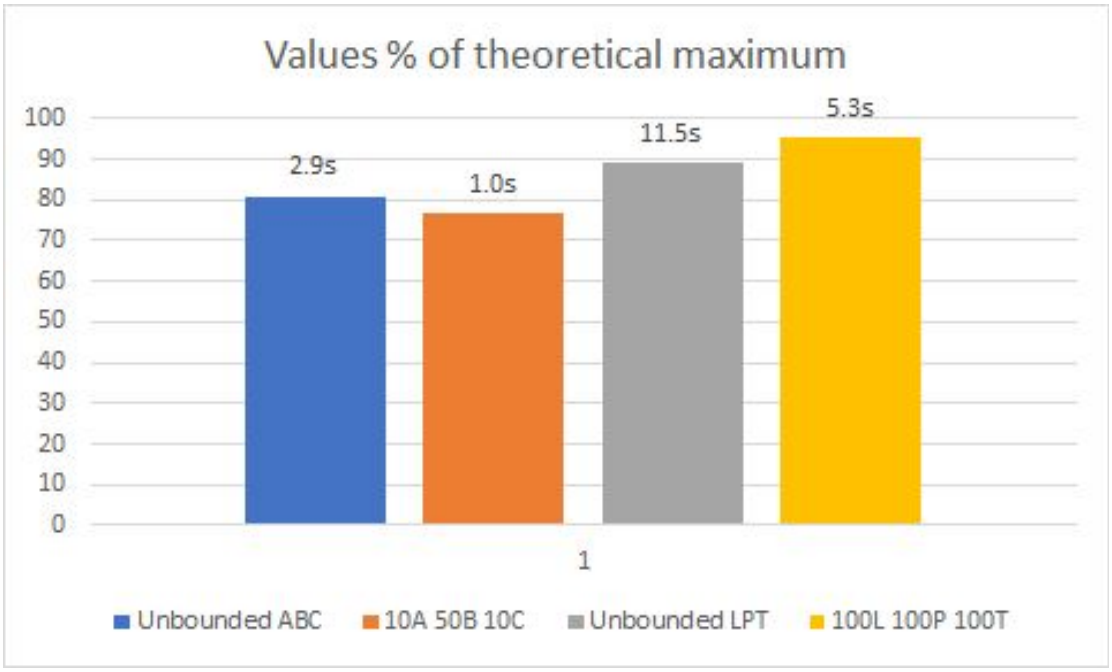
5.2.2 Genetic Algorithm results



(2.1)



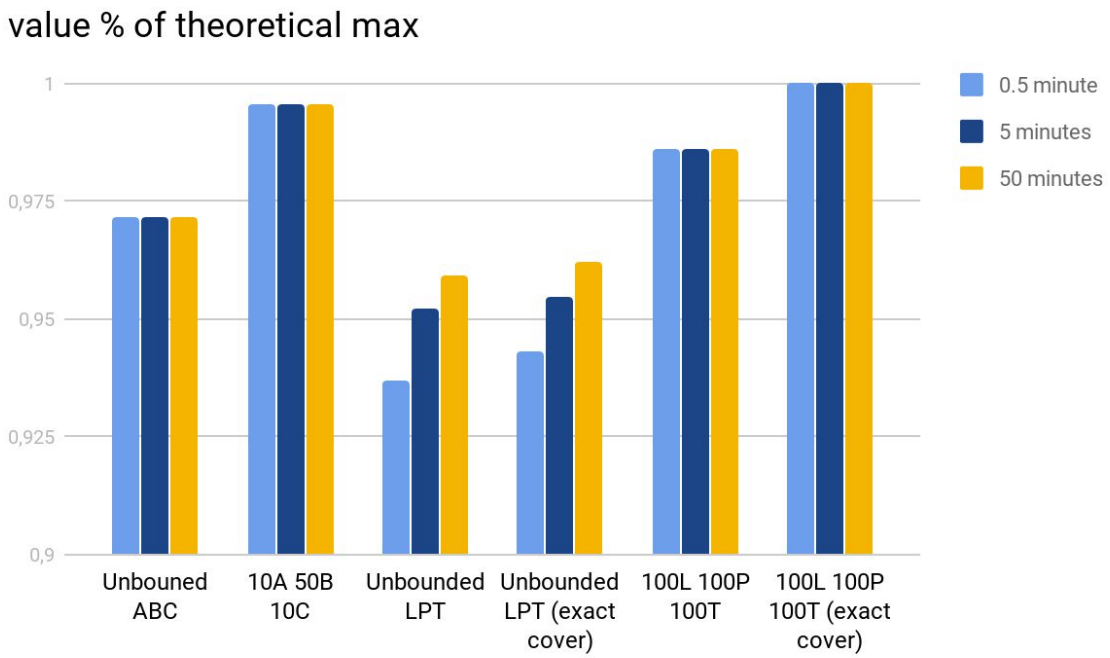
(2.2)



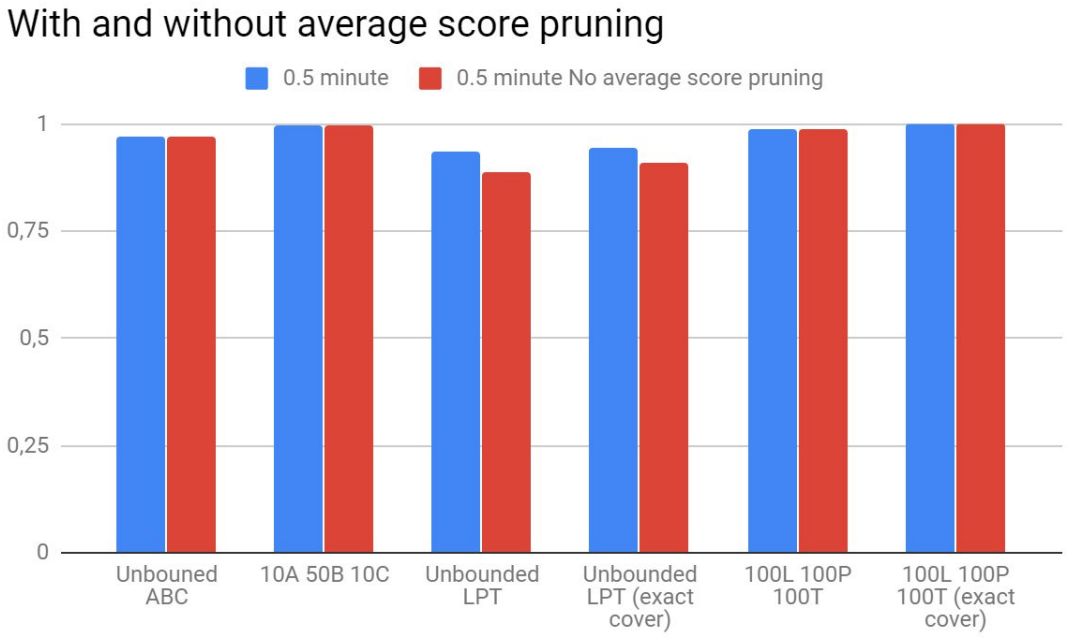
(2.3)

The genetic algorithm was trained for 10 generations. In (2.1) it can be seen that the best score after training for the LPT parcels is 1174. Overall, the average is also increasing. In (2.2), it can be noticed that the three performance measures with the highest weights are the values of the boxes, the depth and the adhesion. The third graph (2.3) shows that the genetic algorithm scores from 78% to 96%. The runtime for every task is also shown on top of that.

5.2.3 Algorithm X+ results

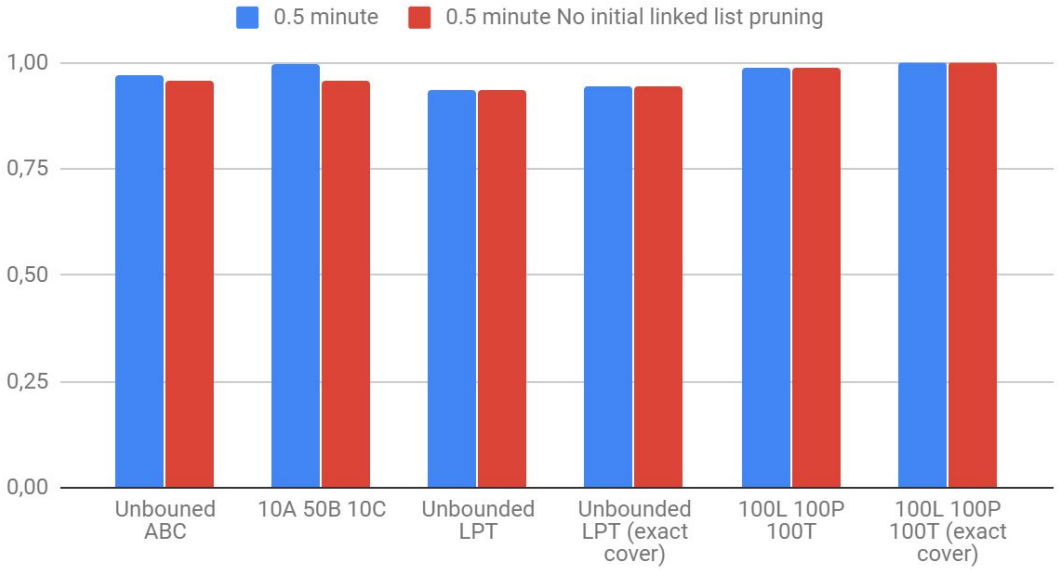


(3.1)



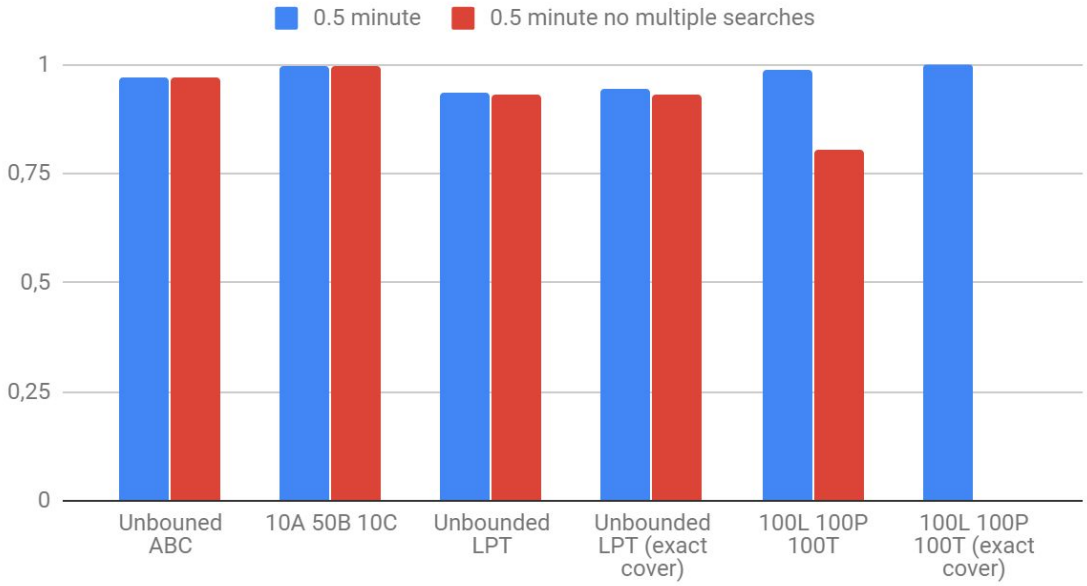
(3.2)

with and without initial linked list pruning



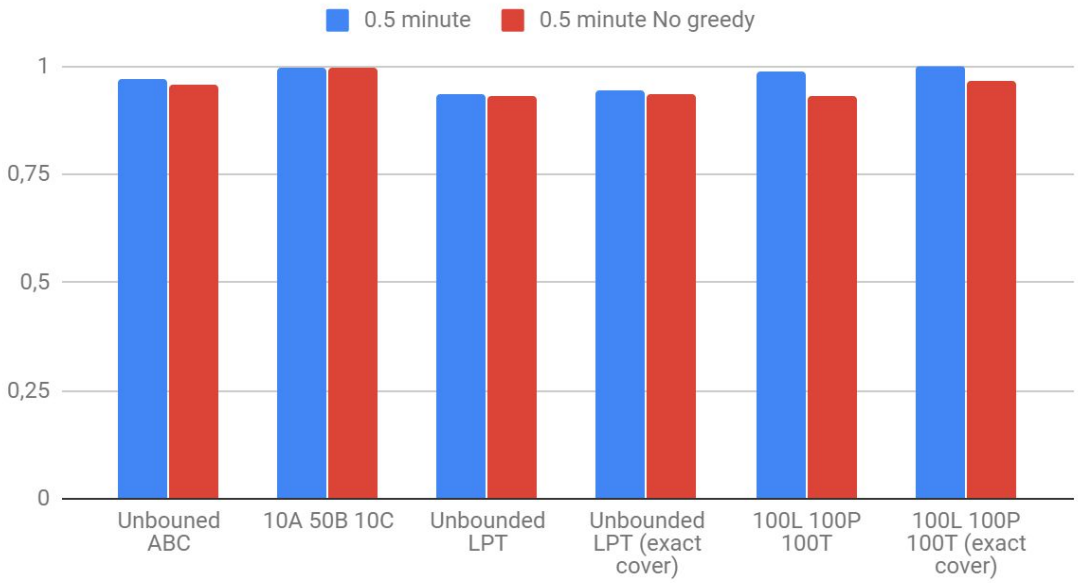
(3.3)

with and without multiple searches



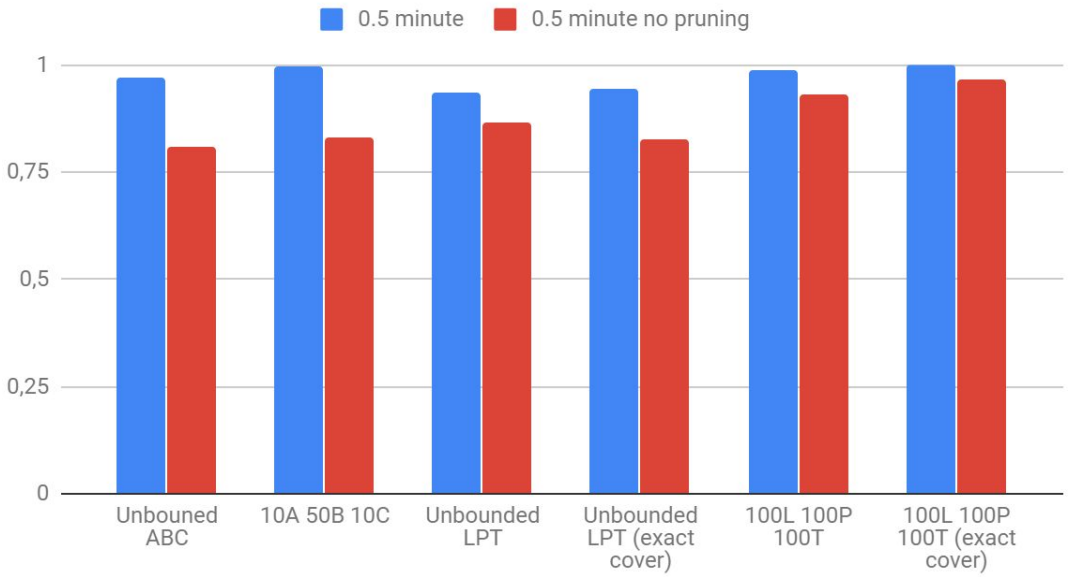
(3.4)

with and without greedy



(3.5)

With and without any pruning



(3.6)

(3.1) The first graph shows the X+ results for different amounts of time. The only change in score with more time is for the unbounded LPT pieces. It went from 93,7% to 95,9% for the Unbounded LPT and from 94,3% to 96,2% for the exact cover. The exact cover performed slightly better. Note that the vertical axis is only from 90% to 100%. Note as well that even though the lowest given time was 30 seconds, some of the final scores were gotten within the first few seconds of the search.
(3.2) The next graph shows a notable decrease in score for the unbounded LPT pieces, from 93,7% to 88,8%, when taking out the pruning for the average score. The other scores are within margin of error.
(3.3) In this graph there is a minor decrease in the scores of the ABC pieces when the initial linked list isn't pruned. The unbounded score goes from 97,2% to 96% and the bounded one goes from 99,6% to 95,6%. The other scores were not influenced.

(3.4) When only a single search is run, instead of multiple ones, there is a minor decrease in both the unbounded LPT and the exact cover scores, from 93,7% to 93% and from 94,3% to 93,1% respectively. The bounded LPT scores however, showed a bigger decrease, the exact cover one not even finding a solution within the 0.5 minute time limit. The other ones score decreased from 98,6% to 80,7%.
(3.5) When the search is done by adding the pieces to the linked list randomly instead of in a greedy way, there are small decreases in all scores except for the bounded ABC one, with the most notable
(3.6) being the bounded LPT score, that went down from 98,6% to 93%. In the last graph, the scores without any pruning are compared to the ones with all of the pruning. Here there is an increase in every single case, with an average increase of 12% over the version without pruning.

6 Discussion

The task of optimization is one which will always present itself in the developmental stages of any program. Through the process of finding ways to maximise the scores and minimise time, the evidence becomes apparent which algorithm performs best. After a lot of number crunching and testing it finally becomes possible to compare the results and draw definitive solutions.

6.1 Interpretation

6.1.1 Sticky Greedy

It was expected that the greedy algorithm obtained below average scores probably due to its lack of versatility. Prioritizing pieces with highest scores, it has neglected configurations with other lower valued pieces, even though it could have ended up with a better final score. This algorithm trades the complexity of combinatorial search for a more direct and simple approach. However, the scores it obtained were exceeding our expectations, especially for the LPT pieces. The placement algorithm managed to arrange the parcels in such a way that allowed for lower value parcels to be placed as well. However, this didn't come without its consequences. The runtime of the algorithm, especially with LPT parcels increased significantly with higher amounts. Overall, the greedy algorithm performs unexpectedly well.

6.1.2 Genetic Fitting

The genetic algorithm was supposed to overcome the lack of versatility of the greedy algorithm by implementing the option of attempting any parcel type in the container and selecting the most optimal configuration out of the set. For the most part, it performed better and was significantly faster.

However, with the unbounded LPT parcels, it performed worse than the greedy algorithm. The reason for these findings is simply that the training was not optimal due to a lack of time. The training was also converging to a local minima, as the performance measures or selection methods might not have allowed for better configurations. A possible solution that could resolve these issues would be to train the algorithm with a sparse amount of moves ahead, i.e. the number of parcels it tries to fit at the simultaneously.

6.1.3 Algorithm X+

Figure 3.1 displays the 'achilles heel' in X+. For a large amount of possibilities, it can't get through many of them and the probability of not finding an optimal solution is somewhat high. This is all relative though, as even these scores are still well above the 90% of the theoretical limit.

The following graphs (3.2-3.5) display the performance of X+ without certain prunings, but with the rest of them. It is important to note that different scores are affected for each removed pruning. This indicates that each of these prunings is significant and each serve a specific purpose. The average score pruning influences the Unbounded LPT scores as it removes a lot of possible bad solutions and therefore decreases the search space. This will have the most effect on cases with the biggest branching factor, which are the unbounded LPT pieces. The multiple searches again makes the biggest difference for the LPT pieces as X+ will take a long time to get back to the first few pieces in these cases. The extra searches will result in the algorithm finding good results more quickly. For example the exact cover 100(LPT) didn't find a solution at all within the given time limit without these extra searches. The initial linked list pruning only influences the ABC pieces, which is expected, as this pruning is not done on the LPT pieces. The increase in score comes from the more optimal placement of the pieces, as putting them all the way at the sides will allow them to fit another piece next to them. The combination of all of these pruning methods (3.6) gives a much higher score for every single test case. It also shows a bigger increase when all the pruning methods are used than what it gets for every single optimization. This shows that the different prunings that were used compliment each other very well.

6.2 Comparison

Evidently, algorithm X+ is the most optimal algorithm, as it is the quickest and gives the highest scores. The performance of the Sticky Greedy algorithm is much closer to the performance of the Genetic Fitting algorithm. However the Genetic Fitting algorithm is much faster at finding a solution. Overall the Genetic Fitting algorithm also usually finds a better solution, but this depends on the problem. While the Sticky Greedy algorithm was slightly worse overall, it did perform better than initially expected.

6.3 Future Outlook

In the real world, these algorithms can be applicable in packaging. If the contents of the package can be represented as polyominoes, and the container is represented with exact dimensions, the algorithms will be able to work to solve these problems. Perhaps if given more time the algorithms could have been optimized to solve the problem for different shaped containers, and with a wider variety of contents. There are a few alternate courses of action that could have been undertaken.

For example, algorithm X+ could implement a way to change the parcels placed at the beginning instead of optimizing the last placed ones, because it now will never get all the way back to the start to replace these pieces in case they are not placed optimally. A way to increase the spread of the pieces, to potentially make them fit together better, choose parcels based on the amount that is left. One last suggestion is to adapt the amount of pruning during the search, to make it more strict if a lot of bad solutions are found or to make it less strict if no new solutions are found (e.g. for the exact cover search).

For the genetic algorithm, there could be looked into more complex performance measures and fitness functions in the future. A different selection algorithm like mtDNA (Shrestha, Ajay) could be used in the future to prevent getting stuck in local minima.

7 Conclusion

In this paper, the 3D bounded knapsack problem was addressed with three different approaches. Out of all the viewpoints taken, algorithm X+ performed best, obtaining an average of 97% of the theoretical maximum score. The other two algorithms established a score of about 80-85%.

After extensive testing, we have, as already mentioned previously concluded that it is not possible to completely fill a cargo space with A, B and/or C parcels, without having any empty spaces. Furthermore, if we give the parcels A, B and C a value of 3, 4 and 5 respectively, then the maximum value that our algorithms have found is 240 (97.2%). Regarding the L, P and T parcels, it is actually possible to fill completely without any gaps. Lastly, if we give the parcels L, P and T a value of 3, 4 and 5 respectively, then the maximum value that our algorithms have found is 1274 (96.5%). Coincidentally, this is an exact cover.

To make our algorithms more compatible with real life situations like the priority shipping that was mentioned, the research could be expanded by turning existing packages into polyomino forms, packing those into a container and trying to optimize for this. The algorithms proposed in this paper could be optimized in any of the ways described in “Future Outlook”.

8 References

1. Knuth, Donald E. *Dancing Links*, Stanford University, 2000.
2. Pisinger, David. “Heuristics for the Container Loading Problem.” *University of Copenhagen*, European Journal of Operational Research 1, 2002, pp. 1–11.
3. Annu, Malik, and Sharma Anju. “Greedy Algorithm.” *SES BPSMV University*, 2013, pp. 1–5.
4. Yadav, Veenu, and Shikha Singh. “Solving 0-1 Knapsack Problem with Genetic Algorithms .” *ASET Amity University Lucknow*, Amity University, pp. 1–3.
1. Kolhe, Pushkar, and Henrik Christensen. “Planning in Logistics: A Survey.” *Georgia Institute of Technology*, pp. 1–6.
2. Falkenauer, Emanuel. "A hybrid grouping genetic algorithm for bin packing." Journal of heuristics 2.1 (1996): 5-30.
3. Thierens, Dirk, and David Goldberg. "Convergence models of genetic algorithm selection schemes." *International Conference on Parallel Problem Solving from Nature*. Springer, Berlin, Heidelberg, 1994.
4. Shrestha, Ajay, and Ausif Mahmood. *"Improving genetic algorithm with fine-tuned crossover and scaled architecture."* Journal of Mathematics 2016 (2016).

9 Appendices

Score for Genetic and Greedy algorithms						
Genetic			Greedy with adhesion		Greedy without adhesion	
	Score	Execution time	Score	Execution time	Score	Execution time
Unbounded ABC	200	~ 2.9s	192	~8.1s	192	~7.7s
10A, 50B, 10A	174	~ 1.0s	128	~3.6s	176	~6.7s
Unbounded LPT	1174	~ 11.5s	1189	~61s	1085	~135.2s
100L, 100P, 100T	1041	~ 5.3s	1047	~90s	987	~161.2s

Score for algorithm X+								
	0.5 minute	5 minutes	50 minutes	0.5 minute no multiple searches	0.5 minute No initial linked list pruning	0.5 minute No average score pruning	0.5 minute no pruning	0.5 minute No greedy
Unbounded ABC	0,972	0,972	0,972	0,972	0,96	0,972	0,81	0,955
10A 50B 10C	0,996	0,996	0,996	0,996	0,9556	0,996	0,833	0,996
Unbounded LPT	0,937	0,952	0,9591	0,930	0,937	0,888	0,867	0,9311
Unbounded LPT (exact cover)	0,943	0,955	0,962	0,931	0,943	0,91	0,827	0,936
100L 100P 100T	0,986	0,986	0,986	0,807	0,986	0,986	0,930	0,930
100L 100P 100T (exact cover)	1	1	1	0	1	1	0,967	0,967

User Manual and Functionalities

Main menu

GUI buttons

1. Solve button: Opens a pop-up menu with more decisions.
2. Select Parcels: Opens a menu to start selecting the parcels and their values.

Keyboard buttons

1. Up arrow: If at least 1 horizontal layer is removed, add another horizontal layer on top of the other (horizontal) layers.
2. Down arrow: If there still is a horizontal layer of the container, remove the current highest horizontal layer.
3. Left arrow: If at least 1 vertical layer is removed, add another vertical layer to the left of the other (vertical) layers.
4. Right arrow: If there still is a vertical layer of the container, remove the current most left vertical layer.
5. Tab: Toggleable button to make the empty spaces white improving the visibility.

Parcel selection menu

Text Fields

1. Amount: Insert numerical value for the quantity of a type of parcel.
2. Value: Insert numerical value for the value of a type of parcel.

GUI Button

1. Done selecting: Confirm selection of parcels and given value to the parcels. Return to the main menu.

Pop-up solve menu

CheckBoxes

5. Exact cover Algorithm: Check the box if you want to try to find an exact solution.

ComboBox

6. Choose which algorithm you want to use: Sticky Greedy, Dancing Links or Genetic Fitter.

GUI Button

7. Solve: Execute the chosen algorithm for the given amount of parcels and their corresponding values.

Text Fields

8. Limit the amount of time the algorithm has to find a solution.