Vrije Universiteit Amsterdam

Solving the container relocation problem using tabu search

Research Paper Business Analytics

Wilte Falkena
2543386
November 2018

Supervised by drs. B. G. Zweers

1 Abstract

The goal of this paper is to find out if tabu search can give new insights in the problem of solving the container relocation problem. The container relocation problem is about the relocation of containers minimizing the number of moves needed to retrieve all the containers. It is assumed the order in which the containers are picked up is known. The relocation of containers is a crucial operation in container ports all around the world. By a process called 'preprocessing', the containers are rearranged in order to minimize the number of relocations needed for picking up the containers.

In this research paper, the container relocation problem is investigated using tabu search and a variety of other algorithms. These algorithms are evaluated on 1440 different instances with varying width and number of containers. These results are then compared to other methods and thoroughly evaluated.

Contents

2 Introduction

This research paper revolves around the Container Relocation Problem, in particular the Stochastic Container Relocation Problem. In container ports all around the world, every day millions of containers are unloaded from ships, to be picked up by a truck at a later moment. The setup in which the containers come from the ship is not always an optimal one. When a containers is picked up, it might be the case that a lot of other containers have to be moved around to retrieve the particular container that is picked up.

This paper revolves around the reshuffling of the containers prior to being picked up (for example the day before retrieval). This process is called preprocessing. The goal of the preprocessing is to find a better container setup, minimizing the number of container relocations needed to retrieve containers that are picked up. Of course taken into account here is information about the time frame in which a container is picked up.

Different algorithms are developed to solve this problem. These algorithms are tested on a large number of different instances and then compared on a number of statistics. The algorithms this paper is about is the tabu search algorithm. Tabu search is a local search algorithm based around declaring previously found solutions as taboo. Taboo solutions can not be visited again in a next attempt to find a solution. There are no previous works on solving this problem with tabu search. The similarities and differences between the algorithms are evaluated and discussed to give a good view on the problem and how to algorithms approach the problem.

First, some standard terms are explained in section three, and some existing algorithms are shown in section four. Next, in sections five and six, the basic approach and tabu search are explained. In section seven, the algorithms which the tabu search is compared to are explained, followed by the results and conclusion in sections eight and nine.

3 General information

To explain the process of solving the preprocessing problem, a few general terms are explained first.

Stack A stack is a number of containers on top of each other. It is possible for a stack to be empty. Only the top container on a stack can be retrieved. New containers can only be placed on top of a stack.

Rows Stacks are placed next to each other. Each stack is placed in a row. A row always has a stack, however this stack can be empty.

Bay A bay are all rows present in the problem. In the real world, a port might have multiple bays with containers. In the problem that is investigated in this paper, only one bay is investigated each time. The problem can also be extended to multiple bays at the same time, however this is out of the scope of this paper.

Relocation A relocation is the moving of a container in a bay from one row to another row. A relocation must always move the top container of stack, and move to the top of another stack. A relocation cannot take place between different bays.

Instance An instance in this problem is the setup of the containers. For an instance, only one bay is used. The number of rows and maximum height of a stack may differ per instance. An example of an instance with 3 rows and maximum stack height of five is shown in Figure 1. The second row with a stack containing 3 is colored orange.

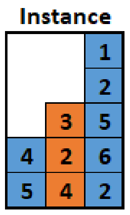


Figure 1: Container Instance

Preprocessing Preprocessing is the process of rearranging the containers. The preprocessing is used to put the containers in a better setup in order to minimize the number of relocations needed when the containers are picked up.

Daily process The daily process is the process of picking up the containers. During the day, the trucks arrive to pick up the containers in the bay.

4 Literature review

The preprocessing problem has been investigated before [4]. A common assumption here is the order in which the containers are picked up is known. It is important to note here, that the algorithms explained below are algorithms which decide of where to move a container. The container that is moved is not always representative of A few of these previous investigations are briefly explained below:

4.a Random

The random heuristic is one of the most basic approaches to investigating the preprocessing problem. This heuristic chooses a random container to be moved from one random stack to another random stack [1]. The only restrictions in this approach are that the stack to which the container is moved is not full, and the container is on top of a stack. Every move here has the same chance of being executed.

With this heuristic it is of course possible to find a good solution to the problem. Depending on the other restrictions of the model this approach might give an optimal solution. It might however take a lot of time to find this solution because the algorithm is given no real direction for which move it has to do. An example of the random search algorithm is shown in Figure 2. A move is indicated by a green container. The place where this container came from, is highlighted in red.

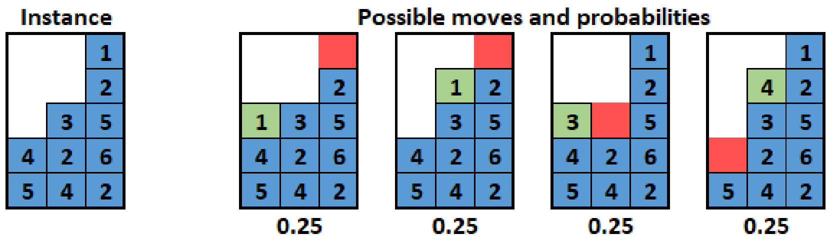


Figure 2: Random Algorithm

4.b Leveling

The leveling heuristic is a little more advanced approach to the problem. When a container is moved, the stack to which it is moved is based on the number of containers in the stack. The algorithm tries to make all stacks of the same height[1]. To do this, the stack with the fewest amount of containers is selected to move the container to.

This heuristic is already an improvement on the random heuristic. It has a smarter approach to finding the right stack to move a container to. By moving the container to the lowest stack, other stacks are left open for the possible moves of other containers. A lower stack often has a lower minimum time frame than a high stack, as there are fewer containers present. The chance of the moved container having to move again is lower than with the random heuristic. An example of the leveling algorithm is shown in Figure 3

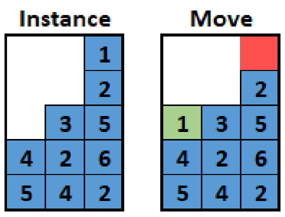


Figure 3: Leveling Algorithm

4.c The Expected MinMax (EM)

The expected Minmax heuristic is a method which tries to make sure a containers only has to be moved once [1]. It tries to pick out the best stack of where a container has to be moved. The stack which the container has to be moved to is determined by two rules. We assume here the time frame of the container to be moved is called t.

- Rule 1: If there is a stack where the lowest time frame is lower than than t, move the container to a stack where this minimum time frame is minimized. A stack with a higher minimum time frame can be used for another container with a higher time frame.
- Rule 2: If there is no stack satisfying rule 1, There are two cases following the same rule. The first case is if there is minimum stack time frame equal to t, then this lowest time frame is the maximum of all minimum time frames. The stack with the lowest number of containers is then selected. The second case is where the minimum time frame is higher than t. Here the same rule holds. The container is moved where the minimum time frame of the stack is maximized. If there are multiple stacks satisfying this condition the stack with the lowest number of containers is chosen.

The idea behind the expected minmax is to try and avoid as many relocation moves as possible while moving containers. This is done by trying to move containers to stacks where they are unlikely to have to be relocated again. If there is no stack where this expected number of relocations after the move is zero, the least bad stack is chosen, trying to leave room for other relocation moves.

A move of the expected minmax algorithm is shown in Figure 4. The algorithm decides to place the container on the second stack. This leaves room for another container to be placed on the first stack. The first stack is now able to receive a container with time frame one, two, or three. If the container was placed on the first stack, the second stack would only be able to receive a container with time frame 1.

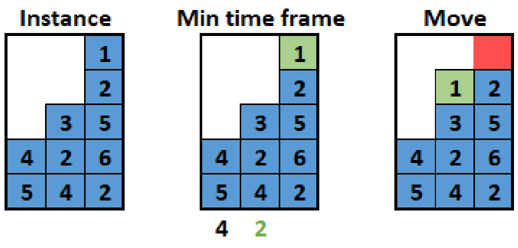


Figure 4: Expected MinMax Algorithm

5 Basic approach

The preprocessing in the stochastic container relocation problem is solved using tabu search and some other algorithms, however with all of these approaches some standard statistics are used. These are explained here.

5.a Instances

The algorithms are evaluated on 1440 different instances [5]. These instances consist of the setup of all the containers on the grid, as well as the time frame in which the containers are picked up the next day. These time frames give an interval of when the containers will be picked up, where these intervals do not have any overlap. However, there might be time frames in which multiple containers are picked up, so there is no way to know beforehand which of these containers in the same time frame is picked up first.

These instances are split up in groups of different width and number of containers. There are 240 instances for each of the widths from five wide to ten wide. Half of these instances are container grids where half of the grid is filled. The other half are instances in which 67% of the grid is filled. These instances are considered harder than the instances with a 50% fill rate. In these instances, usually fewer moves are possible. This limits the search space and a 'bad' move could lead the search to a setup where no more meaningful improvements can be done. For the less full instances, there is more space to correct these 'bad' moves.

5.b Container score

The containers are given a score based on the setup in which they are on the grid. Each container is scored, with four different scores which can be given. The scores are given based on the expected number of relocation moves the container has to do during the day it will be picked up.

For every instance all containers are classified. Next, An algorithm for picking up the containers is ran. This algorithms picks up the containers, as well as relocating the containers if the container to be picked up is currently not on top a stack. When there are multiple containers in the time frame, the algorithm chooses a random container in this time frame to be picked up first. All the relocations are counted. Now for every instance, we have the number of containers of each type, and the expected number of relocations needed to retrieve all containers. Then, using the Excel solver, the scores that need to be assigned to each container class are calculated and rounded to nice values.

These scores and their values are as follows:

5.b.1 Good score

A good score is assigned to a container which will be never be relocated during the day. This means that whenever a container is picked up during the day, this container will not have to do a relocation move to free up another container. The criterion for a good container score are as follows:

- There are no containers under this container which have to be picked up prior to this container.

A container with a good score does not have to be relocated before being retrieved from the container bay, thus the good score has a value of 0.

5.b.2 Medium score

A medium score is assigned to a container which has to relocated once during the day. This means that there is a container of a lower time frame below. In just one move however, this container can be relocated to a place where it will be of good quality. The criteria for a medium container score are as follows:

- There is at least one container with a lower time frame below this container.
- Within at maximum one relocation move, this container can be moved to a place where it is of good quality.

A container with a medium score has to be moved again just once, thus the medium score has a value of 1.

5.b.3 Bad score

A bad score is assigned to a container which has to be relocated at least once during the day. Thus, there is a container below this container which is picked up prior to this container. On top of that there is, at the current moment, no place where this container can be moved and change to a good quality. The criteria for a bad container score are:

- There is at least on container with a lower time frame below this container.
- There is, in the current configuration, no stack where this container can be moved where it will become of good quality.

A container with a bad score most likely has to be moved more than once. This means the score should be higher than one. Because it is likely but not always true that the container has to be moved more than once, thus the bad score has a value of 1.5.

5.b.4 Same score

A same score is assigned to a container which is on top of a container with the same time frame. On top of that, this container has to move only once in the case where the container below is picked up beforehand. The other scenario in which this container is picked up before the container below, There will be no relocation moves needed. Thus this container either has to do one move, or no move at all. The criteria for a container with same score are:

- There is at least one container below this container that has the same time frame as this container.
- At least one of the containers with the same time frame, is of good quality.
- Within at maximum one relocation move, this container can be moved to a place where it is of good quality.

Whenever a same score container has to be moved, it only has to be moved once. Because the picking up of a container is uniformly distributed, there is a chance of 0.5 that the container has to be moved once and a chance of 0.5 it can immediately be retrieved. Because of this, the same score gets a value of 0.5. On average such a container has to be moved 0.5 times.

5.b.5 Example of container scores

The scores previously mentioned get assigned each time a container is moved. The container scores are explained using Figure 5. The number in the container correspond to their time frame. There are containers that have the same time frame.

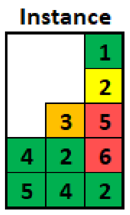


Figure 5: Container Scoring

In Figure 5, containers have colors. A green color means the container has a good score. Orange is for the medium score, yellow for a same score and red is used for a bad score. Containers 5 and 6 have received a bad score. Whenever the container below, container 2, is picked up, there is no place where these containers can be moved without having to move them again. Container 3 receives a medium score. When the container below, container 2, is picked up, this container can be moved to the stack in row one. When moved, it is of good quality and thus does not have to be moved again. Finally container 2 gets a same score. This is because on the bottom of this stack, there is another container with time frame 2. There is a probability this container has to be moved before being picked up, however it might not need to. This depends on which of these containers is picked up first in this time frame. When it has to be moved however, there is a stack it can be placed on where it is immediately of good quality. That stack is in row one.

After all these scores are assigned, the total grid score is calculated. There are two bad containers amounting to 3.0. There is one medium container which adds 1.0, and one same container for the last 0.5. In this case, the total grid score for the instance in Figure 5 is 4.5.

5.c Combined score

The container relocation problem in this paper has the goal of finding a good combined score. The combined score consists of the grid score as well as a penalty per move. This combined score gives the quality of the grid at any given moment.

The grid score is built up of the score of all the containers. These scores are added up to get the grid score. On top of that, the penalty per move is added to the grid score for every move needed to achieve the current configuration. This final number is the combined score.

5.d Search space exploration

For search space exploration, a variety of the depth first search algorithm is used. Depth first search is an algorithm which explores solution until it reaches a stopping condition. In the preprocessing problem, the algorithm starts at the starting configuration. It scores all possible moves, and chooses one move on which to continue. This goes on until the algorithm reaches a stopping condition and the current iteration is stopped. As soon as this stopping condition is reached, the solution is saved. At this point an iteration is finished. When an iteration is stopped, a new iteration is started at the starting configuration. An iteration is basically the searching of the algorithm from the starting configuration, exploring all configurations it finds on the way, until a stopping condition is reached.

While increasing in depth, the algorithm has to decide which solution to explore. This is done semi-randomly. The main goal of the research is to achieve a lower combined score, however to achieve this score, sometimes a move which decreases the combined score has to be done. These moves are not prioritized but should definitely not be ruled out.

Every possible new configuration which can be achieved from the current configuration gets assigned a score. This score is based on the combined score of the configuration. After this, each configuration gets a probability it is chosen as the next move. For a low combined score, the chance is high. For a high combined score, the chance is low. This directs the search in the direction of the highest score, while not ruling out moves which give a lower score.

For the assignment of the probabilities, the scores are first transformed in Formula 1. Lower scores should have higher values than higher scores. This transformation is done using the following formula:

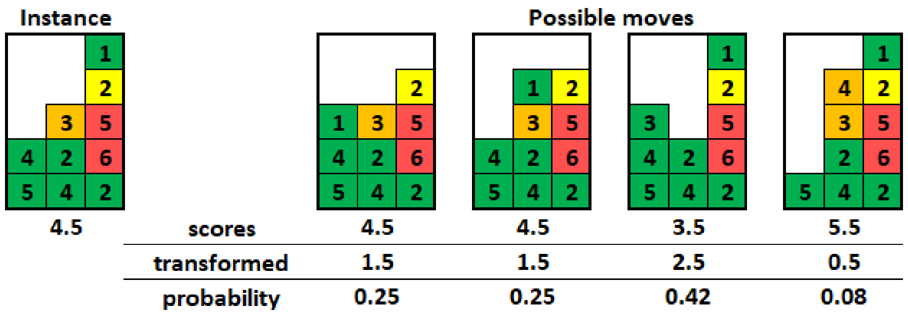
$$\text{Transformed score} = \text{maximum score} - \text{combined score} + 0.5 \tag{1}$$

After this transformation, the probabilities are calculated using Formula 2

$$\text{Probability} = \frac{\text{transformed score}}{\text{sum of transformed scores}} \tag{2}$$

In the Formula 1, a half is added to the transformed score at the end. This is done to make sure the maximum score is not transformed to a value of zero. This is because a value of zero for the transformed score would result in a probability of zero.

An example of the score calculation is shown in Figure 6.



scores	4.5	4.5	3.5	5.5
transformed	1.5	1.5	2.5	0.5
probability	0.25	0.25	0.42	0.08

Figure 6: Directed Random Search

6 Tabu Search

Tabu search is an algorithm based around declaring previously found solutions as 'tabu'. A tabu solution is stored in a tabu list. Every solution in the tabu list cannot be visited again. The next iteration(s) of the algorithm, the configurations in the tabu list will not be explored again, thus limiting computation time. This also helps with making sure a broader range of possible solutions is explored.

An important aspect of tabu search, is that at every step in the process, the algorithm does not always take the seemingly best solution. Due to the nature of the algorithm, sometimes seemingly worse solutions are chosen which hopefully lead to better solutions in the end. This is done to avoid local optima.

This paper focuses on tabu search as opposed to other local search algorithms like Genetic Algorithms or Simulated Annealing. Tabu search uses local search to overcome local optima. This is done by the algorithm allowing non-improving moves [2]. To prevent cycles, the tabu list is used. This directs the algorithm in the direction of not yet explored solution in an attempt to leave a local optima. A clear difference with Simulated Annealing is that tabu search does not just continue on one of the neighbouring states [3], but can explore plenty of them. This is due its iterative structure. Another important note here is that for this problem, the number of steps taken to reach the solution is important. For simulated annealing, a optimal container setup might be found, but the number of steps to reach this setup can be very large. This makes the 'best' setup not necessarily a good solution. The same holds for genetic algorithms.

Genetic algorithms have no real direction in which they search, they combine solutions with good fitness and (randomly) mutate them in an attempt to find a better solution. The clear advantage of tabu search here, is that the search is generally directed towards the optimum. For the genetic algorithm, it might take a lot of time to reach this same optimum. Another big challenge of genetic algorithms is that a suitable representation has to be found for the problem instances. This is not always easy.

6.a Tabu lists

Tabu lists can be used for a different variety of problems. For example, it can be used to make sure a solution is not explored twice in the same iteration. It can also be used to direct the search in such a way that the following iterations does not go in the same direction as in the current iteration. This can only be achieved if the tabu list(s) have a good size. If the list is not long enough, cycles may still occur. When the tabu list size is not limited, there will come a point where there are no possible moves left while the optimal solution is not yet found.

For the preprocessing problem, four different tabu lists will be used. The main function of the tabu lists is to make sure a broad range of grid configurations is explored. On top of that, for every iteration a tabu list is used to make sure a previously found solution is not visited again. To do this, four different tabu lists are used.

The first tabu list makes sure a previously found solution is not explored again in the same iteration. A solution gets added to the tabu list as soon as it is explored. This tabu list has an unlimited size. After every iteration it is emptied.

The other tabu lists are used to make sure a wider area of the search space is explored. Three tabu lists are used for this. Every first, second and third move is added to a separate tabu lists. In the next iteration, These solution will not be explored again, thus decreasing the possible amount of moves for this iteration. These tabu lists have a maximum size, varying on the width of the container grid. When this maximum size is reached, the configuration that was added first is deleted again. It works using a first in first out system. These tabu lists make sure the algorithm explores a part direction of the search space at the start of the iteration.

The size for these tabu lists is three. For the small instances of width 5, this should very much suffice, given the relatively small amount of possible moves which can be done. Using this size, the chance of all possible moves being tabu will be low. If all possible moves are tabu, the algorithm has no more moves to do and will automatically despite probably not having found a good solution yet.

6.b Stopping conditions

The tabu search uses multiple stopping conditions. The first stopping condition is the maximum number of iterations. This stops the algorithm after a set amount of iterations. When this condition is not present, the algorithm keeps going for ever.

The second stopping condition stops the algorithm when there is no improvement after a set amount of iterations. This amount is of course lower than the maximum number of iterations. This stopping condition is used to make sure the algorithm does not keep going when it has most likely found the optimal solution. An example is if the algorithm found the optimal solution in the first iteration, but still has to do all of the other iterations because it is not stopped. This of course takes up a lot of computation time so it is not favourable.

The third stopping condition makes sure the algorithm stops an iteration whenever there is no more improvement possible. As mentioned before, a penalty is awarded for every move that is done. This penalty is added to the combined score. When the total penalty of all moves together is higher than the score of the starting configuration, doing any extra moves will only give a higher scores than at the starting configuration. There is no point in doing any extra moves and the current iteration is stopped if this condition is violated.

The final stopping condition is when the algorithm has found a optimal configuration. Whenever the score of a configuration is zero, no further moves can be done to improve the current configuration, thus the iteration is stopped. Of course this does not stop the whole algorithm, as there might be a possibility of the algorithm finding the same score using fewer moves.

6.c Advantages

Tabu search is an improvement over other search space exploration algorithms, because it gives a clearer direction to the search. By elimination possible search directions that have already been evaluated, the search is direct towards exploring new and possibly better solutions.

On top of directing the search towards new solutions, tabu search also attempts to avoid loops. By doing this, the search has a clearer direction as well as limiting computational time by trying to make sure solutions are not explored twice.

Tabu search also allows algorithms to do moves which makes the solution of worse quality. Sometimes this is needed to achieve a better final solution. By doing this, the algorithm does not get stuck in local minimums. Algorithms which only allow improvements to the solution often get stuck in these minima.

6.d Example of tabu search

To further explain the concept of tabu search, and example is given in this section. The instance that is used here is the same that was used to explain the other algorithms. In Figure 7, all possible moves are shown and the probabilities are calculated.

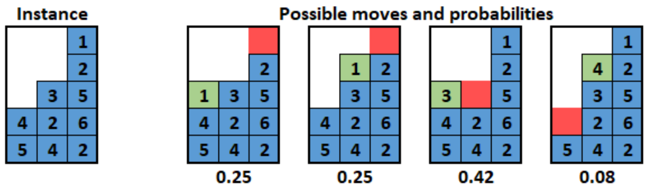


Figure 7: Tabu search starting point

The third move that is shown has the highest probability of being chosen. If this move is executed we end up in the situation in Figure 8.

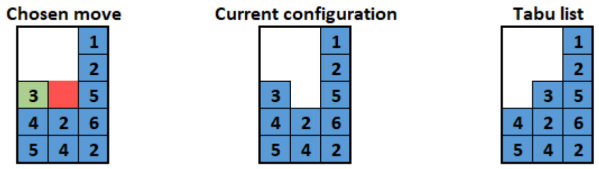


Figure 8: Tabu search situation after first move

The previous configuration is added to the tabu list. This configuration can no longer be visited. When the next possible moves are explored in Figure 9, one of these possibilities is in the tabu list so the algorithm is no longer able to choose this move. The other three possibilities are again scored and one of them is chosen. This continues until a stopping condition is reached.

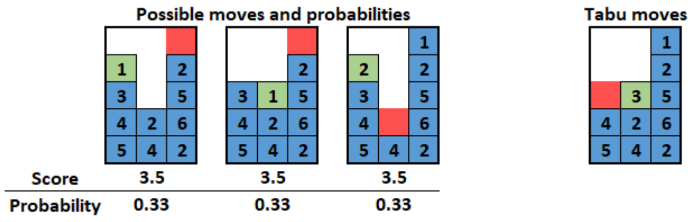


Figure 9: Tabu search new moves and scoring

7 Other algorithms

The tabu search algorithm was tested against other algorithms. All of these algorithms are run on the same instances and use the same container scores as the tabu search. These algorithms are explained in this section.

7.a Directed random search

The first algorithm the tabu search is tested against is the directed random search algorithm. This algorithm has a lot of things in common with the tabu search.

The directed random search is, just like the tabu search, a depth first search algorithm. The direction in which the algorithm goes is determined by the same chances as with the tabu search algorithm. This means the search space exploration mechanics are the same for both algorithms. This allows for a good comparison between them.

The stopping conditions for the directed random search are also the same as for the tabu search. The only difference between the tabu search and the directed random search are the tabu lists.

This algorithm is hit or miss. It is really dependent on the algorithm choosing the right moves at the right moments to achieve the best possible solution. Given enough computation time (unlimited) however, this algorithm should always find the optimal solution. This might however not be very efficient.

7.b Best only search

The best only search algorithm is a completely different algorithm. This algorithm takes a deterministic approach to choosing which move to do.
The idea of the algorithm is that it only allows moves which make the combined score better. As soon as there are no more moves which make the combined score better, the algorithm is stopped. The choice of which move to do is based on the criteria explained below. These criteria are in order.

1. The algorithm chooses the move which gives the maximum improvement in combined score.
2. If the are multiple moves which give the maximum improvement, the algorithm chooses the move which moves a container to the stack with the lowest priority. The stack priority is defined as the highest container time frame in the stack.
3. If there are multiple stacks with the lowest priority, the container from the highest stack is moved.
4. If there are multiple highest stacks, the container with the highest time frame is moved.
5. If there are multiple highest container time frames, the move which moves a container to the lowest stack is done.
6. If there are multiple moves which move a container to the lowest stack, the algorithm chooses the move with the maximum improvement it has found first.

Using these criteria, the algorithm tries to find the best solution. The algorithm is completely deterministic. This makes the algorithm very quick. On top of being completely deterministic, the algorithm often chooses 'smarter' moves than other algorithms, because the criteria on which it decides which move to pick are very logical and often amount to the 'best' move being done.

The disadvantage of this algorithm is it might get stuck in a local optimum very quickly. When one of the first few moves looks really good but actually leaves no more room for improvement, the algorithm does not continue. This would not happen with the tabu search and directed random search algorithm.

If the same instance as in Figure 6 is used, the algorithm would choose the move with the lowest score, in this case moving container 3 from row 2 to row 1. This move would allow for the biggest improvement (and only) improvement in score, and is thus the only allowed move.

8 Results

8.a Parameters

The algorithms are run and tested on the same instances and evaluated on the same statistics. The different parameters for the evaluations are discussed here:

8.a.1 Maximum number of iterations

The number of iterations outs itself in two things. Having a high number of iterations will amount to the algorithm finding the optimal solution more frequently. The downside of a lot of iterations is that the algorithms take a lot of time to run.

For the evaluation, a maximum of 100 iteration per instance in chosen. This allows for the algorithms to explore plenty of different configurations, while also allowing the tabu list to built up by itself.

8.a.2 Maximum number of iterations without improvement

The number of iterations without an improvement is important to make sure the computational time is reduced. Of course the algorithms should not be stopped to early to allow for more improvements.

For the evaluation, a maximum of 50 iteration without improvement is chosen. This allows for plenty of room for the algorithms to improve on previously found solution while also making sure they are not endlessly trying to improve on the optimal solution

8.a.3 Penalty

As discussed in the basic approach, a penalty is given for every move that is done. This penalty can be the difference between moving a container or stopping the iteration. Because of this, multiple penalties are evaluated.

- The first penalty that is evaluated is 0.2. For every move, a penalty of 0.2 is added to the combined score. This penalty is very low, thus often allows the algorithms to find a minimum grid score.
- The second penalty is 0.5. This score is already somewhat higher. With this score, not every move which gives an improvement in grid score is done.
- The last penalty that is evaluated is 1. This is a very high penalty, thus it might be very hard for the algorithms to find suitable moves. Only the best moves will have a place in the solution, and it might even be possible no moves are done at all.

These penalties show multiple ways for which the algorithms can be compared to each other. The penalties will show what different decisions the algorithms make at certain configurations.

8.b Penalty 0.2

Container grid		Average Improvement			% Improvement		
Width	**Starting Score**	**Tabu**	**Random**	**Best**	**Tabu**	**Random**	**Best**
5	5.400	-1.712	-1.683	-1.219	-39.3%	-38.6%	-28.5%
6	6.206	-1.895	-1.848	-1.614	-37.1%	-36.4%	-30.1%
7	7.219	-1.917	-1.921	-1.800	-34.5%	-34.1%	-31.7%
8	8.104	-1.933	-1.890	-1.940	-32.0%	-31.5%	-31.1%
9	9.135	-2.148	-2.157	-2.577	-30.7%	-30.7%	-34.0%
10	10.040	-2.087	-2.118	-2.706	-27.9%	-28.4%	-34.3%

Table 1: Score comparison for penalty 0.2

The difference between the different algorithms is clearly visible in the results. In Table 1, the score improvements of all three algorithms are compared. For all three algorithms, a clear trend is visible. As the width of the container grid increases, so does the score improvement. Bigger instances are able to receive greater improvements than smaller instances. This is logical, because the bigger instances have a higher starting score and often leave more room for improvement. The strongest trend is present with the best only search algorithm. This can be explained by the fact that the best only search is able to find the 'smartest' moves more easily, while the tabu and random algorithms may not choose for these smarter moves. Another important thing to note, is that with smaller instances, the random and tabu search receive great improvements. As the width of the container grid increases however, the best only search surpasses the other algorithms. This can be explained by the by the fact that the random and tabu search algorithms require more computational time. With the small instances, the tabu and random search perform better than the best only search. This could mean the computation time for these instances is sufficient and the algorithms are able to find good solutions. For the bigger instances the computation time does not seem to be sufficient. The best only search shows a steeper improvement while the random and tabu searches have a relatively slow increase.

For the percentage increase, a different trend is visible. The complete opposite of the average improvement seems to be happening. When the instances get bigger, the percentage improvement of the tabu and random search algorithm show a decrease. This can be explained by the fact that these algorithms require a lot of computation time for big instances, which is not given. Where the total improvements do increase, the percentage improvements decrease. The opposite happens for the best only search. As the instances get bigger, a greater improvement is achieved. For bigger instances, a greater improvement is usually possible. With the relatively 'smart' decision of the best only search, the percentage improvements increase as the size of the instances increase.

	Moves		
Width	**Tabu**	**Random**	**Best**
5	5.971	5.854	2.113
6	6.763	6.458	2.492
7	6.571	7.021	2.717
8	7.158	6.850	2.904
9	7.771	7.404	3.667
10	7.638	7.600	3.750

Table 2: Move comparison for penalty 0.2

For the moves comparison shown in Table 2, there is a big difference visible between the algorithms. The tabu and random search algorithms on average use a lot of moves to receive the final instance score. For the best only search, only around half of these moves is used. This can be explained by the fact the best only algorithms takes a better look at the good moves and then chooses the seemingly best move. The algorithm is also able to choose a efficient sequence of moves to receive the final score. The tabu and random search algorithms need more moves. This is logical, because these algorithms are not as 'smart' as the best only algorithm. While some of their moves might be better, a lot of less useful moves can be done in between with a relatively low penalty. This is not possible for the best only search.

8.c Penalty 0.5

Container grid		**Average Improvement**			**% Improvement**		
Width	**Starting Score**	**Tabu**	**Random**	**Best**	**Tabu**	**Random**	**Best**
5	5.400	-0.565	-0.569	-0.469	-13.4%	-13.7%	-11.2%
6	6.206	-0.675	-0.658	-0.773	-13.3%	-12.8%	-14.3%
7	7.219	-0.735	-0.742	-0.852	-13.3%	-13.2%	-14.8%
8	8.104	-0.713	-0.717	-0.992	-12.3%	-12.3%	-16.1%
9	9.135	-0.819	-0.798	-1.344	-11.9%	-11.6%	-18.0%
10	10.040	-0.817	-0.827	-1.496	-11.2%	-11.5%	-19.1%

Table 3: Score comparison for penalty 0.5

For the penalty of 0.5, a same trend is visible as with the penalty of 0.2. All algorithms shown an increase in score improvements as the size of the instances increase. The best only search algorithms shows a steeper trend than the tabu and random search algorithms. For larger instances, the best only algorithm surpasses the tabu and random algorithms.

For the percentage improvement, as with the average improvement, the same happens as with penalty 0.2. The percentage improvement is decreasing for the tabu and random search, while is it increasing for the best only search. This can again be attributed to the computation time given to the algorithm. This is not a problem for the best only search.

The difference however with the penalty of 0.2, is the size of the improvements. It is clearly visible that the average improvement with penalty 0.5 is lower. This is logical, because the algorithm needs to find a very good move to justify carrying it out. This is because the penalty is so high. Moves also have greater contribution to the combined score, thus the smaller improvement. The same observation is visible at the percentage improvements.

	Moves		
Width	**Tabu**	**Random**	**Best**
5	1.358	1.567	0.867
6	1.392	1.625	1.454
7	1.600	1.717	1.621
8	1.479	1.696	1.946
9	1.754	1.775	2.642
10	1.742	1.800	2.946

Table 4: Move comparison for penalty 0.5

As previously shown for the score improvement, the same things happen for the number of moves. When comparing the algorithms in Table 4, the same trends as with penalty 0.2 are visible. The number of moves is increasing as the width of the instances increases. The best only search shows greater increases than the random and tabu search algorithms. This can again be explained by the computation time needed for the tabu and random algorithms to achieve good score improvements with relatively few moves. The best only algorithm does not need this extra time and chooses the 'smartest' moves to achieve the best scores of all three algorithms for big instances.

8.d Penalty 1.0

Container grid		Average Improvement			% Improvement		
Width	**Starting Score**	**Tabu**	**Random**	**Best**	**Tabu**	**Random**	**Best**
5	5.400	-0.029	-0.029	-0.027	-0.281%	-0.281%	-0.249%
6	6.206	-0.048	-0.050	-0.046	-0.434%	-0.449%	-0.420%
7	7.219	-0.040	-0.038	-0.035	-0.377%	-0.349%	-0.333%
8	8.104	-0.017	-0.021	-0.017	-0.128%	-0.174%	-0.110%
9	9.135	-0.029	-0.025	-0.023	-0.209%	-0.159%	-0.150%
10	10.040	-0.025	-0.023	-0.023	-0.167%	-0.144%	-0.157%

Table 5: Score comparison for penalty 1.0

For the penalty of 1.0, a different observation is made. As is shown in Table5, for every algorithm which is run with this penalty, hardly any improvement is achieved. It is important to note that the best only algorithm performs worst of all algorithms for every width. Because the random algorithms are not allowed to do a lot of moves here (because the stopping condition is reached quickly), a lot of configurations can be explored and a better solution is found than doing just the 'best' move.

The percentage improvement is decreasing for all three algorithms. This could mean that even for the big instances, there are nearly no good moves left using this penalty. This means the relative improvements decreases as the container grid width increases.

	Moves		
Width	**Tabu**	**Random**	**Best**
5	0.050	0.050	0.042
6	0.067	0.071	0.058
7	0.054	0.046	0.042
8	0.029	0.038	0.021
9	0.046	0.046	0.038
10	0.050	0.050	0.038

Table 6: Move comparison for penalty 1.0

For the number of moves used with penalty 1.0, something else is visible The best only search consistently uses the fewest moves. This could mean the tabu and random search find different moves to receive the slightly better result and the seemingly 'best' move does not necessarily give the biggest improvements.

8.e Tabu versus Random

As seen in the previous comparisons, for each penalty the tabu and random search algorithms seem to perform relatively equal. As a better comparison is needed for these two algorithms, they are run again using the same parameters, but with more iterations. Because of the long computation time, sometimes as long as two days, this was only done for container grids with a width of five and six. For good comparison of the algorithms, a penalty for each move of 0.2 was used.

Container grid	**Tabu - Random**			**Tabu - Random %**		
Width	**Score**	**Moves**	**Iteration**	**Score**	**Moves**	**Iteration**
5	0.059	-0.458	-0.330	-1.353%	6.871%	2.339%
6	0.032	-0.306	0.098	-0.770%	5.342%	0.617%

Table 7: Comparison of tabu search versus random search

In Table 7, it is visible the tabu search performs just a slight bit better than the random search algorithm in terms of score improvement. For both container grid widths, there is a slight improvement. This improvement however is very small, around one percent. On the other side, we see the tabu search algorithm uses more moves and more iterations. This can be explained by the fact that the score of tabu is lower. To achieve a lower score, it might be possible that more moves were used to achieve this score. The same is possible for the iterations. More iterations are needed to achieve this slightly higher score. The differences between tabu search and directed random search stay really small nonetheless.

8.f Best Tabu search

In an attempt to achieve the best score with these algorithms, a combination of best only search and tabu search was made. First the best only search algorithm is used. When it is done, the tabu search is ran on the grid achieved thus far. This gives best only the opportunity to choose the 'smartest' moves in the beginning, while tabu search is able to improve this setup even further using the directed random search and tabu lists. The best tabu search is compared to both the default tabu search and the best only search algorithm. Because of the minimal improvements for penalty 1.0, the algorithms are only compared on penalties of 0.2 and 0.5.

8.f.1 Penalty 0.2

After running the algorithms, a first look is had at what the best tabu algorithm really does. In table 8, a few things stand out. When best tabu is compared to the best only algorithm, we see for each increase in width, a greater improvement is achieved. This is logical, as for bigger instances it can be easier to find moves which give an improvement. The improvements however, are not very big. For the percentage improvement here, no real pattern appears. For the number of moves, it is clear that the best tabu every time the best tabu algorithm does more moves. This is of course the way it should be, as the tabu part only starts after the best only algorithm is done. We can see there are around five extra moves per instance, however no real pattern comes up. For the percentage move difference between the algorithms, the same result comes out. There seems to be a small trend downwards as the width of the container grid increase, however this trend is not continuous as width six and seven have different values. The only real difference seems to be a widths nine and ten, where the value is a little lower. This could be caused by the limited computing time given to the tabu algorithm. This was previously discussed for the standard tabu algorithm.

Container grid	**Best - Best Tabu**		**Best - Best Tabu %**	
Width	**Score**	**Moves**	**Score**	**Moves**
5	0.978	-5.096	-30.4%	235%
6	1.045	-6.725	-29.3%	198%
7	1.049	-4.908	-30.5%	238%
8	1.113	-5.988	-27.9%	227%
9	1.318	-4.858	-31.5%	178%
10	1.245	-4.733	-28.3%	166%

Table 8: Best Tabu versus Best for penalty 0.2

When comparing the tabu versus the best tabu algorithm in Table 9, a clear improvement is shown, especially for the larger instances. It is clear both for the absolute improvement and the percentage improvement, the bigger the instances get, the greater the improvement is. The number of moves however shows some slightly different statistics. The number of moves improvement seems to be a little lower for the large instances. This could be explained again by the computing time. The best tabu and the tabu algorithms seem to have to same difficulties when solving the larger instances. It is clear however that the tabu achieves a better score in slightly more moves when compared to the standard tabu search algorithm.

Container grid	**Tabu - Best Tabu**		**Tabu - Best Tabu %**	
Width	**Score**	**Moves**	**Score**	**Moves**
5	0.486	-1.238	-17.3%	79%
6	0.764	-2.454	-20.9%	32%
7	0.933	-1.054	-26.2%	55%
8	1.120	-1.733	-26.4%	43%
9	1.747	-0.754	-32.7%	61%
10	1.864	-0.846	-33.2%	61%

Table 9: Best Tabu versus Tabu for penalty 0.2

8.f.2 Penalty 0.5

When looking at the statistics for the penalty of 0.5, it is obvious the improvements are smaller and fewer moves are used. When comparing the best tabu against the best only algorithm, mostly the same patters arise. When looking at the average score improvement, there is a clear upwards trend when the width increases. For the improvement in the moves, just as with the penalty of 0.2, no clear trend is visible. When looking at the percentage score improvement however, the same trend as with the average improvement becomes visible. This was not the case for penalty 0.2. This could mean that the best only algorithm still is not very optimal for the bigger instances, and the tabu search is able to find good improvement for larger instances. For the larger penalty scores, fewer computing time is needed because the algorithm is stopped earlier. This could be the cause for the trend in percentage improvements as the container grid width increases.

Container grid	**Best - Best Tabu**		**Best - Best Tabu %**	
Width	**Score**	**Moves**	**Score**	**Moves**
5	0.485	-1.254	-13.3%	70%
6	0.721	-1.225	-16.1%	74%
7	0.760	-1.250	-18.1%	70%
8	0.898	-1.146	-18.5%	61%
9	1.235	-1.271	-22.4%	61%
10	1.340	-1.283	-23.0%	59%

Table 10: Best Tabu versus Tabu for penalty 0.5

For the comparison with tabu search, for the score improvement the same trends as when comparing to best only are visible. For the number of moves however, a very clear trend shows up. When the instances get bigger, increasingly more moves are used. Both in average number of moves and in the percentage improvement. The penalty score of 0.5 gives the algorithm a better shot at finding a good solution in more moves. The computing time of big instances was a big issue with the tabu search, but this no longer seems to be the case for the best tabu search with a penalty of 0.5.

Container grid	**Tabu - Best Tabu**		**Tabu - Best Tabu %**	
Width	**Score**	**Moves**	**Score**	**Moves**
5	0.381	-0.763	-10.7%	67%
6	0.819	-1.288	-16.5%	93%
7	0.877	-1.271	-18.6%	92%
8	1.177	-1.613	-21.1%	115%
9	1.760	-2.158	-26.4%	145%
10	2.019	-2.488	-28.6%	172%

Table 11: Best Tabu versus Tabu for penalty 0.5

9 Conclusion

In this paper, the preprocessing of the stochastic container relocation problem is explored using a variety of algorithms, in particular tabu search. The quality of the solutions is very reliant on the size of the instance. For small instances, the tabu search was able to find very good solutions, while for bigger instances, computation time rose by a lot and solution quality decreased. For the large instances, the best solutions were found with the best only algorithm. The tabu search and directed random search algorithms were also thoroughly compared. The differences in solution between the algorithms was very low, even when as much as 50 runs were used. The solutions for tabu search did show a small improvement.

In an attempt to achieve even better results, especially for the larger instances, tabu search and best only search were combined. This gave a great improvement, especially for the biggest instances. It was clearly shown that the tabu search, where non-improving moves were allowed, was in the end able to improve the solutions by a large amount. This best tabu algorithm was the best algorithm found in this paper.

An important improvement of the tabu search algorithm is the computation time. The tabu search works better when the evaluation of the container setups is faster. In this way, more iterations and runs can be done. Another point of investigation is the tabu list size. In this paper, only set sizes for the tabu list were used, while it might be better to have a varying size of these lists to allow for larger tabu list for larger instances. That is however out of the scope of this paper. The best tabu search already gave very good results and with more work, even better solution can be found, especially for the largest instances.

References

[1] Galle, V., Borjian Boroujeni, S., Manshadi, V. H., Barnhart, C., Jaillet, P., The Stochastic Container Relocation Problem, 2018, Transportation Science, September-October 2018, 1035-1296,
`https://arxiv.org/pdf/1703.04769.pdf`

[2] Gendreau, M., Potvin, J.-Y., Tabu search, 2010, Handbook of Metaheuristics,
`https://www.springer.com/cda/content/document/cda_downloaddocument/9781441916631-c1.pdf?SGWID=0-0-45-993383-p174027062`

[3] Simulated Annealing
`http://www.cs.nott.ac.uk/~pszgxk/aim/notes/simulatedannealing.doc`

[4] Expósito-Izquierdo, C., Melián-Batista, B., Moreno-Vega, M., 2012, Pre-Marshalling Problem: Heurstic solution method and instance generator. Expert Systems with Application. 39(9): 8337-8349.

[5] Problem instances
`http://crp-timewindow.blogspot.com/`