A multi-algorithm approach to solving the single container loading problem

24 January 2018

Maastricht University

Department of Data Science and Knowledge Engineering

Period 3 - Block 1.3 - Project Group 9

Samuel Kopp, Jordan Peshev, Barbara Futyma, Blazej Dolicki, Yvar Hulshof and Stijn van den Berg

Project coordinator: Jan Paredis

PREFACE

This project report attempts to bring under one cover the entire hard work and dedication put in by our group in the completion of the project on Knapsack problems.

This report can be used to better understand and help solving a three dimensional knapsack problem and is meant for the project examiners at DKE.

SUMMARY

In this report several algorithms are proposed to solve the single container loading problem, in which the objectives are to either completely fill the container or achieve the highest total value of packed parcels. A backtracking and Divide & Conquer algorithm were implemented, tested and compared using different variations of given amounts of parcels, container dimensions and running time. A maximal layer filling algorithm was investigated and will be explained in the report, but has not been fully implemented.

TABLE OF CONTENTS

TERMS, ABBREVIATIONS AND SYMBOLS

To avoid fractions a coordinate system was used where 1 denotes 0.5 m, so for instance dimensions of parcel C are 3 x 3 x 3.

Container(also mentioned as **cargo**) is the cargo-space of 33 x 8 x 5 that needs to be filled.

Value density of a parcel - value/volume ratio specific for each type of a parcel.

CLP Container loading problem.

Possible packing - a packing of parcels in which there is either no more room for any other parcel, or in which we have ran out of parcels.

INTRODUCTION

In this section the assignment of the project Phase 3 is described and the reason of the research is explained. Furthermore the structure of the report is presented.

In the third phase of the project each group had to build a computer application that could be used for solving three dimensional knapsack problems. The first condition was to assume that a company owns trucks with a cargo-space of a given dimension and three types of parcels of given types (with a certain dimension and value). Moreover the application had to compute, for a given set of parcels, a packing configuration that would maximize the total value. The result had to be shown by a 3D-visualization from different perspectives. The final condition was that the company transported pentomino shaped parcels (of given types and sizes) and four questions needed to be answered about different possibilities of filling the cargo space.

Background, history and importance of the problem

Efficient solutions to the problem are of great importance, which is the reason why this problem is well known in the field of operations research. The first reason for its importance is economic: shipping companies saves money by shipping as much parcels using as little cargo space as possible. Not only shipping companies but also the cutting and packing industry benefits from these solutions, as filling a container with parcels is analogous to dividing a block of wood or foam into smaller pieces with as little loss of material as possible.

Secondly, solutions to the problem also help the environment: the transport of goods makes for high CO_2 emissions, so lowering the amount of needed transportation also lowers these emissions.

Tobias Dantzig was the first one to study knapsack problems in 1897 (Dantzig, 1930). As electronic computers started to come up, the first computational knapsack-solving algorithms were created. The first real research on the subject was done by Gilmore and Gomory in the 1960's (Gilmore, 1965). They were the first to research a multi-dimensional knapsack problem, using a linear programming approach. As computational power began to increase further, the efficacy of CLP algorithms steadily increased and the economic incentive for the problem resulted in a large increase in research on the topic.

Problem definition

The problem of filling a container with objects to either fill the space entirely or to achieve the highest total value belongs to the class of 0-1 Knapsack problems. An object cannot be split into fractions and it can be rotated by 90 degrees on each axis. It is specifically a single container loading problem (CLP) (Parrero), on which a lot of previous research has been done. The research on CLP's is quite varied, considering many different constraints such as weight, stability and practicality of packing the objects in real life. In this problem, such constraints are not considered, the parcels objects need to fit the container-space.

There are two kinds of objects given that must be loaded into the container space:

Parcels

Rectangular cuboids with dimensions of A(3x3x3), B(2x3x4) and C(2x2x4).

Pentominoes

Polycubic figures with five joint cubes in three different shapes (L,T,P).

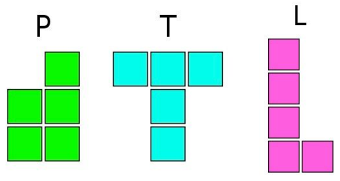


Figure 1. The 12 pentominoes (2016).

A value can be assigned to each object (3,4,5). The total container value is computed by the amount of loaded object types times their respective value.

Algorithms which solve this problem have several objectives:

a) Fill up as much volume as possible
b) Fill the container with as much value as possible
c) Find a solution in a reasonable amount of time
d) The algorithm can be used with arbitrary dimensions for the container and the objects.

In the next sections algorithms implemented in the application will be explained, then the experiments will be described and in the end the conclusion and references will be provided.

BACKTRACKING ALGORITHMS

Simple backtracking algorithm

The first algorithm that was implemented was a simple backtracking algorithm. It was implemented because it was reasonably simple to create, this ensured that at least one working and properly tested algorithm was finished before the end of the project. Furthermore, the algorithm provides useful context to the performance of the other implemented algorithms.

The algorithm is allowed to run for a specified amount of time, using a specified selection of parcels. During this time the algorithm finds some amount of possible packings, saving the one with the highest value.

The algorithm is backtracking in the sense that if a next parcel can not be placed, the last placed parcel is removed and substituted for a different parcel. It is however also backtracking in the sense that once we have found a possible packing, we save its total value and remove the last placed parcels to try and find other possible packings.

In the program, two options are given for running through the parcel list. Using the first option, the list is ordered by parcel type and the program tries to place parcels until either this type of parcel has ran out or the next parcel of this type doesn't fit, in which case the next parcel type in the list is selected.

Using this option, the parcel types can be sorted by value or by value-density, which makes the program a best-first algorithm. The most promising node of the search tree, the node before which the highest amount of valuable or value-dense parcels have been packed, is expanded first.

Using the second option for ordering the parcel list, the parcels are ordered randomly. This option provides context to the other orderings, as it shows the difference between ordering parcels using a clear strategy and arbitrarily trying to place them.

A greedy variation of the algorithm

A variation of this algorithm was made with as a goal to minimize its running time. In this variation, the backtracking approach is abandoned with respect to having multiple possible packings to create a greedy algorithm. In this algorithm the first found possible packing is returned, as opposed to comparing multiple packings and returning the one with the highest value. The reason why this is a greedy approach is because a locally optimal choice is made for each parcel that is placed. The parcel with the highest value or value-density is placed and, because there is no container-level backtracking, this parcel remains placed, resulting in a possible packing consisting only of locally optimal choices.

DIVIDE & CONQUER ALGORITHM

Divide & conquer is a widely used algorithm paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. [6]

Initially cuboid subspaces of the container are generated and stored in a List **S**. This works for arbitrary container dimensions.

Initialize S = (s^1 (minVertex, maxVertex),s^2..., s^n)

For each subspace $\mathbf{s^n}$ in **S**, the backtracking algorithm searches for a packing solution $\mathbf{P^n}$ of this subspace. As long as the number of available objects $\mathbf{Q^i}$ is high enough to fill $\mathbf{s^n}$ again, **P** is multiplied and spread in the container. If there is still empty container space **E(C)** left to fill after this, then apply a greedy filling with single objects. Least the packing solution of the whole container is stored in a List C(P).

```
For $s^n$ : S {
    P($s^n$) = findSolution
    While ($s^n$(C)  left) And ($Q^i$ >=$P^i$) → load P in C
    While(E(C)>0) → load o in C
    Store P(C)
}
```

In the end the highest value container solution is picked.

MAXIMAL LAYER FILLING ALGORITHM

The final implemented algorithm is based on the concepts of empty maximal layers(EML). An EML has to be a not occupied rectangular plane which

a) cannot be further extended towards any dimension
b) is not a subspace of another EML

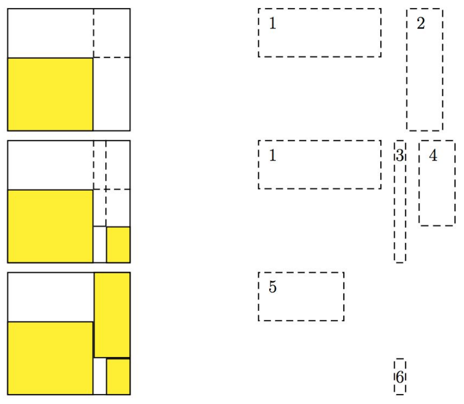


Figure 2. Maximal spaces 2D (Parreno, Alvarez-Valdes, Oliveira, Tamarit, (2008)).

This approach is an alteration of the Maximal Space Filling(MSF) Algorithm created by Parreño, Alvarez-Valdes, Oliveira and Tamarit (Parreño,2008)[2]. The MSF uses three-dimensional maximal spaces, instead of the two-dimensional maximal layers.

The reason for the alteration has two main reasons.

First of all is the space and time complexity decreased by finding empty spaces in a two dimensional layer. With a stack-based approach list of all existing EMLs can ideally be found in $O(3n^3)$. n stands for the elementary operation of finding a rectangle in a row of the container space. This has to be performed for each dimension with each orientation of the layer(XY,YZ,XZ).

The second reason for the alteration is that we assume our pentominoes to be flat objects, so they can be arranged in layers of different size.

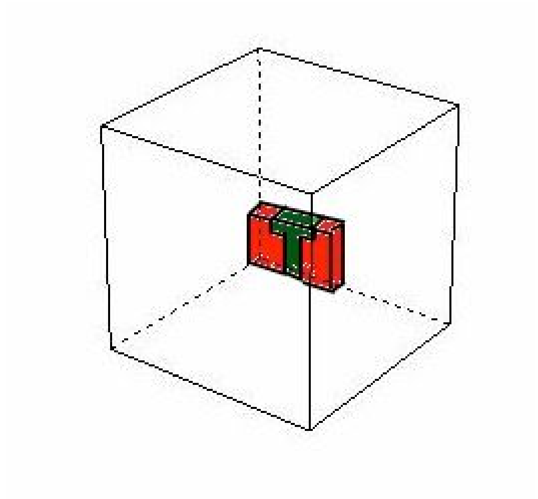


Figure 3. Layer with 3 available pentominoes.

The algorithm consists of two phases. A constructive phase in which a solution is found deterministically and an improvement phase in which the constructive phase is applied with an non-deterministic way for k% of the first loaded pentominoes. This way we generate a variety of different solutions for which we can pick the best one.

Constructive Phase

1. Initialization:

Finding configurations of pentomino boxes that can be used to fill the empty maximal layers. Those configurations consist consist of two to five pentominoes, tiled in a small rectangular shape. A short run of the tiling algorithm from phase 1 will find us solution for those shapes.

Example:

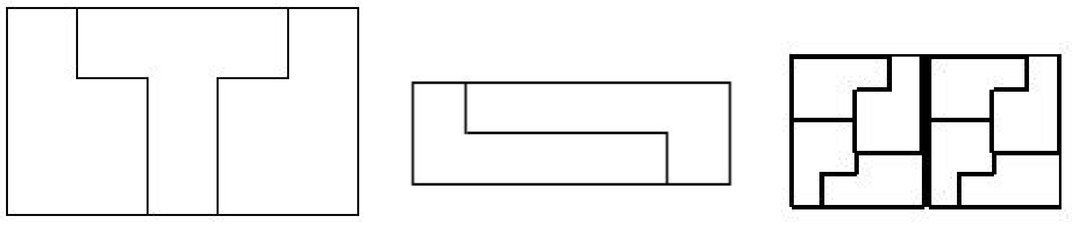


Figure 4. P-T-P, L-L and P configuration.

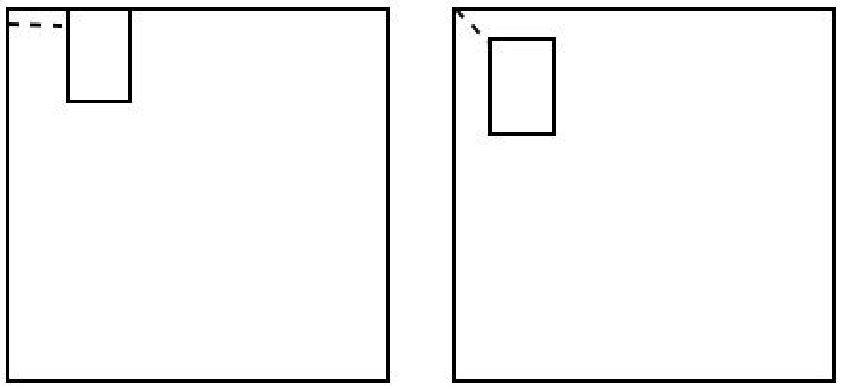
2. Recursive algorithm:

a) Find all empty maximal layers in the container and store them in a list **EMS**.
b) Choosing a layer **J** of **EMS** which is best to be filled next.
c) Find the best configuration of a pentomino layer to fill **J** by certain criterias.

b*: **J** is chosen by taking the the distance from each corner of each layer of **EMS** to the closest corner of the container. The EMS is represented by the coordinates of its maximum and minimum vertices. We use the lexicographical distance of each vertex after sorting the dimensions by distance in non-decreasing order. This way the algorithm tries to fill the corners first, then the edges, the wall and least the inner space.

Example: We have **EMS(i)** = {(1,1,3),(5,5,3)} in a (5,5,5) container. Then the nearest corner of **i** to the container is (5,5,3) and its distance is (0,0,2).

c*: For each pentomino box **B** we cover the area of **J** with multiples of **B** in two different orientations.



Example Figure: Figure 2 is further by lexicographical distance

We choose a configuration by its total value and the area covered of the EMS. As a tie-breaker, the amount of pentominoes is taken. A constraint is the amount of pentominoes left to choose from. If no box can be build anymore, a single pentomino is placed onto the layer.

Improvement Phase

For the improvement phase we let the algorithm run for a desired amount of iterations by restarting it each time a solution is found or the container cannot be loaded with more pentominoes. Here we choose non-deterministically one of the k% best configuration of boxes until m% of the container is filled to continue deterministically. This way we introduce flexibility in our final solution by having a different start in each run. The final solution is the solution which could fill the highest value in the container.

Note: The java implementation of the Maximum Layer Filling Algorithm was not completed by the deadline of 24.01.2018. The reason behind it, was the first attempt of implementing this with maximal spaces to let it also work for parcels. Finding three dimensional maximal spaces in reasonable time is not a trivial task. A solution of time complexity $O(n^3)$ for this problem can be looked up in the work of Nandy a. B[].

F.Parreno R.Alvarez-Valdes J.F Oliveira J.M Tamarit (2008) A Maximal-Space Algorithm for the Container Loading Problem

PERFORMANCE OF ALGORITHMS

In this section, results of a number of experiments will be shown and the following questions will be answered:

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 you can store in your cargo-space?
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 you can store in your cargo-space?

After some computations it becomes clear that the limit of the value of a cargo packed with A, B, C parcels is 247,5. It is certain that the best possible value theoretically can be obtained by filling the cargo space entirely with the most valuable parcel, however it is impossible to fill completely the cargo space with only one type of a parcel. A variation of the Divide & Conquer Algorithm which only tries to fill parcel A can prove this. This can be Therefore, the limit of total value (which is not a real solution) can be obtained by taking a maximal value of a coordinate (voxel), which is the highest value density, and multiplying it by number of coordinates(voxels) in the space.

0,1875 - the best value density (in this case, the value density of parcel A)

x 1320 - number of voxels in space (33 x 8 x 5)

247,5 - the maximum limit of total value.

EXPERIMENT 1

In this first experiment, different specified amount of parcels A, B and C are added to a container which has the following dimensions: x = 5, y = 8 , z = 33. Besides varying the given amount of parcels, the parcel ordering and runtime are varied as well. Specifying the amount of parcels as 100 means the number of parcels of this type is essentially unlimited, as it is impossible to fill the given container with more than 100 parcels of type A, B or C.

Backtracking algorithm results

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	192	110	216	**230**	206
0	by value/volume ratio	192	192	192	110	192	192	206
10	by value	**230**	192	196	110	217	**230**	208
10	by value/volume ratio	196	192	196	110	196	194	208
30	by value	**230**	192	196	110	217	**230**	208
30	by value/volume ratio	196	192	196	110	196	194	208

Table 1: Backtracking computational results of experiment 1.

The best found total value is 230, which is reached when parcels are ordered by value and an unlimited amount of parcels are available, as well as when only an unlimited amount of type A and C are available.

The lowest found total value is 110, which is reached when only an unlimited amount of parcels of type C are available.

The results show that ordering parcel types by value results in higher total values than ordering them by value/volume ratio. Because of this it might be the case that although when the parcels are ordered by value/volume ratio the parcels themselves are value-dense, they do not fit into the container as well as when ordered by value, resulting in less space-efficiency and an overall less value dense packing.

Finally, the results show that the value of the packing does not increase when changing the running time from 10 to 30 seconds. This is the case because the computation is simple enough for the algorithm to find its best solution in under 10 seconds.

Backtracking algorithm results using random parcel ordering

Run for [s]	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
	B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
	C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	200/199/209	192	192	110	203/203/208	202/194/205	189/194/183
10	215/219/214	192	192	110	214/215/214	216/218/220	197/200/199
30	219/215/213	192	192	110	217/216/216	221/222/221	199/200/197

Table 2: Backtracking algorithm computational results of experiment 1 using random parcel ordering.

The algorithm was ran three times for each specific test, the goal with these results is to provide some simple context to the results of backtracking using the other parcel orderings, which is why it was not deemed necessary to run the algorithms for a statistically significant number of times.

The best found total value is 222, which is reached when an unlimited amount of parcels of type A and C are available.

The lowest found total value is 110, which is reached when only an unlimited amount of parcels of type C are available.

The results generally vary but unlike the previous results of backtracking using determined parcel ordering, the found values notably increase when run time is increased.

Divide & Conquer algorithm results

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	192	110	224	216	**230**
0	by value/volume ratio	192	192	192	110	208	192	110
10	by value	**230**	192	200	110	224	216	**230**
10	by value/volume ratio	216	192	200	110	216	200	110
30	by value	**230**	192	200	110	224	216	**230**
30	by value/volume ratio	216	192	200	110	216	200	110

Table 3: Divide & Conquer computational results of experiment 1.

The best found total value is 230, which is reached when an unlimited amount of parcels of all types are available or only an unlimited amount of parcels of type B and C are available.

The lowest found total value is 110, which is reached when only an unlimited amount of parcels of type C are available.

As with the standard backtracking algorithm, found values do not increase when increasing the running time from 10 to 30 seconds.

Summary of experiment

When packing an unlimited amount of parcels of type A or an unlimited amount of parcels of type C we get the same result equal 192 or 110 respectively for all three algorithms.

The backtracking algorithm finds a best solution using unlimited amounts of parcel A and C, while the Divide & Conquer algorithm finds the same value of 230 using parcels B and C.

Changing running time from 10 to 30 seconds does not make change the found total value, except for the random ordering variation of the backtracking algorithm.

The Divide & Conquer algorithm gets better results than the backtracking algorithm when parcels are ordered by value/volume ratio and when only two types of parcels are used, except when parcels of type A and C are used.

EXPERIMENT 2

In this experiment, varying amount of parcels of type A,B and C are added to a container in which the algorithm runs through the containers dimensions in several different orders. This is achieved by changing the dimensions of the container, swapping the height, width and length values. As in the previous experiment, we varied the running time as well. (to see more detailed results of this experiment go to Appendix 3 on page).

Results

As these results were rather similar to those of experiment 1, no tables are provided, however, the following notable results have been found:

Packing an unlimited amount of parcels of type A and C results in the same generated value for every ordering of dimensions.

The best total value of 233 was found by the backtracking algorithm using the dimensions: x = 8, y = 5, z = 33, using an unlimited amount of parcels and a running time of 10 seconds.

The second best total value was found by Divide & Conquer algorithm when infinite many parcels of each type ordered by value were placed in the container in dimensions equal: x=33, y=5 and z=8 with run time 10 or 30 seconds.

The second best total value was found by the Divide & Conquer algorithm using an unlimited amount of parcels of each type ordered by value, using the dimensions: x=33, y=5 and z=8, and a running time of 10 seconds.

From these results it can be concluded that allowing the algorithms to run through the container by cycling through its dimensions in different orders can have notable effects on the found value.

EXPERIMENT 3

In this experiment, we compare the three algorithms using 30 parcels of type A, 30 of type B and 10 of type C, using the dimensions: x = 5, y = 8 and z = 33.

ALGORITHM	Packing order	Run time [s]	A: 30 B: 30 C: 10
BACKTRACKING	by value	0	218
	by value/volume ratio	0	196
	by value	10	218
	by value/volume ratio	10	200
	by value	30	218
	by value/volume ratio	30	200
DIVIDE & CONQUER	by value	0	224
	by value/volume ratio	0	204
	by value	10	224
	by value/volume ratio	10	210
	by value	30	224
	by value/volume ratio	30	210
RANDOM	random	0	200/207/205
		10	213/218/212
		30	222/214/214

Table 4: Computational results of experiment 3.

Results

The Divide & Conquer algorithm gives better results than the backtracking algorithm in all cases.

Packing parcels when ordering them by value is in all cases better than ordering them by value/volume ratio.

There is no difference when increasing the running time from 10 to 30 seconds, except for the backtracking algorithm using random parcel type ordering in which the results do improve.

The best found total value of 224 was reached by Divide & Conquer algorithm the parcels are ordered by value.

The second best found total value of 222 was reached by the backtracking algorithm using random parcel type ordering. Interestingly, for less than an unlimited amount of parcels using a random parcel type ordering can provide better results than using a determined ordering.

The lowest found total value of 196 was reached by the backtracking algorithm when parcels were ordered by their value/volume ratio. An explanation why ordering by value/volume ratio in this test and in others provides worse results than other orderings is because their value density is very similar: 0.187, 0.185, 0.166. Because of this the ordering is somewhat arbitrary.

As in the previous experiment, changing the running time from 10 to 30 seconds only makes a different for the backtracking algorithm using random parcel type ordering.

EXPERIMENT 4

In this experiment, varying amount of parcels are added to a container 8 times the size of the original container, using the following dimensions: x = 10, y = 16, z = 66.

ALGORITHM	Packing order	A: infinite B: infinite C: infinite	A: 1000 B: 0 C: 0	A: 0 B: 1000 C: 0	A: 0 B: 0 C: 1000	A: 1000 B: 1000 C: 0	A: 1000 B: 0 C: 1000	A: 0 B: 1000 C: 1000
BACKTRACKING	by value	1650	**1980**	1584	1650	1584	1650	1650
	by value/volume ratio	**1980**	**1980**	1584	1650	**1980**	**1980**	1650
DIVIDE & CONQUER	by value	1914	**1980**	**1760**	1650	**1980**	**1980**	1804
	by value/volume ratio	1764	**1980**	**1760**	1650	**1980**	**1980**	1650
RANDOM	random	1684	**1980**	1584	1650	1744	1801	1535

Table 5: Computational results of experiment 4.

Results

The best found total value found is equal to 1980, in which an unlimited amount of parcels of type A are used to completely fill the container.

Besides this packing, another packing was found which filled the entire container, reaching a total value of 1760 using an unlimited amount of parcels of type B using the Divide & Conquer algorithm.

The lowest found total value was 1535 which was reached by the backtracking algorithm using random parcel type ordering, while using an unlimited amount of parcels of type B and C.

EXPERIMENT 5

In this final experiment, we used pentomino shaped parcels to fill a container. Varying amount of parcels were added.

Algorithms	Run time [s]	Packing order	L: infinite P: infinite T: infinite	L: 120 P: 120 T: 100	L: 100 P: 120 T: 120	L: 120 P: 100 T: 120
BACKTRACKING	0	by value	1151	1079	1104	1099
	0	by value/volume ratio	1151	1079	1104	1099
	10	by value	1154	1079	1104	1099
	10	by value/volume ratio	1154	1079	1104	1099

Results

The best found value of 1154 was reached by the backtracking algorithm, using an unlimited amount of all types of parcels.

Ordering the parcels by value and value/volume ratio using pentomino shaped parcels does not change the outcome as all parcels have the same volume, resulting in the same ordering.

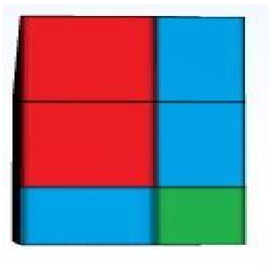
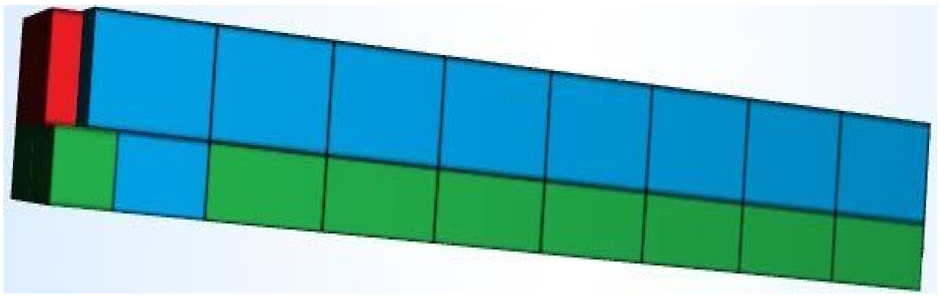
Increasing the running time generally did not change the found value, except for unlimited amounts of all types of parcels in which the found value increased marginally.

Is it possible to fill the complete cargo space with A, B and/or C parcels, without having any gaps?

None of our algorithms could find a complete filling of the cargo space without any gaps. The backtracking approach would eventually find one, but the search tree seems to be large to be computed in a reasonable amount of time. The representation of the parcels placement could potentially be reduced to increase the speed of the algorithm. But as the NP-Hardness of the Container Loading Problem is well known, discovering the whole search tree will not be possible unless P=NP.

If parcels of type A, B and C represent values of 3, 4 and 5 units respectively, then what is the maximum value that you can store in your cargo-space?

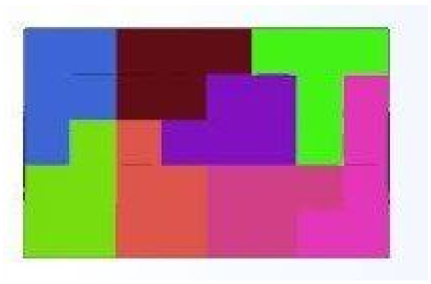
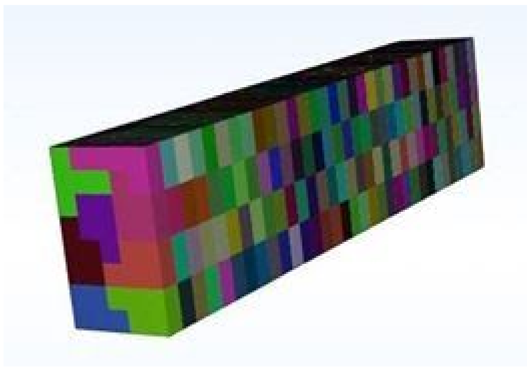
The best solution of 233 which could be found was with the value-greedy backtracking algorithm by traversing through the container matrix by switched x and y dimensions. The

reason behind that can be that the algorithm finds a well fitting (5x8) layer fast and just places this layer towards the length.

Is it possible to fill the complete cargo space with L, P and/or T parcels, without having any gaps?

Running the Divide & Conquer Algorithm as well as the Maximum Layer Algorithm will find such a solution in fast time. The reason behind this, as could be discovered, is that the layers of pentominoes perfectly fit into the dimensions of the container.

If parcels of type L, P and T represent values of 3, 4 and 5 units respectively, then what is the maximum value that you can store in your cargo-space?

The maximum result which could be obtained was a total value of 1154/1320 with a runtime of 10s. In a second run this should be improved by applying Divide & Conquer with only T-shaped pentominoes.

CONCLUSION

Dividing an NP-Hard problem into multiple sub-problems can be an effective strategy to find the best global solution even for very large search trees. However if the container cannot be effectively divided into subsets then this approach might not be more effective than a backtracked greedy approach. This could be observed with the Divide & Conquer

Algorithm. While there were no good subsolutions for parcels, it was successful in placing pentominoes.

The Divide & Conquer Algorithm could also be improved by not using the first found solution for a subset, but the complete solution with the highest value.

Furthermore introducing well chosen heuristics, an outside-inside placement-order and a non-deterministic improvement phase would potentially lead to better results and bring more flexibility to the discovered solutions. This research is to be continued by finishing and optimizing the Maximal Layer Algorithm.

SOURCE REFERENCE

- [1] Bortfeldt, A., Wascher, G. (2012). Container Loading Problem - A State-of-the-Art Review**.** 12-01-2018, http://www.fww.ovgu.de/fww_media/femm/femm_2012/2012_07-EGOTEC-503ec3895182dc0d922a6bd7feebb3a5.pdf
- [2] Parreno, F., Alvarez-Valdes, R., Oliveira, J.F., Tamarit, J.M. (2008). A maximal-space algorithm for the container loading problem. 11-01-2018, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.418.7210&rep=rep1&type=pdf
- [3] Dantzig, T. (1930). Number: the language of Science. http://www.engineering108.com/Data/Engineering/Maths/Number_the_language_of_science_by_Joseph-Mazur_and_Barry-Mazur.pdf)
- [4] Gilmore,P. C., Gomory, R. E. (1965). Multistage Cutting Stock Problems of Two and More Dimensions. http://people.math.yorku.ca/chensy/GG1964.pdf
- [5][Nandy and Bhattacharya(1996) Maximal Empty Cuboids Among Points and Blocks, *Computers Math Applic Vol 36 No 3. 1998*]. https://www.researchgate.net/publication/240110435_Maximal_empty_cuboids_among_points_and_blocks
- [6]Quote from the wikipedia page of the Divide and conquer algorithm https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm

Figures

- (2016). The 12 pentominoes. Retrieved from https://simple.wikipedia.org/wiki/Pentomino

APPENDIX 1

Project Assignment phase 3

In Block 1.3 we want you to build a computer application with a user friendly interface that can be used for solving so-called three dimensional knapsack problems:

Assume that your company owns trucks with a cargo-space of 16.5 m long, 2.5 m wide and 4.0 m high. Assume that your company 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 x 1.5

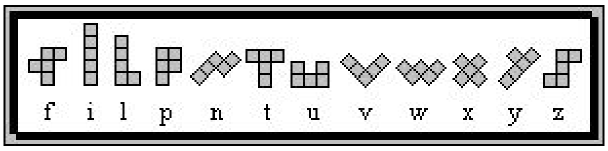
A parcel of a given type also has a certain value. Denote these values by v_A, v_B and v_C for types A, B and C respectively. Now, make a computer application that computes, for a given set of parcels (that may or may not fit into a truck), a packing that maximizes the total value.

The application does not necessarily have to find the best answer in all cases, but it should be able to find a good approximation. The application should also make a 3D-visualization of its answer – from different perspectives.

Use your application to answer the following questions:

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

Once you have answered these questions, assume that your company now transports pentomino shaped parcels of types **L, P and T**, where each of these pentominoes consists of 5 cubes of size 0.5 x 0.5 x 0.5.



Adapt your application from Phase 1 to answer the following questions:

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

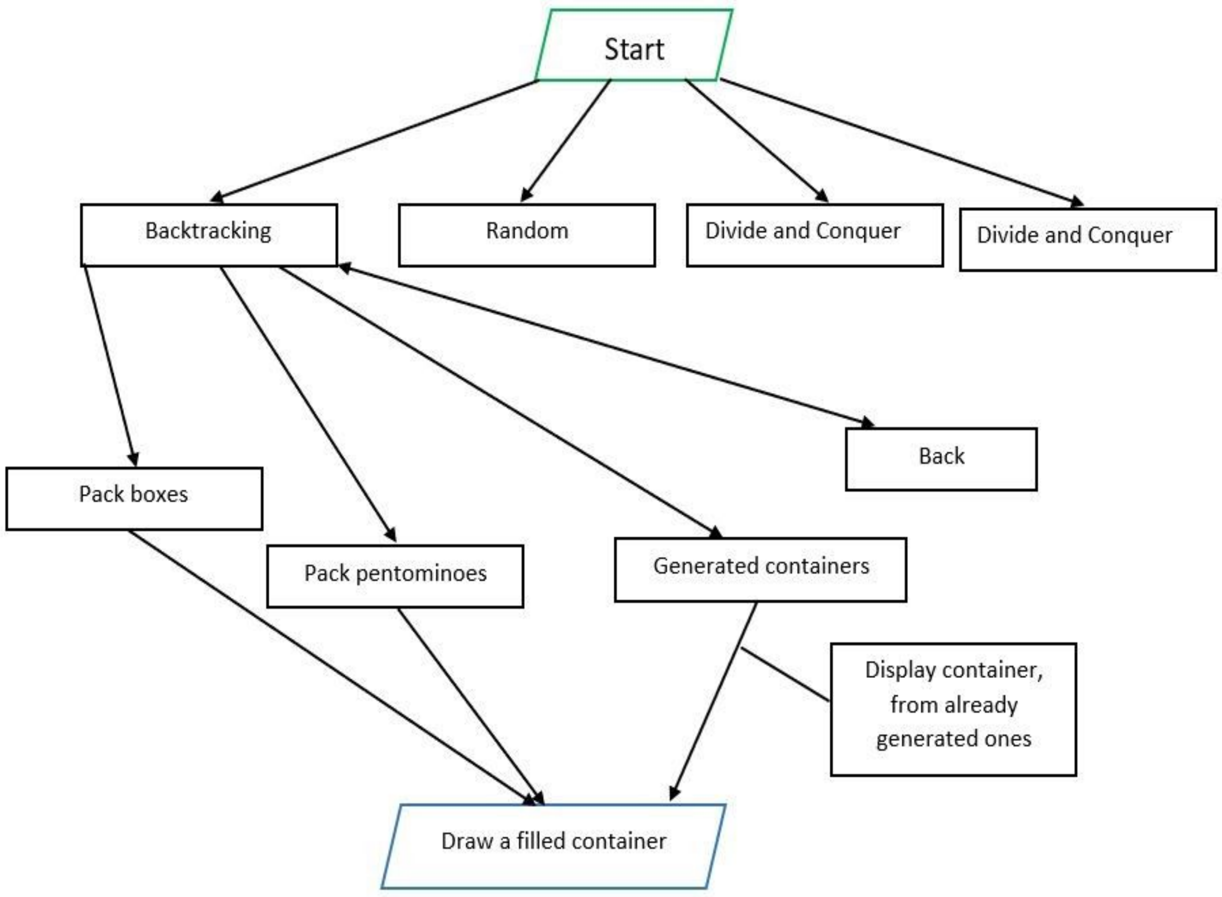
In case you are wondering, all the phases of this project are motivated by "packing" problems in computer science. These are very natural problems that occur when you want to make the most economically efficient use of a given space, subject to conditions on the total size of the space, the shapes of the items you are packing and their value. It probably won't surprise you to learn that good algorithms for packing problems are important in, for example, shipping container loading.

APPENDIX 2

APPLICATION DESIGN

3D library
JavaFX was used to create the graphical user interface.

Application structure
The application consists of a window with a container in the left and comments and options on the right. The user can choose an algorithm and more options will appear. Then he can choose the way he wants the algorithm to solve the container. After the container is solved, a visual representation of the added boxes/pentominoes is displayed. The user can rotate the container and see it from each side. The boxes and pentominoes are represented through JavaFX 3D Triangle Mesh. A custom class Box was created that generates boxes out of Triangle Mesh. After a possible solution has been generated and displayed, it is being stored and through option Generated Containers, all found solutions can be displayed and scrolled through.



APPENDIX 3

Tables

Results of changing dimensions of the container for three algorithms.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	222	192	176	110	176	230	198
0	by value/volume ratio	192	192	176	110	192	192	198
10	by value	222	192	176	110	178	**230**	206
10	by value/volume ratio	194	192	176	110	194	192	206
30	by value	222	192	176	110	181	**230**	206
30	by value/volume ratio	194	192	176	110	194	192	206

Table 6: Backtracking computational result of experiment 2, dimensions: x=33 , y=8 , z=5.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	216	110	216	**230**	**230**
0	by value/volume ratio	192	192	216	110	192	192	**230**
10	by value	**230**	192	216	110	218	**230**	**230**
10	by value/volume ratio	196	192	216	110	196	194	**230**
30	by value	**230**	192	216	110	218	**230**	**230**
30	by value/volume ratio	196	192	216	110	196	194	**230**

Table 7: Backtracking computational result of experiment 2, dimensions: x=5 , y=33 , z=8.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	218	192	172	110	172	**230**	218
0	by value/volume ratio	192	192	172	110	192	192	218
10	by value	221	192	172	110	176	**230**	218
10	by value/volume ratio	197	192	172	110	196	196	218
30	by value	221	192	172	110	176	**230**	218
30	by value/volume ratio	198	192	172	110	196	196	218

Table 8: Backtracking computational result of experiment 2, dimensions: x=8 , y=33 , z=5.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	176	110	176	**230**	**230**
0	by value/volume ratio	192	192	176	110	192	192	**230**
10	by value	**230**	192	184	110	180	**230**	**230**
10	by value/volume ratio	197	192	184	110	196	195	**230**
30	by value	**230**	192	184	110	180	**230**	**230**
30	by value/volume ratio	197	192	184	110	196	195	**230**

Table 9: Backtracking computational result of experiment 2, dimensions: x=33 , y=5 , z=8.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	160	110	184	**230**	206
0	by value/volume ratio	192	192	160	110	192	192	206
10	by value	**233**	192	168	110	188	**230**	206
10	by value/volume ratio	197	192	168	110	196	196	206
30	by value	**233**	192	168	110	188	**230**	208
30	by value/volume ratio	198	192	168	110	196	196	208

Table 10: Backtracking computational result of experiment 2, dimensions: x=8 , y=5 , z=33.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	222	192	176	110	180	216	198
0	by value/volume ratio	196	192	176	110	192	192	110
10	by value	222	192	192	110	216	216	204
10	by value/volume ratio	211	192	192	110	204	200	110
30	by value	222	192	192	110	216	216	210
30	by value/volume ratio	222	192	192	110	216	200	110

Table 11: Divide & Conquer computational result of experiment 2, dimensions: x=33 , y=8 , z=5.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	216	110	224	216	230
0	by value/volume ratio	196	192	216	110	224	192	110
10	by value	**230**	192	216	110	224	216	230
10	by value/volume ratio	216	192	216	110	216	200	110
30	by value	**230**	192	216	110	224	216	230
30	by value/volume ratio	222	192	216	110	216	200	110

Table 12: Divide & Conquer computational result of experiment 2, dimensions: x=5 , y=33 , z=8.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	222	192	176	110	180	216	218
0	by value/volume ratio	194	192	176	110	196	196	110
10	by value	222	192	180	110	216	216	204
10	by value/volume ratio	211	192	180	110	204	200	110
30	by value	222	192	192	110	216	216	210
30	by value/volume ratio	211	192	192	110	204	200	110

Table 13: Divide & Conquer computational result of experiment 2, dimensions: x=8 , y=33 , z=5.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	192	110	192	216	**230**
0	by value/volume ratio	192	192	192	110	196	192	110
10	by value	**231**	192	216	110	216	216	**230**
10	by value/volume ratio	211	192	216	110	204	200	110
30	by value	**231**	192	216	110	216	216	**230**
30	by value/volume ratio	222	192	216	110	216	200	110

Table 14: Divide & Conquer computational result of experiment 2, dimensions: x=33 , y=5, z=8.

Run for [s]	Packing order	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
		B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
		C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	by value	**230**	192	192	110	224	216	206
0	by value/volume ratio	192	192	192	110	192	192	110
10	by value	228	192	192	110	224	216	210
10	by value/volume ratio	216	192	192	110	216	204	110
30	by value	228	192	192	110	224	216	210
30	by value/volume ratio	222	192	192	110	216	204	110

Table 15: Divide & Conquer computational result of experiment 2, dimensions: x=8 , y=5, z=33.

Run for [s]	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
	B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
	C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	179/176/187	192	176	110	184/181/179	187/183/197	175/181/162
10	196/194/200	192	176	110	188/188/187	222/214/217	205/199/194
30	200/201/202	192	176	110	189/188/189	214/215/215	198/202/198

Table 16: Backtracking algorithm using random parcel ordering computational result of experiment 2, dimensions: x=33 , y=8, z=5.

Run for [s]	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
	B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
	C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	207/208/200	192	216	110	216/209/216	206/203/202	208/208/205
10	217/218/215	192	216	110	221/222/221	216/219/220	218/222/218
30	220/217/216	192	216	110	222/223/222	219/221/218	220/221/222

Table 17: Backtracking algorithm using random parcel ordering computational result of experiment 2, dimensions: x=5 , y=33, z=8.

Run for [s]	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
	B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
	C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	178/176/167	192	172	110	176/166/168	189/197/188	181/193/181
10	195/192/197	192	172	110	182/181/180	209/213/214	206/204/202
30	202/200/200	192	172	110	183/186/182	219/217/215	210/204/206

Table 18: Backtracking algorithm using random parcel ordering computational result of experiment 2, dimensions: x=8 , y=33, z=5.

Run for [s]	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
	B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
	C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	197/196/190	192	176	110	185/185/181	202/201/207	192/210/198
10	207/210/206	192	176	110	191/190/191	216/214/218	216/216/214
30	211/207/210	192	176	110	193/192/192	219/219/224	220/216/215

Table 19: Backtracking algorithm using random parcel ordering computational result of experiment 2, dimensions: x=33 , y=5, z=8.

Run for [s]	A: infinite	A: 100	A: 0	A: 0	A: 100	A: 100	A: 0
	B: infinite	B: 0	B: 100	B: 0	B: 100	B: 0	B: 100
	C: infinite	C: 0	C: 0	C: 100	C: 0	C: 100	C: 100
0	192/194/181	192	160	110	178/183/170	196/196/202	179/184/191
10	208/210/209	192	160	110	185/187/186	219/214/219	199/198/199
30	209/209/210	192	160	110	187/189/188	221/218/217	199/198/199

Table 20: Backtracking algorithm using random parcel ordering computational result of experiment 2, dimensions: x=8 , y=5, z=33.