Filling cargo spaces optimally

Insight into the three-dimensional knapsack problem
and three approaches to solve it

Project 1 Phase 3

Group 8:
Adam Eljasiak
Nicola Gheza
Daniel Kaestner
Raffaele Piccini
Henri Viigimäe
Simon Wengeler

Maastricht, 20th January 2016

University Maastricht
Bachelor Data Science and Knowledge Engineering

Filling cargo spaces optimally

Insight into the three-dimensional knapsack problem
and three approaches to solve it

Project 1 Phase 3
Group 8

Maastricht, 20th January 2016

Group members:
Adam Eljasiak
Nicola Gheza
Daniel Kaestner
Raffaele Piccini
Henri Viigimäe
Simon Wengeler

Project coordinator:
Jan Paredis

Project examiners:
Dr Pietro Bonizzi
Dr Evgueni Smirnov

Preface

The following report is the result of a student project that is part of the first year program of the Bachelor Knowledge Engineering at Maastricht University. As such its purpose is in part to comprise the results of said project and is meant to give insight in it for teachers and other students of the faculty.

To understand the context of the report the reader is advised to read the following summary as well as the introductory chapter. The chapters 3 and 5 are mostly of interest for those with access to the application built for this project, all other chapters mostly discuss the actual problem.

Summary

While the immediate context of the following report is a specific assignment (see 1.1), that particular problem is only discussed very briefly. Instead the point of focus is the presentation of different approaches for solving three-dimensional knapsack problems, e.g. the optimisation of packing cargo in a restricted space.

Three algorithms were implemented to solve the knapsack problem given by the project assignment: a greedy approximation algorithm, a hill-climbing algorithm and a genetic algorithm. The idea and implementation behind all three algorithms is mostly based on knowledge acquired previously in the study program Knowledge Engineering. The main purpose of the later described experiments is to test which factors of the algorithms have an impact on their performance and under which conditions that performance is optimal. Furthermore, the experiments serve as a way to determine the overall best of the approaches to the knapsack problem for practical purposes.

To summarise the results, it became clear that the genetic algorithm performs best out of the tested algorithms under the given conditions. It finds a very good solution (presumably the best) in a reasonable amount of time. However, its performance in terms of computation time lacks significantly for different conditions, especially a larger number of packages. Furthermore, it is restricted to finding near-optimal solutions for rectangular packages only. A hill-climbing algorithm has been found to be a good and very fast alternative.

Table of contents

1. Introduction

1.1 Assignment description

The assignment for the project was to build a computer application with a user friendly interface that can be used for solving so-called three dimensional knapsack problems.

The assumptions are that a company owns trucks with a cargo space of 16.5 m long, 2.5 m wide and 4.0 m high and that it transports parcels of three different types: A, B and C. The sizes of the types are:

A: 1.0 x 1.0 x 2.0
B: 1.0 x 1.5 x 2.0
c: 1.5 x 1.5 x1.5

A parcel of a given type also has a certain value, denoted by v_A, v_B and v_C for types A, B and C respectively. The computer application should compute, for a given set of parcels (that may or may not fit into a truck), a packing that maximises the total value.

The application does not have to find the best answer in all cases, but it should be able to find a good approximation. The application should also be able to present a 3D-visualisation of its answers from different perspectives and should be used to answer the following questions (see 3.1 and 3.2):

a. Is it possible to fill the complete cargo space with A, B and/or C parcels, without having any gaps?
b. If parcels of type A,B and C represent values of 3, 4 and 5 units respectively, then what is the maximum value that can be stored in the cargo space?

In addition, after answering the two previous questions, it should be assumed that the company transports pentomino shaped parcels of types L, P and T (see Appendix A, Figure 1), where each of these pentominoes consists of 5 cubes of size 0.5 x 0.5 x 0.5. On the basis of those assumptions the following questions were posed:

c. Is it possible to fill the complete cargo space with L, P and/or T parcels without having any gaps?
d. If parcels of type L, P and T represent values of 3, 4 and 5 units respectively, then what is the maximum value that can be stored in the cargo space?

1.2 Problem definition

The main purpose of the application is to devise an algorithm (or multiple) that maximises the total value of the solution while fitting all the packages within the given space without overlapping, defined as a three-dimensional knapsack problem.

While similar kinds of optimisation or knapsack problems can occur in a wide variety of fields and similar algorithmic approaches to the ones chosen for the purpose of this project may be applicable, this project focuses on the packing of a three-dimensional space. Consequently the algorithms developed during the research are optimised to fill a cargo space of a truck or any similar sort of container.

1.3 Structure

Chapter 2 of the report describes the three algorithmic approaches to solve the assigned problem (a greedy approximation algorithm, a hill climbing and a genetic algorithm. In chapter 3 the newly conceived graphical representation of the solution is explained. An insight into the implementation of the algorithms in the program is given in chapter 4 and chapter 5 contains a system guide for the application built for the project. Chapter 6 gives concise answers to the four individual questions posed by the project assignment (see 1.1) without going into a lot of detail regarding the implication of the results. In chapter 7 several experiments are described in which certain parameters crucial for the performance of the three chosen algorithms are varied, including their results. Lastly, in chapter 8, conclusions are drawn from the previously described results of the experiments.

2. Algorithms for the knapsack problem

2.1 Greedy algorithm

While not in the form of a three-dimensional knapsack problem, such as the one that is subject of this project report, the idea of a so called greedy approximation algorithm originates from the American mathematical scientist George Dantzig (1957). In his version of the algorithm the items (in this case packages) to be placed in the knapsack are sorted by their value per weight (which is the volume for this problem) and then placed in the knapsack in the resulting sequence.

The algorithm was chosen mainly due to the fact that it works with very simple heuristics to decide the ordering of packages that are placed. Thus the method to select an ordering can be implemented and changed quite easily. Furthermore, a simple placement method can be used for the algorithm (see chapter 4.1) that is also applicable for the genetic algorithm.

2.2 Hill-climbing algorithm

A different optimisation algorithm for the application is a hill-climbing algorithm. It is a relatively simple algorithm that is easy to implement and can achieve decent results. Despite the drawback of the algorithm to only find local optima and not the global optimum and therefore the optimal solution, it was expected to give better results than the greedy algorithm.

2.3 Genetic algorithm

The third algorithm implemented in the program is a genetic algorithm. It evolves chromosomes, encoded in an appropriate manner, that represent a solution to the problem. The encoding for the genetic algorithm implemented for this project is one proposed by Lawrence Davis in 1985. In order to solve a two-dimensional bin packing problem using a genetic algorithm, according to him "the [chromosome representation] that worked best was a simple list of rectangles to be packed. [A] decoding algorithm proceeded by placing the first member of the list into the first place it would fit in the bin […] and so forth"[1] until the chromosome was interpreted in its entirety.

Using such an encoding of the chromosomes is very easy to implement and enables the use of a simple decoding mechanism as well. Given the usually good performance of genetic algorithms and the complexity of the problem, implementing one was considered the most promising approach to the given problem. Additionally, some consideration went into the use of similar representations of individuals for the hill-climbing and the genetic algorithm. This could have reduced the amount of code needed overall. However, the finished product uses different approaches.

[1] Davis, Lawrence (1985). Applying Adaptive Algorithms to Epistatic Domains. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp.162-164). Los Altos, CA: Morgan Kaufmann Publishers, Inc.

3. Graphical Representation in 3D

3.1 Creating 3D cubes

In order to represent the cargo space only polygons utilising the 2D graphics classes are used. For the creation of the three-dimensional representation of the cargo space an array of packages containing x-, y- and z-position as well as the length, width and height of every package have to be provided. Given this data the program is able to calculate the eight corner points of every package. With this information it creates a cube consisting of six polygons for every face of the package that can be drawn from different perspective.

3.2 Perspective

To avoid drawing certain packages incorrectly in front of others the cubes have to be drawn in a specific order. This order is determined by the cubes' depth in the cargo space (looking from the front), where the cube with the "deepest" position is drawn first. This it is overdrawn by cubes that should appear closer to the viewer because they are not positioned as deep in the cargo space. In addition, since it is only possible to see at most three faces of a cube, the same sorting principle is applied on the cubes and only the three front faces are drawn.

3.3 Rotation

To rotate cubes around a specific angle and axis, rotation matrices in the third dimension are used. These matrices are applied on every corner of each package and the centre-point of a cube's position to achieve a realistic rotation.

3.4 Implementation

The classes used for the implementation of the three-dimensional graphical representation of solutions can be seen in Figure 1.

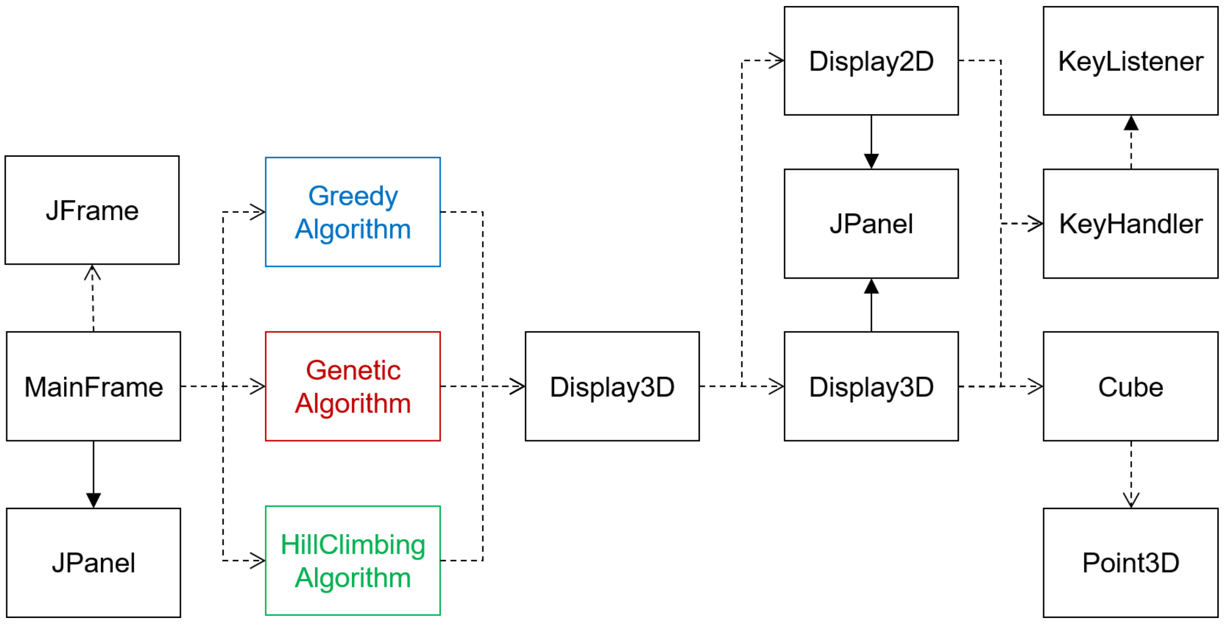


Figure 1 - UML diagram of the classes used by the GUI

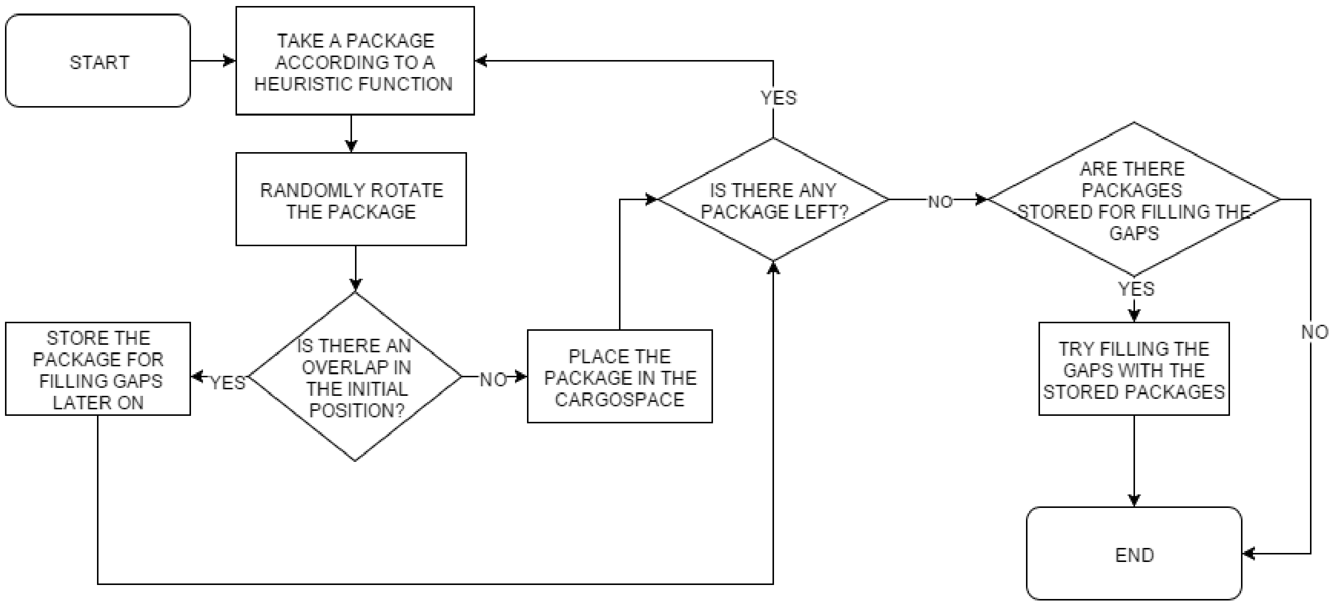
4. Implementation of the algorithms

All of the implemented algorithms use a three-dimensional array to represent the cargo space and determine whether packages can be placed in certain positions. The array's dimensions are twice those of the actual cargo space to compensate for the fact that the latter's dimensions cannot all be expressed in integers.

Package objects contain information about the coordinates that they occupy relative to a set of coordinates belonging to a base point (lower left back-most corner).

4.1 Greedy algorithm

The same principle as the one described in 2.1 is applied in the greedy approximation algorithm. From the packages that are chosen to be placed in the cargo space the ones with the highest value to volume ratio are placed first as long as there is a supply of them. When the supply of packages of the first type is exhausted and there is empty space left, the next type of package will be placed. That process is repeated until all packages have been placed or none of the packages left can be placed anymore. (UML diagram in Appendix A, Figure 9)


START
TAKE A PACKAGE ACCORDING TO A HEURISTIC FUNCTION
RANDOMLY ROTATE THE PACKAGE
IS THERE AN OVERLAP IN THE INITIAL POSITION?
YES
STORE THE PACKAGE FOR FILLING GAPS LATER ON
NO
PLACE THE PACKAGE IN THE CARGOSPACE
IS THERE ANY PACKAGE LEFT?
YES
NO
ARE THERE PACKAGES STORED FOR FILLING THE GAPS
YES
TRY FILLING THE GAPS WITH THE STORED PACKAGES
NO
END

Flowchart 1 - Functionality of the greedy algorithm

The placement method employed in the application (both in the greedy and the genetic algorithm) tries to place a new package in the top right front corner of the cargo space and moves on in the sequence if it cannot be placed. From that initial position the package is first moved as far back, then as far left and finally as far down in the cargo space as possible (corresponding to movements along the y-axis, x-axis and z-axis, see Figure 2). Additionally the algorithm will then test whether the package can still be moved in any of the three directions listed above.

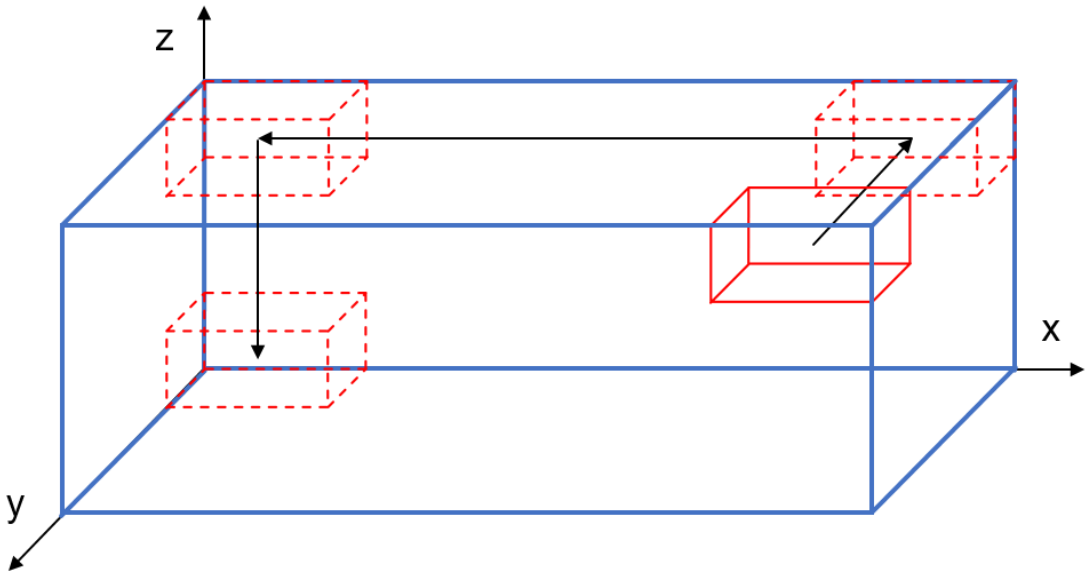
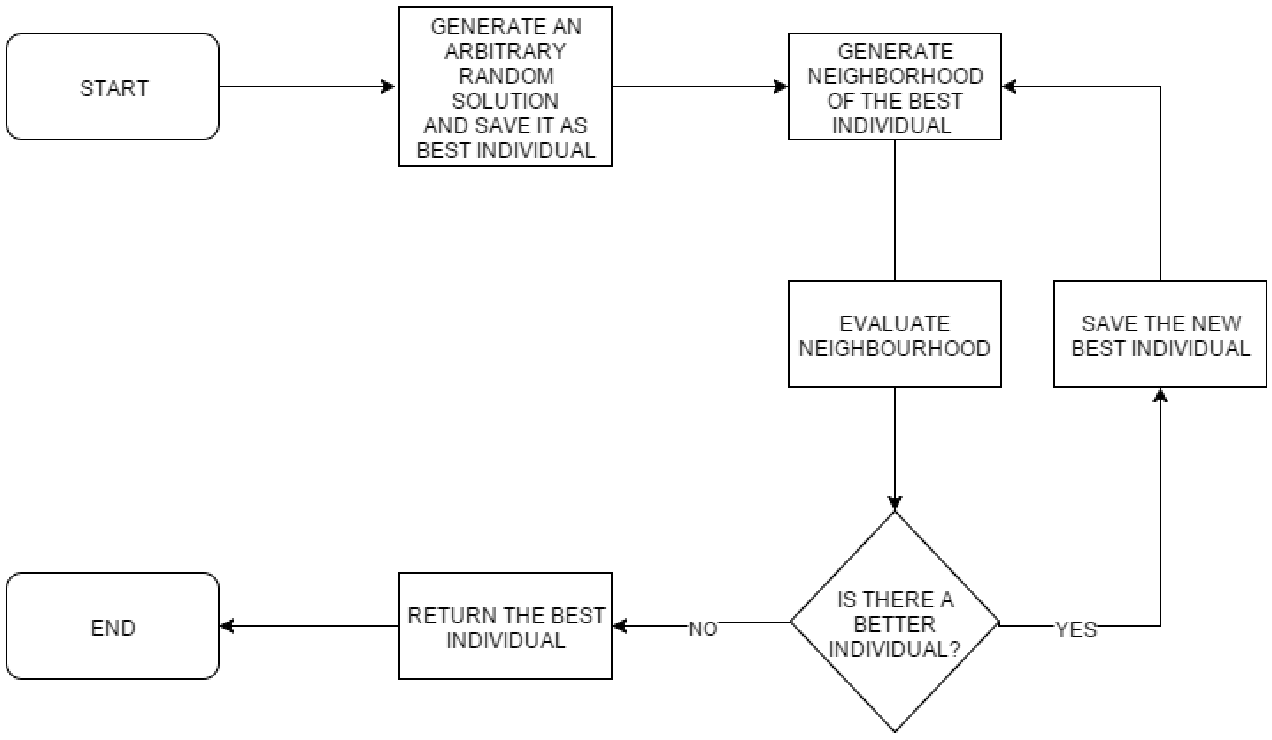


Figure 2 - Placement mechanism for the greedy and the genetic algorithm

4.2 Hill-climbing algorithm

The hill-climbing algorithm uses a randomly generated solution (i.e. a cargo space filled with random packages at random positions) as a starting point. It then searches in the neighbourhood of that solution (created by removing a number of packages and filling the gaps again) for one that has a higher value. It repeats that process until it cannot "climb" any more, i.e. no better solution can be found. (UML diagram in Appendix A, Figure 10)


START
GENERATE AN ARBITRARY RANDOM SOLUTION AND SAVE IT AS BEST INDIVIDUAL
GENERATE NEIGHBORHOOD OF THE BEST INDIVIDUAL
EVALUATE NEIGHBOURHOOD
SAVE THE NEW BEST INDIVIDUAL
IS THERE A BETTER INDIVIDUAL?
YES
NO
RETURN THE BEST INDIVIDUAL
END

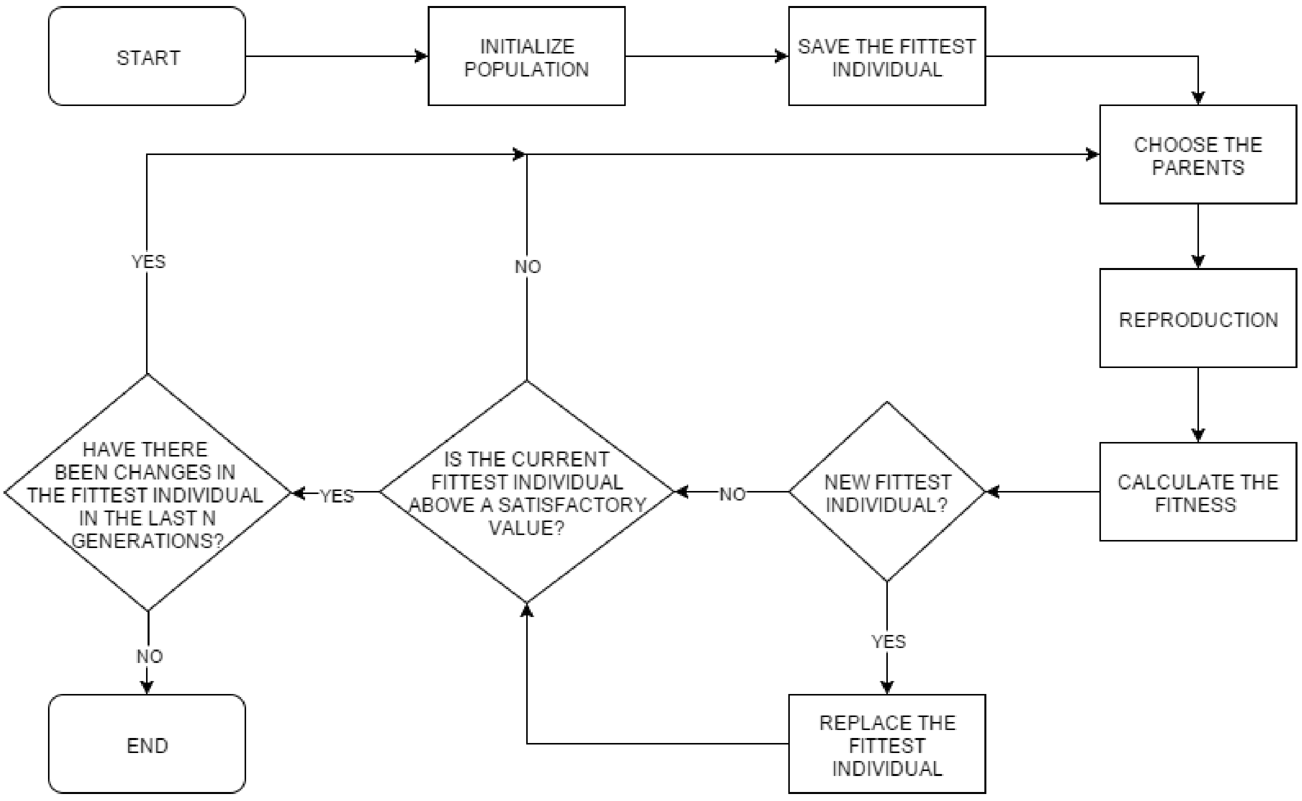
Flowchart 2 - Functionality of the hill-climbing algorithm

4.3 Genetic algorithm

As described in chapter 2.3 a chromosome encoding was chosen for the genetic algorithm which consists only of packages. The single packages or ”genes” should be placed in the same order as they occur in the genetic code. The same placement scheme as the one used for the greedy approximation algorithm (see Figure 3) is employed to decode the chromosomes.

The genetic algorithm uses a “modified crossover”[2] in order to retain a favourable order of the packages as well as ensure that only the designated amount of packages of each type are placed. The fitness of each individual is determined by the value that the decoded chromosome amounts to (as a filled cargo space). While the default mutation method is the switching of “genes” (i.e. changing the placement order of the packages), genes can optionally be altered by setting a different state of rotation for the package they represent.

For the purpose of experimenting with the effects of changing vital methods and parameters on the performance of the genetic algorithm, three different selection methods have been implemented in the program. Furthermore it is possible to change the crossover frequency (during reproduction/combination of chromosomes), the frequency of mutation for both variants of it as well as parameters specific to the selection methods. (UML diagram in Appendix A, Figure 11)


START
INITIALIZE POPULATION
SAVE THE FITTEST INDIVIDUAL
CHOOSE THE PARENTS
REPRODUCTION
CALCULATE THE FITNESS
NEW FITTEST INDIVIDUAL?
NO
YES
REPLACE THE FITTEST INDIVIDUAL
IS THE CURRENT FITTEST INDIVIDUAL ABOVE A SATISFACTORY VALUE?
NO
YES
HAVE THERE BEEN CHANGES IN THE FITTEST INDIVIDUAL IN THE LAST N GENERATIONS?
YES
NO
END

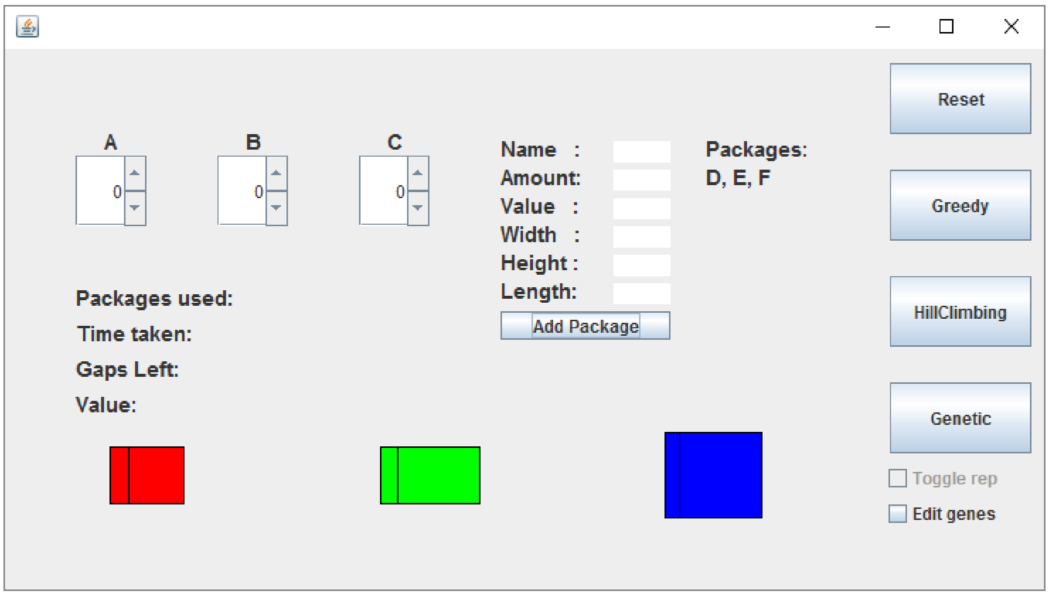
Flowchart 3 - Functionality of the genetic algorithm

[2] Davis, Lawrence (1985). Applying Adaptive Algorithms to Epistatic Domains. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp.162-164). Los Altos, CA: Morgan Kaufmann Publishers, Inc.

5. System Guide

The user interface of the application that was built for the project is shown in the following images. Its functionality includes the addition of the package types A, B and C, which are predefined within the Package class. Additionally, new package types can be added by specifying a name, the amount that should be used for finding a solution, the value, and all three dimensions.

After adding the desired amounts and types of packages, the user can choose to run either of the three implemented algorithms using the buttons on the right. Once the solution has been computed and is displayed in a new frame, the three-dimensional graphical representation can be toggled off (instead showing a two-dimensional one). Furthermore, some statistics about the algorithms performance are displayed.



A: 0
B: 0
C: 0

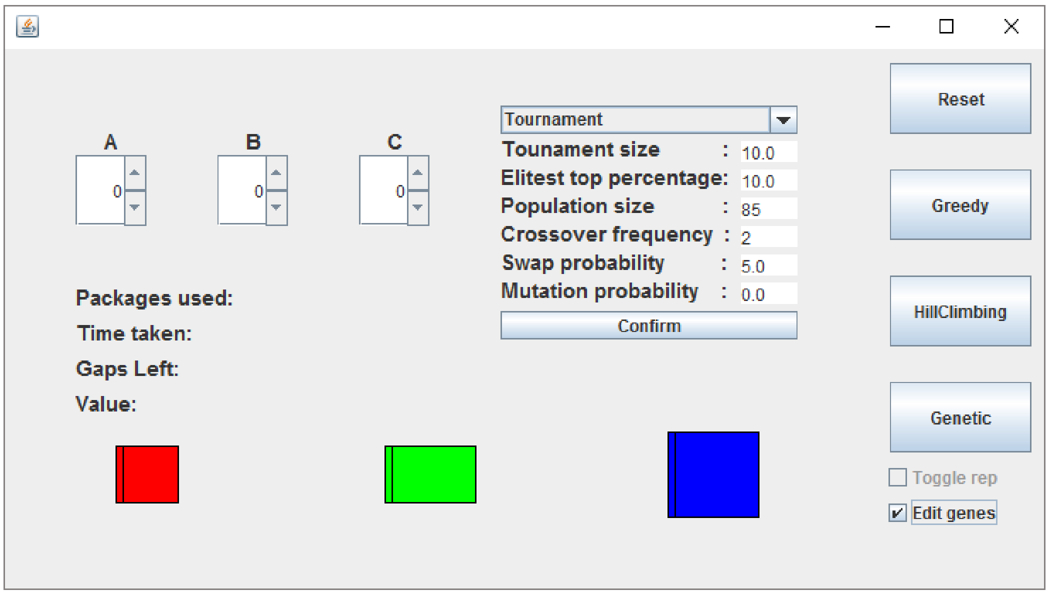
Name :
Amount:
Value :
Width :
Height :
Length:

Packages:
D, E, F

Packages used:
Time taken:
Gaps Left:
Value:

Figure 3 - User interface: Default view with three new package types

Toggling the checkbox labelled “Edit genes” lets the user set the parameters of the genetic algorithm. This includes the selection method and the parameters shown in Figure 4.



A: 0
B: 0
C: 0

Tournament

Tounament size :	10.0
Elitest top percentage:	10.0
Population size :	85
Crossover frequency :	2
Swap probability :	5.0
Mutation probability :	0.0

[✓ Edit genes]

Packages used:
Time taken:
Gaps Left:
Value:

Figure 4 - User interface: View with GA settings enabled

6. Assignment results

6.1 Using rectangular packages

The best result in regard to filling the entire cargo space was obtained by the genetic algorithm. It was not able to completely fill the cargo space but only left a single gap with the dimensions 1.5m x 1m x 0.5m. That result was achieved using tournament selection with the following settings: $\mathrm{S}_T = 0.1 * \mathrm{P},\ \mathrm{P} = 85,\ \mathrm{C}_F = 2,\ \mathrm{M}_S = 0.05,\ \mathrm{M}_C = 0$. Since the genetic algorithm was the best performing algorithm that was implemented and repeatedly managed to leave only that small gap empty but never any less it can be concluded that it is not possible to fill the entire cargo space with only the packages A, B and/or C.

As for the best result achieved by any of the algorithms maximising the value of a packing, the genetic algorithm achieved a value of 240 using the same settings as those described above. Both maximising the occupied space and the value resulted in the same packing.

6.2 Using pentomino-shaped packages

Using a brute-force backtracking algorithm a solution could be found in which the entire cargo space is filled with pentominoes. The cargo space is filled with eight layers of pentominoes measuring 16.5m x 2.5m x 0.5m or alternatively 33 layers measuring 0.5m x 2.5m x 4m. Only T and P pentominoes are used.

The highest value achieved using the same algorithm (and the same ordering which could fill the entire cargo space) was 1144.

7. Experiments and results

7.1 Principles of evaluation

7.1.1 Measures of performance

The most important factor taken into consideration during the evaluation of an algorithm's performance is its ability to maximise the value of a packing. Additionally, the amount of time it takes the algorithm to compute a solution is considered as well since it has relevance in terms of practicability as an application for actual use. It also gives some insight into the complexity/amount of computations that the algorithm requires to find a solution.

7.1.2 Comparison between results

In order to be able to compare individual results all factors that might play into the performance of the algorithms other than the one which is being tested were kept constant. This applies both to the tests done on a single algorithm, e.g. changing the population size for the genetic algorithm, as well as to the comparisons between multiple algorithms (as far as parameters could be kept constant).

It should be noted that the number of packages of each available type is considered "infinitely large" for the purpose of the test if there are just enough to fill the entire volume of the cargo space with only one type. The same applies to the packages that are used, which are usually the standard A, B and C packages. Packages with different notations and dimensions will be specified as such.

7.1.3 Other notes

It should be noted that in all of the following figures the connections between data points is for clarity and should not necessarily imply that the data gathered contained the additional implied information. Furthermore, all graphs display both the average value achieved by the algorithm (on the left, blue-coloured y-axis) as well as its average runtime (on the right, orange-coloured y-axis).

7.2 Greedy algorithm

The algorithm's performance depends on a couple of parameters that can be switched on or off. In spite of the fact that in some cases some of them disrupted achieving the highest score, it eventually turned out that each parameter is crucial in overall performance.

When it comes to random rotation method, better results were achieved when it was not used in the case of using rectangular packages. This method as well as one that randomly orders the packages for placement are undesired when there is prepared ordering, which will provide compact composition.

The average score that was achieved among the tests using packages A, B and C is 154.23. It turned out that the greedy algorithm performs the best using small and compact parcels (shaped closest to a cube). Moreover, the tests show that the best solutions were computed quickly, whereas non-optimal solutions took the most time. When it comes to cargo space size, doubling each dimension of given space will prolong runtime by approximately 25 times.

7.3 Hill-climbing algorithm

All experiments were conducted under the same conditions. For all of the tests, the algorithm was run 100 times using a i7 processor with a clock-speed of 2,9 GHz.

7.3.1 Varying neighbourhoods and mutation rates

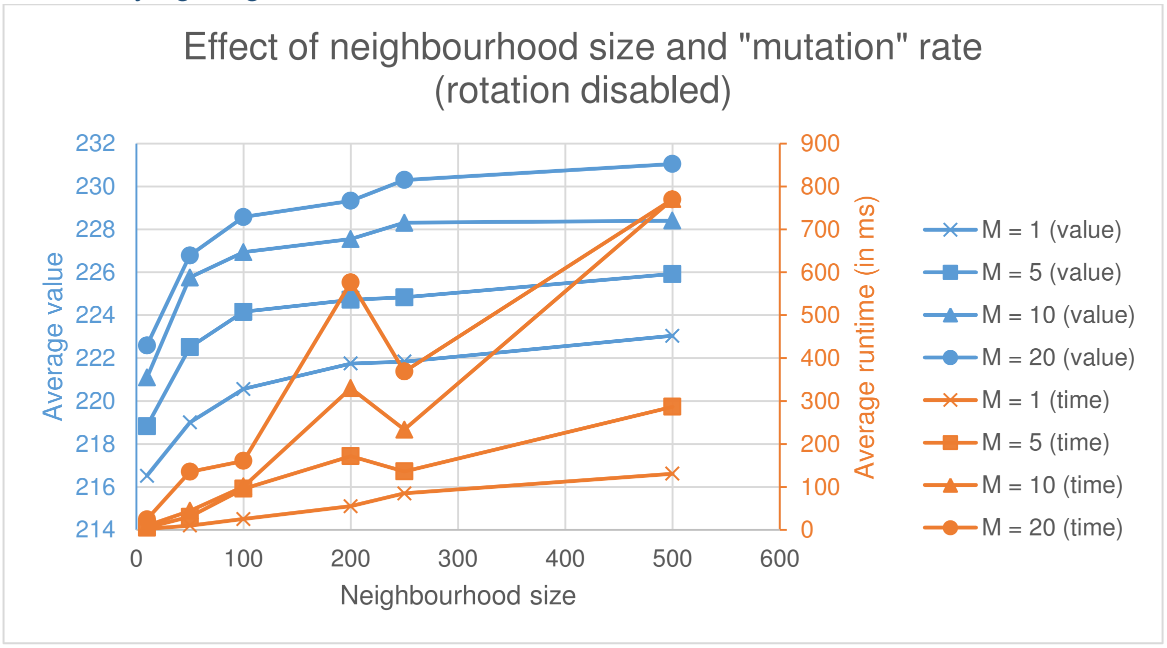


Figure 5 - Effect of neighbourhood size and mutation rate on the performance of the hill-climbing algorithm

As can be seen in Figure 5 increasing mutation rates can be attributed to an improved performance of the algorithm in terms of increasing the value of the solutions. A higher neighbourhood size has a similar effect, due to the fact that the higher diversity makes it more likely to find a good solution or solutions that can be improved further.

7.3.2 Rotation

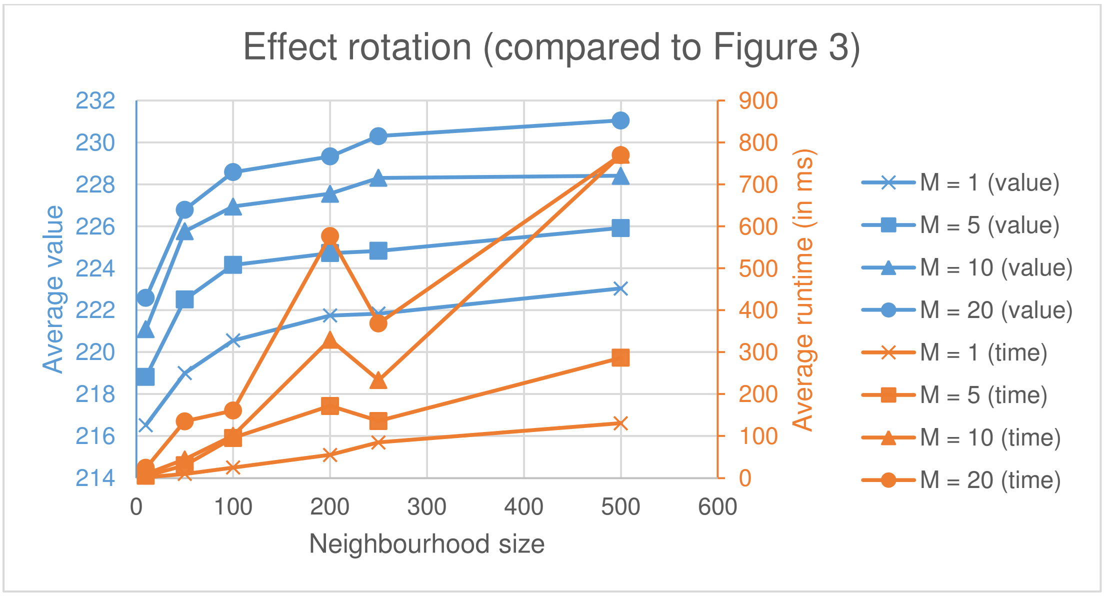


Figure 6 - Effect of allowing rotation of packages on the performance of the hill-climbing algorithm

With rotation of packages enabled the algorithm doesn't perform as well as without rotation. The same effects of increasing neighbourhood size and mutation rate can still be observed though.

7.4 Genetic algorithm

The selection of individuals that are allowed to reproduce can have an effect on a genetic algorithm's performance. The three selection methods tested in this project are elitist (EL), tournament (TS) and roulette selection (RO). The first chooses only individuals out of the fittest in the population, the second creates a "tournament" (a part of the population) and chooses the fittest individuals for reproduction and the third assigns certain probabilities to each individual to be selected (higher for fitter individuals).

In the following experiments all tests have been conducted for all three selection methods under identical conditions (as far as it was possible to do so). They were performed using an i5-4690k processor with 3.5GHz clock speed. The average values have been calculated from the results gathered over 25 runs of the algorithm. If not specified otherwise, the following parameters were used: $S_T/S_E = 0.1 * \mathrm{P},\ \mathrm{P} = 85,\ \mathrm{C}_F = 2,\ \mathrm{M}_s = 0.05,\ \mathrm{M}_C = 0.$

7.4.1 Population size

For any genetic algorithm the population size can have a dramatic effect on its performance. The larger genetic diversity can vastly decrease the number of generations it takes the algorithm to find a solution (as long as there is a definitive solution) or improve the solution given a specific number of generations to run for. At the same time there may be an increase in computation time due to the need to deal with combining, mutating and evaluating more individuals.

As can be seen in Figure 7 with increasing population size the performance of all algorithms in terms of the average value increases. The increase is especially large for the tournament selection method due to the fact that selecting a tournament from a small population (or selecting a very small tournament) results in (almost) random selection. Further proof for the importance of a large enough pool of individuals to choose from is the significant decrease of computation time for the tournament selection method. Since the algorithm either reaches a satisfactory result (in this case a value larger than 230) or runs for the full 1500 generations, this drop-off signifies a much quicker convergence on a good result.

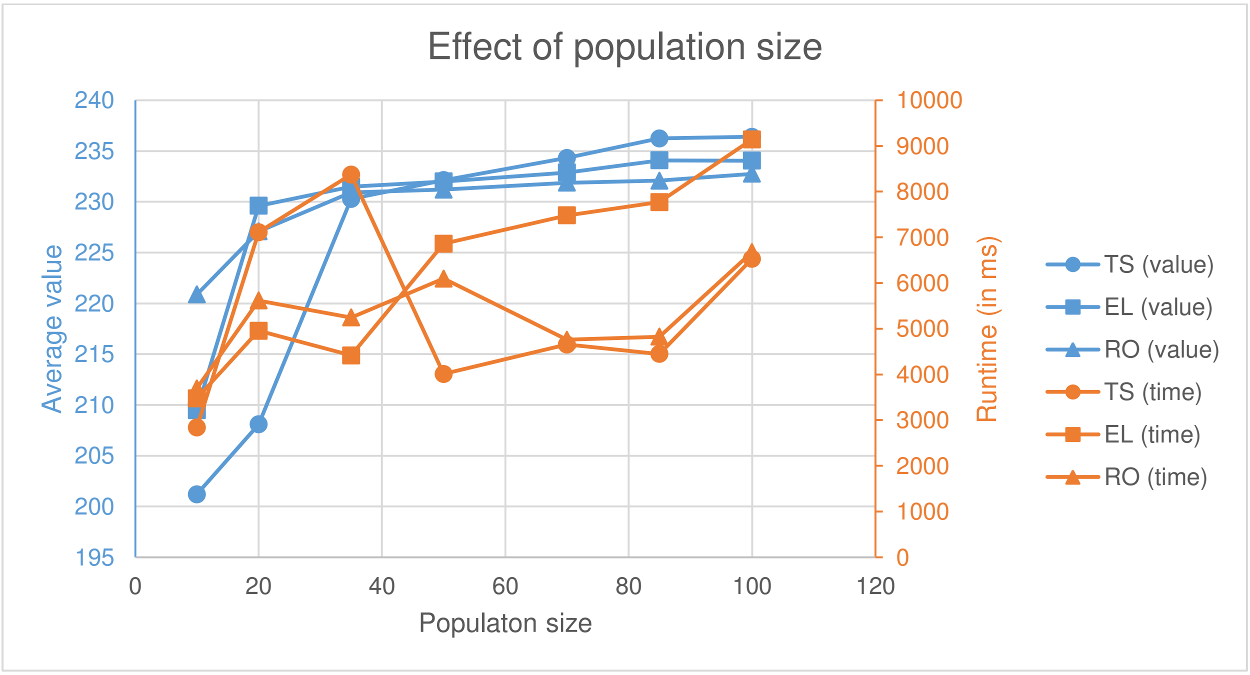


Figure 7 – Effect of population size on the GA's performance (Appendix A, Table 2)

Other experiments performed using more and different types of packages (instead of only 83 A, 55 B and 50 C) saw a significant increase in runtime and a decrease in performance. The latter could be alleviated to some degree by increasing the population size, even though runtimes remained high. This shows the importance of a large genetic diversity if there are more possibilities to order different kinds of packages due to more package diversity. Furthermore, the high runtimes of more than 30 and often over 60 seconds show one of the major disadvantages of the algorithm.

7.4.2 Mutation

As described in 2.3 the mutation method used for the genetic algorithm is one that only swaps single genes. As became apparent in the conducted experiments this is crucial for changing the chromosomes between generations enough to produce sufficiently different individuals (see also Appendix A, Figure 12). Beyond that, the special "modified crossover" implemented in the algorithm causes an effect that goes beyond copying parts of the parents' chromosomes and thus "mutates" the children's chromosomes. Interestingly enough additionally mutating genes by changing the package's rotation state results in noticeably worse performance (see Figure 8).

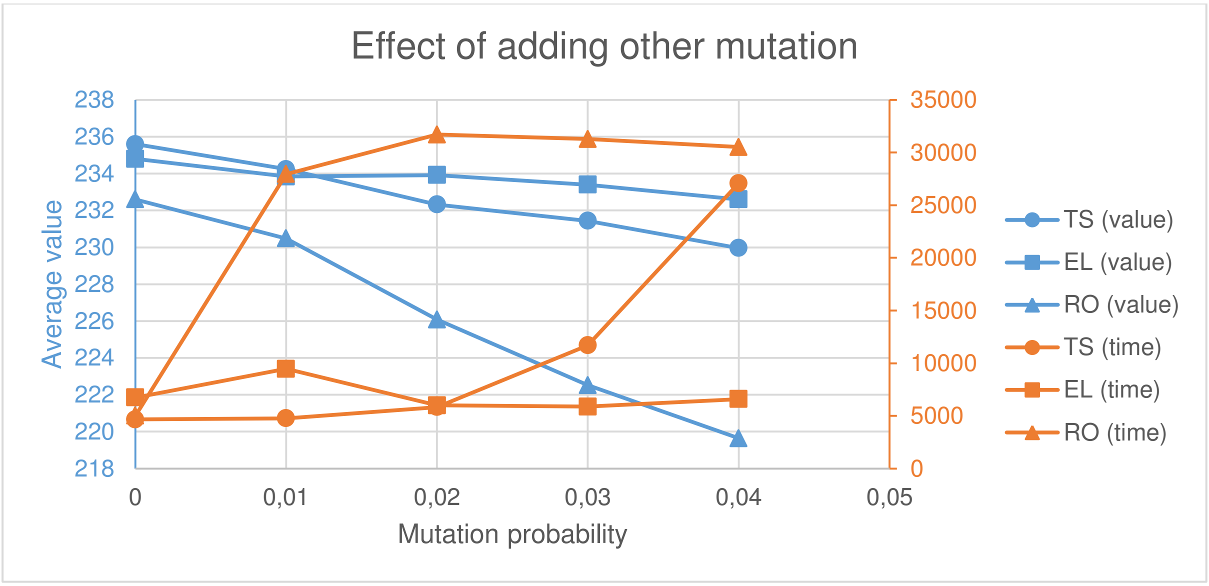


Figure 8 - Effect of adding other mutation on the GA's performance (Appendix A, Table 3)

This result may be linked to the tendency that the best solutions achieved by the algorithms are usually quite strictly sorted packings of the cargo. Packages of the same type usually occupy continuous areas of the cargo space in which they are oriented the same way. In that case rotating a package that would otherwise have been aligned with all of the other packages of its type can reduce the fitness of the resulting individual.

7.4.3 External factors

Aside from the internal parameters that have an influence on the algorithms performance, external factors can affect it as well. Several experiments were conducted using different numbers of packages as well as new package types with different dimensions but all with a value of 1. The results of those experiments can be found in Table 1 (where the defining property of each test is described, for the dimensions of the packages used for each test see List 1). The number of packages used is given in brackets (inf = infinite for the purpose of the test).

Defining properties	V	T (ms)	G	% 1D	Test No.
Small packages (inf)	289.26	12479.16	50	0.877	1
Big packages (inf)	10	61883.96	123	0.909	2
Cubes (inf)	108.56	45626.04	116	0.658	3
Long packages (inf)	248.6	85670.68	29.72	0.942	4
Flat packages (inf)	130.68	81555.08	121.44	0.895	5
A, B, C + 1 similar (15)	186.24	33130.76	165.52	0.955	6
A, B, C + 2 similar (15)	187.6	39986.68	163.8	0.947	7
A, B, C + 3 similar (10)	140.04	37154.92	220.4	0.946	8
A, B, C + 1 similar (inf)	213.36	51289.88	80.2	0.864	9
A, B, C + 2 similar (inf)	201.36	72838.04	66.56	0.815	10
A, B, C + 3 similar (inf)	204.12	73388.08	74.24	0.826	11

Table 1 - Experiments for the GA involving different packages (V = average total value, T = average runtime, G = average gaps left, %1D = percentage of the value achieved for the equivalent 1D problem, taking only volume into account)

1	1x1x1 (330), 0.5x1x1.5 (220), 1x1x1 (165)
2	2x3x3 (10), 2x2.5x3 (11), 2.5x2.5x2.5 (11)
3	1x1x1 (165), 1.5x1.5x1.5 (50), 2x2x2 (21)
4	2.5x0.5x0.5 (264), 3x1x1 (55), 4x1x0.5 (83)
5	2x2.5x0.5 (66), 1.5x1.5x0.5 (147), 3x2x1 (28)
6	A (15), B (15), C (15), 2.x2.05 (15)
7	A (15), B (15), C (15), 2.x2.05 (15), 3x1x1 (15)
8	A (10), B (10), C (10), 2.x2.05 (10), 3x1x1 (10), 2.5x1x1.5 (10)
9	A (83), B (55), C (50), 2.x2.05 (83)
10	A (83), B (55), C (50), 2.x2.05 (83), 3x1x1 (55)
11	A (83), B (55), C (50), 2.x2.05 (83), 3x1x1 (55), 2.5x1x1.5 (44)

List 1 - Packages used in experiments for the GA [length x width x height (amount)]

The general conclusion that can be drawn from these experiments is that the genetic algorithm still performs quite well in terms of achieving a high value in almost all of them (in comparison to the theoretically achievable value in 1D), meaning that a correlation between the performance of the algorithm and certain package shapes could not be found.

Similar to earlier experiments (see 7.5.1), the significantly lower runtimes in tests 6-8 compared to tests 9-11 show the influence of chromosome length on the algorithm's performance. While the last three tests do not actually use an infinite amount of packages, they still use up to 182 more.

The comparatively low value in test 3 is likely due to the fact that the algorithm does rarely only place packages of one only type in the time it is given to run (1500 generations). But since the highest theoretically achievable value can best be achieved by only including packages of the first type (since they have the highest value per volume), the value is lowered quite a bit. For example three packages of the first type could fit into one of the second type only taking volume into consideration, which would reduce the value by 2 since all packages have values of 1.

8. Conclusions

8.1 Greedy algorithm

While the algorithm is very easy to implement and has faster computation times than the genetic algorithm, it does not achieve very high scores. It performs decently well using cubic packages but falls short otherwise.

8.2 Hill-climbing algorithm

The hill-climbing algorithm performs quite well, especially in terms of computation time. It often finds solutions of over 85% of the theoretically achievable value using A, B and C packages in under a single second. Thus its performance is even better than the very simply and suboptimal greedy algorithm.

8.3 Genetic algorithm

The performance of the genetic algorithm for the specific task of filling the cargo space given by the project assignment with A, B and C packages is the best out of the three algorithms. Using the standard parameters described in 7.4 it finds a very good solution to the specific knapsack problem in a reasonably short amount of time.

However, the conducted experiments have shown that the algorithm (with its current parameters) shows worse performance once certain external parameters change. Especially when package numbers increase the computation time increases drastically as well and the achieved values become less optimal (although still quite good).

8.4 Comparison between algorithms

Given the results of the experiments described in the previous chapter, if performance is strictly measured by reliably achieving high values, the genetic algorithm is best suited to the task without a doubt. Even in the experiments described in 7.4.3 it achieves very good results, despite the fact that the parameters that were used in those tests were tuned for a good performance using packages of type A, B and C.

Using the hill-climbing algorithm it has also become clear that it may have better applicability in some cases than the genetic algorithm. While not quite as optimal it still achieves good results in very low computation times. Depending on whether fast computation times are preferred over the most optimal performance or not, using the hill-climbing algorithm can still be recommended for achieving good solutions. However, it should also be noted that changing the internal parameters of the genetic algorithm often resulted in an improved performance in regards to computation time. Therefore adjusting them to different problems/different package types might yield even better values of the packings in reasonably short amounts of time.

Appendix A: Additional material

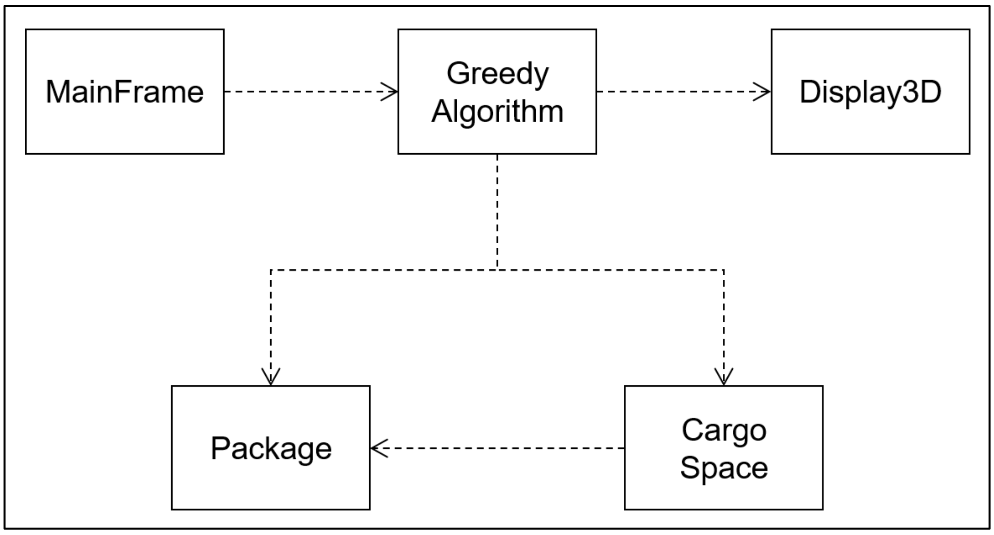


Figure 9 - UML diagram for the greedy algorithm

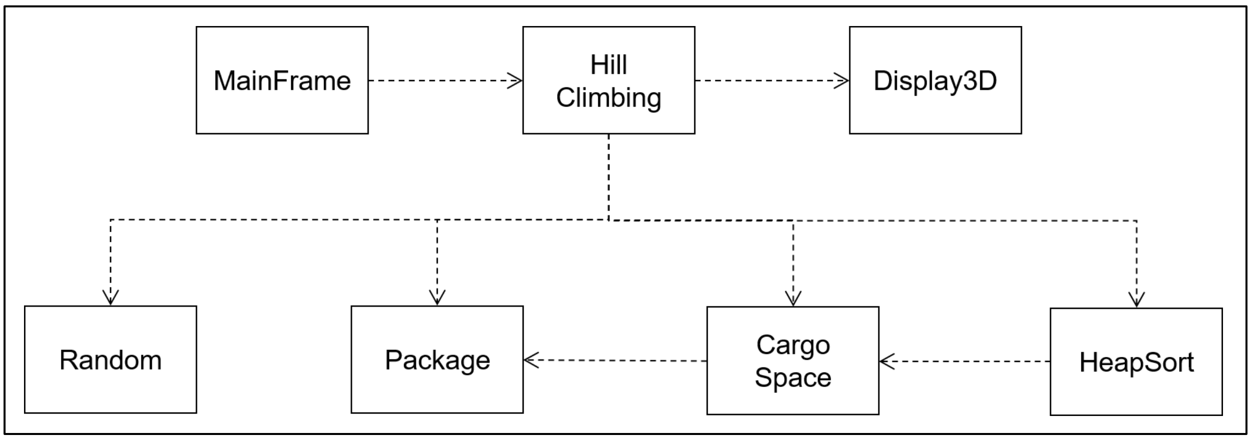


Figure 10 - UML diagram for the hill-climbing algorithm

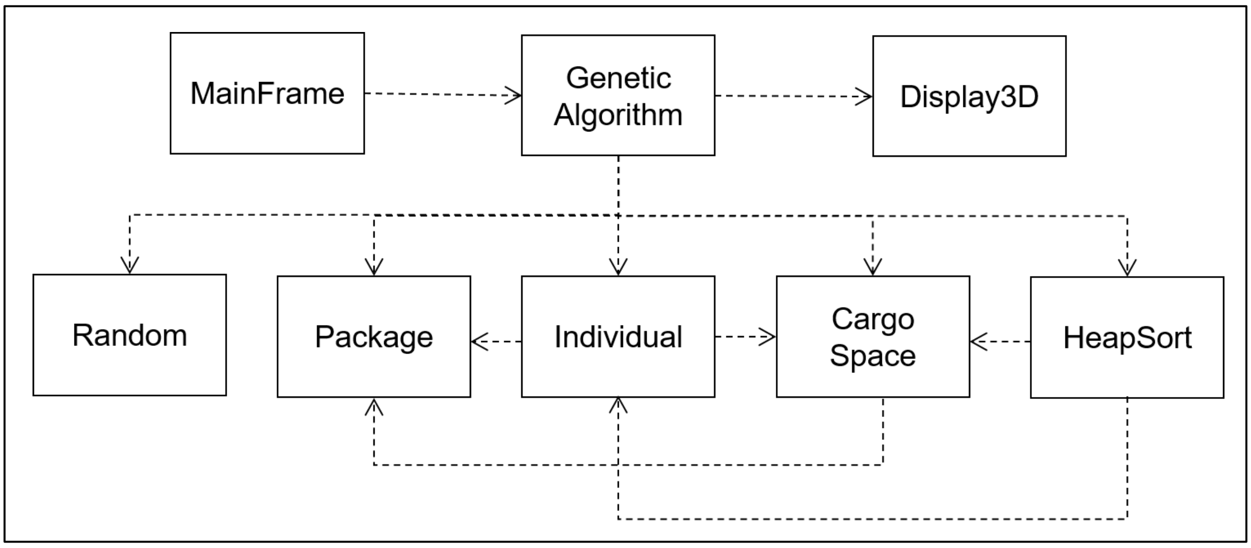


Figure 11 - UML diagram for the hill-climbing algorithm

P	V	T (ms)	G	Average A	Average B	Average C	% 1D
10	201.2	2837.72	214.76	34.84	10.72	10.76	0.818
20	208.08	7112.32	180.48	32.88	9.36	14.4	0.846
35	230.28	8371.24	63.52	35.48	8.36	18.08	0.936
50	232.12	4008.56	52.72	34.52	8.64	18.8	0.944
70	234.32	4652.84	41.4	33	8.28	20.44	0.953
85	236.24	4446.8	30.68	32.68	8.4	20.92	0.960
100	236.4	6531.96	28.4	31.84	8.92	21.04	0.961

Table 2 – Effect of population size on the GA's performance (P = population size, V = average total value, T = average runtime, G = average gaps left, %1D = percentage of the value achieved for the equivalent 1D problem, taking only volume into account)

CrF	V	T (ms)	G	Average A	Average B	Average C	% 1D
0	226.84	31417.76	78.88	30.44	9.28	19.68	0.922
1	226.2	31804.6	83.08	30.8	9	19.56	0.920
2	236.52	5750.12	28.92	32.4	8.48	21.08	0.961
3	236.76	4583.4	29.12	32.68	7.88	21.44	0.962
4	237.84	4510.4	22.48	32	8.16	21.84	0.967
5	237.32	4475.2	25.64	32.32	8.04	21.64	0.965

Table 3 – Effect of different numbers of crossover points on the GA's performance (CrF = crossover frequency, V = average total value, T = average runtime, G = average gaps left, %1D = percentage of the value achieved for the equivalent 1D problem, taking only volume into account)

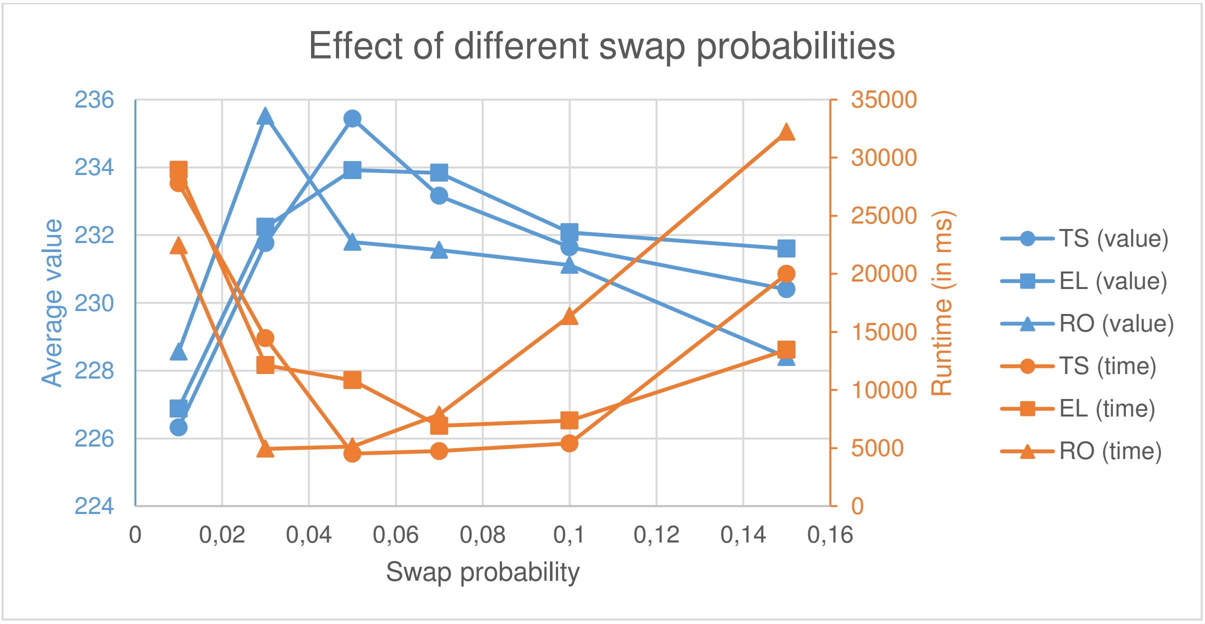


Figure 12 - Effect of different swap probabilities on the GA's performance