Container stacking and retrieval strategies for problems with full information

Thesis proposal

June 25, 2012

Radek Timmer

Supervisor: Prof. Rommert Dekker

Co-reader: Dr. Y. Zhang

Abstract

In this thesis I examine stacking problems with both known and unknown arrival and departure orders. I investigate different stacking strategies that attempt to minimize the number of reshuffling moves that have to be performed to move all containers in and out of the stacking area. First I generate the sequences of arriving and departing containers, which will be used to test the heuristics. Next I implement a brute force approach that checks every possible unique combination of placing the containers and reshuffling them. This method by definition finds the most optimal solution of a problem, but will only be practical for small problem sets as the calculation time increases exponentially with larger problem sets. Using the results from this method I create a function the calculates the minimum number of reshuffling moves that have to be performed for a given problem. Finally some more practical heuristics are developed and tested.

Contents

1 Introduction

In 2006 the Emma Maersk was launched, the first in a series of 7 E-class ultra large containers ships that can carry almost 15.000 containers. These ultra large container ships are part of a trend towards ever larger vessels. There are now orders out for a fleet of even larger container ships that can carry up to 18.000 containers. Each year more than one hundred million containers are transported across the globe, with volumes expected to grow even bigger (Vos 2012).

As there is limited space on the dockyard, containers are stacked on top of another as they await their respective pickup. This gives rise to the question of what the best order is in which to stack these containers. An inefficiently stacked set of containers can mean a lot of time is wasted 'digging out' containers that sit below other containers. The same problem applies to the ship itself. It too might have to move containers meant for another port that are sitting atop containers that have to be unloaded now. Doing this as efficiently as possible is of great economic importance, as idle ships cost thousands of dollars in lost revenue.

Many papers have investigated stacking strategies and there are of course computer programs that plan and operate the stacking of containers, running in virtually every seaport. All of these programs deal with the more difficult problem of planning containers under a great amount of uncertainty. My research will limit itself to the simpler deterministic model in which the order in which containers arrive and leave the dockyard is known beforehand. The hope is that an investigation of these simpler deterministic problems will yield results that can give us an insight into possibly effective strategies in real world problems. There might also be other industrial processes, where the arrival and departure order is known beforehand, where this research could be applied.

In order to find an effective solving strategy I will investigate the possibility of finding optimal solutions, when time is not an issue. Due to the complexity of these types of problems, this will have to be a brute force approach that checks all the possible permutations of possible solutions. Having certain problems solved absolutely might then help us to write down some general rules about the minimum number of required extra moves for a given problem. This minimum will give us a lower bound on the solution beyond which we do not need to search for solutions. Finally I will investigate a number of practical solving strategies.

2 Methods

2.1 Data Generator

The first part of this research will be to create a function that can generate a list of randomly arriving and departing containers. These generated problems can then be used to test and evaluate possible solution algorithms. These generated *problems* will take the form of a matrix D with two columns. An example arrival and departure sequence is shown below. Each row represents the arrival or departure of a single container. Where the first column of Matrix D indicates whether there is an arrival (represented by $D_{i,1} = 1$) or a departure (represented by $D_{i,1} = 2$). The second column indicates the container number. For example in the matrix below, $D_{1,1} = 3$ represents the container number of the first event. Each container will have such a unique number, which besides identifying the container also represents the order in which the containers will depart. Where the container with the lowest number will depart first, followed sequentially by the remaining containers. So in the first row in the table below $D_1 = [1\ 3]$ represents the arrival of container 3. The next event $D_2 = [1\ 1]$ is the arrival of container 1. The event after that $D_3 = [2\ 1]$ is the departure of container 1.

Example Generated arrival and departure sequence $D_{i,j}$

1	3
1	1
2	1
1	2
2	2
2	3

$D_{i,j}$, $i = 1 \ldots 2n$

$j = 1\ 2$

We will want the ability to vary certain aspects of Matrix D, to allow us to adjust the difficulty of the generated problem. Firstly we will want to vary the number of arriving containers and subsequently departing containers. Therefore the first input parameter n, indicates the number of arriving containers. The example above has a size of n = 3, meaning 3 arrivals and 3 departures.

Next we will need to put a limit L on the number of containers in the system at the same time. This has two reasons. It will allow us once again to vary the difficulty of the problem, as problems with more containers in the system simultaneously will in general be more difficult. But it will also allow us to make sure the problems are feasible. For if we allow all the available slots to be filled with containers, we might not be able to retrieve containers buried below other containers. Take the situation in Table 1, where there are 9 containers that fill all the spaces in 3 stacks. If we assume we cannot stack 4 containers on top of each other, this problem cannot be solved. Therefore there has to be a limit on the number of containers that can be in the system at the same time. But even leaving one or two spots open might not be enough to retrieve a container as illustrated in Table 2. We are again unable to access container 1 that has to leave first.

Table 1

7	8	9
6	5	4
1	2	3

Table 2

7	8	
6	5	4
1	2	3

Thus the container limit parameter L will be allowed to vary, but as a general rule the maximum number of containers that can be present at the same time for a problem to still be feasible is given by the following function:

$$L = (S \times H) - H + 1$$

where S = number of stacks

H = maximum height of the stacks

Finally we will program a parameter P, which determines the likelihood of having an arrival over a departure. This will allow us to create more difficult problems, as it enables us to increase the chance of having a larger number of containers in the system at the same time. This parameter can take the values between 0 and 1. A the closer the parameter is to 1, the higher the chance the next event will be an arrival. Conversely, the closer the parameter is to 0, the higher the chance that the next event will be a departure.

In order to evaluate the difficulty of a generated dataset, we will use two statistics. First we calculate the average number of containers in the system. This is the average of the number of containers in the system after an arrival or departure. A high average number of containers indicates the problem is probably difficult to solve and may require a large number of reshuffling moves. The second statistic is the maximum number of containers present at any one time during the arrival and departure process. Again a higher maximum means an increased chance for a more difficult problem. Later on, we will add a third statistic that calculates the minimum number of reshuffle moves that definitely have to be performed to solve the problem.

To summarize, to test any algorithms or heuristics we will need test data. A set of randomly generated arrivals and departures. We want to be able to specify the length of the dataset, specifically the number of containers that will arrive. And we also want to specify a parameter that will enable us to increase the average number of containers present . As well as the maximum number of containers that can be present at any one time. When any of the above three factors is increased, the chance for a more difficult problem is increased.

To accomplish this we generate two sets of random numbers between 0 and 1, R_1 and R_2. The first set R_1 will determine whether a container will arrive or depart. The second set R_2 will determine in the case of a departure, which container will leave. We also generate a list of temporary container numbers.

Then by use of a formula we determine whether there will be an arrival or departure. The formula takes the first number from the first set of generated random numbers R_1 and compares it to the parameter P described above. When the random number is smaller or equal to parameter P, a container enters the system. The first container from the list of temporary container numbers is added to a list of containers that have arrived in the system, but not yet departed. When this number is larger than the parameter, a container is picked to leave the system. This is done by taking a number from the second list of generated random numbers R_2 , multiplying it by the total number of containers presently in the system and rounding up. This will give the index of the container that will depart. This process is repeated until all generated containers have arrived and departed.

This formula has a couple of restrictions:

- When there are no containers present in the system, there has to be an arrival.
- When the maximum number of allowed containers is reached, there has to be a departure.
- When all the containers generated have entered the system, there can be no further arrivals.

Finally the temporary container numbers are renamed. The container leaving first will be renamed 1, the container leaving second will be renamed 2, and so forth.

2.2 Brute Force

The second part of this thesis is the implementing of a brute force approach, that will solve a given container arrival/departure problem by testing all possible solution combinations. Although a brute force strategy is not practical as an optimization strategy, it can be instrumental in creating and evaluating optimization strategies or in creating a function that determines the minimum number of moves. By implementing a brute force strategy we will be able to absolutely solve a number of small problems, which can then be used to help create a function that calculates the minimum number of reshuffling moves that will have to be performed on a problem. Such a function will allow our eventual algorithm to stop when it reaches that minimum and not waste time exploring solutions that are not attainable. Having a number of problems solved absolutely can also give us an insight into what an optimal solution for particularly difficult problems might look like. And finally it will help us evaluate the performance of other algorithms by seeing how much they deviate from the absolute solution calculated by this brute force approach.

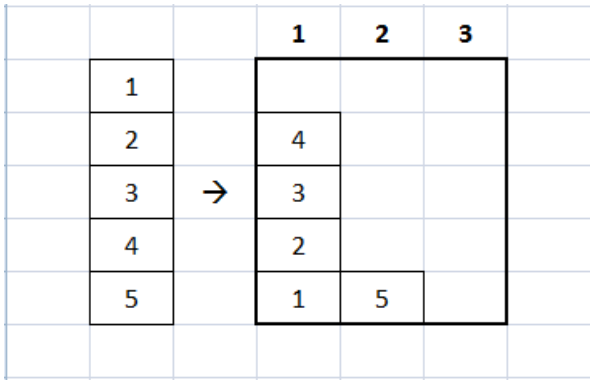
There are three levels of complexity in the solutions that will be checked by the brute force algorithm. First we have to cycle through all the different sets of position vectors that determine in which stacks the arriving containers will be placed. secondly for every solution the algorithm will have to run through all the sets of different combinations of locations where a reshuffle can take place. The last level of complexity runs through the set of possible reshuffles. And finally every possible solution will have to be evaluated by some function that determines if it produces a valid outcome.

The first level we modeled contains the different sets of positions, the arriving containers are assigned to. Every set will be a vector of integers *n* long with numbers ranging from 1 through K. Where n represents the number of arriving containers and K represents the number of different stacks the container can be placed in. For example a set of 5 arrivals, where the first 4 containers are put in the first stack and the last container in the second stack will be represented by the following vector:

[1 1 1 1 2]

Resulting in the following situation:

Figure 1



Where the left stack represents the arriving containers, with container 1 arriving first. The right part of the diagram represent the 3 stacks where the containers will be deposited in the order shown above.

In order to model this we could generate all the possible numbers n long with base K. For example for n = 4 and K = 3, we would get the following list:

1 1 1 1

1 1 1 2

1 1 1 3

1 1 2 1

1 1 2 2

1 1 2 3

1 1 3 1

…

3 3 3 2

3 3 3 3

This would represent all the different ways the 4 arriving containers can be placed on the 3 available slots. This set has 3*3*3*3 = 81 different vectors. However, there are a great number of vectors that for the intents and purposes of this research are exactly the same. In the list above the second and third vectors represent the exact same situation as the specific location of the containers is not important. The only thing of consequence is that the first 3 containers are deposited on atop the other in the first stack and the last next to it, leaving one space open. These two vectors would yield the same results and we therefore want to eliminate one of them, saving us from having to check the same solution twice. There are many other vectors that can be considered equal as well. The first and last vectors for example are identical as well and so are the fourth and seventh. In fact out of the 81 vectors there are only 14 that are truly unique. So by filtering these out we would reduce the number of solutions that have to be checked by 83%.

As the number of different solutions runs into the millions very quickly as N and K increase, we cannot practically generate a list of all possible combinations and then eliminate the doubles. In fact the strains on a computers available memory prevent us from generating a list than long anyway, so we had to find a function that will generate the next solution in the list given only the previous solution. Allowing us to cycle through all the possibilities without having to maintain a huge lists of data. This function takes the previous vector, staring with the initial vector 1 1 1 1, and begins at the last digit in the vector and then works its way back. For every number it calculates a Coefficient Mn, which is equal to the minimum of K and the maximum of all the previous numbers in the vector plus one. So with:

V = 1 1 1 2 with i = 1-4

Mn(i) = min [K , max[V(1: (i-1))] + 1] = 2

This number represents the highest number occurring in the vector until that point plus one, with a maximum of K. If this number Mn is larger than the number at point i the number at point I is increased by one and we stop. If the number Mn is not larger, the number i is set to 1 and we move to the number i-1. In this case at i=4 Mn is not larger than V(4)=2, so we set V(4) equal to 1 and move on to i3. At i3 Mn is equal to 2, which is larger than V(3)=3. Therefore we increase V(3) by one and stop the procedure. This results in a vector equal to:

V= 1 1 2 1

The idea behind this procedure is that we should only increase the last number until it is no more than one higher than the maximum of the previous numbers. If not, we should move on the the previous number in the list. So in the last example it would not have been useful to raise the last digit to 3, making the vector 1 1 1 3. Because this is equal to the previous vector 1 1 1 2. When this simple procedure is followed we are left with only the unique vectors.

In this example the full list of 14 vectors is:

1 1 1 1	1 2 1 3
1 1 1 2	1 2 2 1
1 1 2 1	1 2 2 2
1 1 2 2	1 2 2 3
1 1 2 3	1 2 3 1
1 2 1 1	1 2 3 2
1 2 1 2	1 2 3 3

The totals of these vectors correspond to the horizontal sums of Stirling numbers of the second kind in combinatorics, which is the number of ways to partition a set of n objects into k non empty subsets. Where total number of partitions added over the k subsets 1:K is equal to the number of different solution vectors we calculated above.

In the table below the number of possible solutions that have to be checked for different numbers of arriving containers in a system that has 4 stacks is shown in column 2. The third column shows the factor increase resulting from one additional container. The fourth column shows the percentage of unique solutions that will be checked compared to the total number of ways the containers can be positioned. Here we see the savings continue to increase, but by ever smaller increments, as the number of arrivals increases.

Table 3

	n	possible solutions	factor increase	Percentage of 4^n
number of arriving containers	1	1	1	0.2500
	2	2	2.000	0.1250
	3	5	2.500	0.0781
	4	15	3.000	0.0586
	5	51	3.400	0.0498
	6	187	3.667	0.0457
	7	715	3.824	0.0436
	8	2,795	3.909	0.0426
	9	11,051	3.954	0.0422
	10	43,947	3.977	0.0419
	11	175,275	3.988	0.0418
	12	700,075	3.994	0.0417

The second level of complexity are the reshuffle times. The number of reshuffles will be fixed, but the locations will vary. For example we can reshuffle one of the containers after the first container arrives or after the second container leaves. After each container event one or more reshuffles are possible. Where an event is considered to be either the arrival or departure of a container. To model this we will generate all possible solutions, where each solution consists of a vector of numbers equal in length to the total number of reshuffles and each number m in that vector indicates that after the mth container arrival or departure event one container is moved. It is possible that more than one container is moved after a container event, this will be represented in the vector by all numbers m equal or larger than the number of the container event n but smaller than the next integer n+1. For example the vector [2 5 5¼ 5½ 5¾ 8] means that after the second event one container is moved, then four containers are moved after the fifth event and one is moved after the eight event.

Of course it doesn't make sense to move a container directly after the first arrival or to move more than one directly after the second arrival, so our model will not contain the possibility to do a reshuffle move after the first arrival and only have the possibility to move once after the second arrival. For example the different possible numbers representing the possible times of reshuffle in a solution for n=6 arrival events and G = 4 possible moves are:

2 3 3.5 4 4 1/3 4 2/3 5 5.25 5.5 5.75 6 6.25 6.5 6.75

From these all possible combinations of 4 are then selected, making sure not to include non whole numbers if the previous numbers equal or larger than the whole base are not included. For example we do not want to include 5.5 if 5.25 and 5 are not included. As the following vectors would be equal:

[2 3 5 5.25] and [2 3 5 5.5]

Both indicate one move after events 2 and 3 and two moves after event 5. Only the first vector is correct in our model to avoid duplicating answers.

The two tables below show the number of possible combinations of reshuffle locations for different numbers of arriving containers and number of reshuffles and their relative increase over the previous number of containers respectively. In the Table on the left it obviously doesn't make sense to calculate the number of different times a reshuffle can take place when there is only one container arriving. Similarly, when only two containers arrive it doesn't make sense to move more than once. All these irrelevant results are represented by NR. We can see that the increase in possibilities is fairly gradual as the number of containers increases. However, there is an exponential growth in possibilities when the number of reshuffles is increased.

Table 4

Number of possible reshuffle locations					
		number of reshuffles			
	n	1	2	3	4
number of arriving containers	1	NR	NR	NR	NR
	2	1	NR	NR	NR
	3	2	2	NR	NR
	4	3	5	6	NR
	5	4	9	15	20
	6	5	14	29	49
	7	6	20	49	98
	8	7	27	76	174
	9	8	35	111	285
	10	9	44	155	440
	11	10	54	209	649
	12	11	65	274	923

Table 5

Factor increase locations over previous n					
		number of reshuffles			
	n	1	2	3	4
number of arriving containers	1	NR	NR	NR	NR
	2	NR	NR	NR	NR
	3	2.00	NR	NR	NR
	4	1.50	2.50	NR	NR
	5	1.33	1.80	2.50	NR
	6	1.25	1.56	1.93	2.45
	7	1.20	1.43	1.69	2.00
	8	1.17	1.35	1.55	1.78
	9	1.14	1.30	1.46	1.64
	10	1.13	1.26	1.40	1.54
	11	1.11	1.23	1.35	1.48
	12	1.10	1.20	1.31	1.42

The last level of complexity are the different combinations of actual moves. Every number in a vector in the previous section indicates that a move occurs at that point, but what that move actually is represented in this last level. For every move in the previous section we have two numbers in this section. The first representing the number of the stack from which the top container will be removed and the second indicating onto which stack that container will be placed. A sample vector for the vector in the previous section would be:

[1 2 1 2 1 2 1 3]

Which would represent that a container from the first stack would be moved to the second stack after the second, third and fifth events and an additional container will be moved from the first to the third stack after that.

The algorithm very simply cycles through the possible combinations of moves, resulting in (K * (K-1))^G possible combinations. Where K equals the number of stacks and G the number of total moves. The number of possibilities for the case of 4 stacks is shown in the table below. This again indicates that an increase in the number of reshuffles causes a rapid increase in the complexity of the problem. The number of possibilities does not increase on the count of this last level of complexity if the number of arriving containers is increased.

Table 6

	number of ways to reshuffle			
number of reshuffles	1	2	3	4
possibilities	12	144	1728	20736

Having these three separate levels of complexity means that the number of possible solutions will increase very rapidly as both the number of arriving containers and the number of reshuffles increases. So finally in order to get the total number of possibilities, the totals of the previous three sections are combined to create the following table. Here we see the number of solutions that will have to be checked for a given number of arriving containers and number of reshuffles.

Table 7

		number of possible solutions				
		0	1	2	3	4
number of arriving containers	1	1	NR	NR	NR	NR
	2	2	24	NR	NR	NR
	3	5	120	1,440	NR	NR
	4	15	540	10,800	155,520	NR
	5	51	2,448	66,096	1,321,920	21,150,720
	6	187	11,220	376,992	9,370,944	190,003,968
	7	715	51,480	2,059,200	60,540,480	1,452,971,520
	8	2,795	234,780	10,866,960	367,061,760	10,084,538,880
	9	11,051	1,060,896	55,697,040	2,119,670,208	65,308,757,760
	10	43,947	4,746,276	278,448,192	11,770,764,480	400,965,396,480
	11	175,275	21,033,000	1,362,938,400	63,300,916,800	2,358,792,057,600
	12	700,075	92,409,900	6,552,702,000	331,465,910,400	13,398,965,049,600

Finally the arrival vector, the move locations vector and the move vector are entered into an evaluation function that checks whether the solution is feasible.

2.3 Minimum moves function

Having a function that can quickly calculate the minimum number of extra moves required for a given arrival and departure dataset, is an important part in implementing an optimization strategy. Such a function is the only way to tell the optimization strategy it has reached an optimal or close to optimal solution. It will also allow us to not waste time looking for solutions that are unattainable.

We are looking for a function that best approximates the number of reshuffling moves required, without going over. Such a function will have to give a strict minimum, where the actual number of reshuffling moves can be higher, but never lower.

In order to find this minimum we first recognize that whenever a container that has to leave after a container below it in a stack, that container will have to be moved before it departs the system. In our model a container with a lower number will always leave before a container with a higher number. In all three situations sketched below, the number 7 container blocks the number 1 container from moving and will have to be moved before 1 can leave.

		7			7		
7	3	8	3		1	3	
1	4	1	4		8	4	

Realizing this we can quickly calculate an absolute minimum number of moves that will be required to remove the containers in a given situation. For example in the three situations below we will need at least 4, 0 and 1 reshuffling moves respectively:

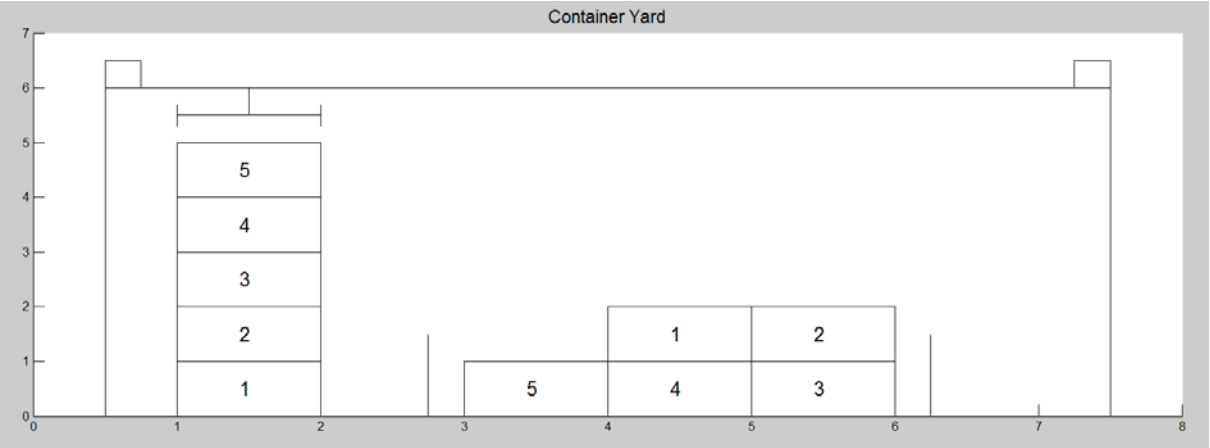
6							
1	7	1					
8	5	2	4		4		
3	4	3	5	6	1	2	3

This can easily be determined on a stack by stacks basis. Where staring at the bottom we find the first non-decreasing container and for every container above and including that container that has a higher number than the one below it, we add one extra reshuffling move. For example above in the first column on the left, only the containers 6 and 8 block container 3. Container 1 will leave before this becomes an issue and is not counted. The fact that 6 also blocks container 1 is irrelevant, as a single container can only create one extra reshuffling move.

The next part of the function in based on the idea that containers arriving in exactly reversed order from which they are leaving will not create a need for reshuffling moves, but containers arriving in the same order as they will leave does create a need for possible reshuffling. When containers arrive in the reversed order from how they will leave, we can create neat stacks where the departure

numbers on the lower containers in a stacks are always higher. This means the top containers will always leave first, resulting in no need to reshuffle anything. However, when the containers arrive in the same order as how they will leave (i.e. increasing order) we run into trouble. In order to avoid placing a container with a higher number on top of a container with a lower number, we have to put each arriving container in a separate stack. When the list of arriving containers in ascending order becomes larger than the number of available stacks, containers will have to be put on containers with a lower number. Each additional container that arrives in ascending order without having one of the previous containers in the list leave, results in another block. This concept is illustrated below.

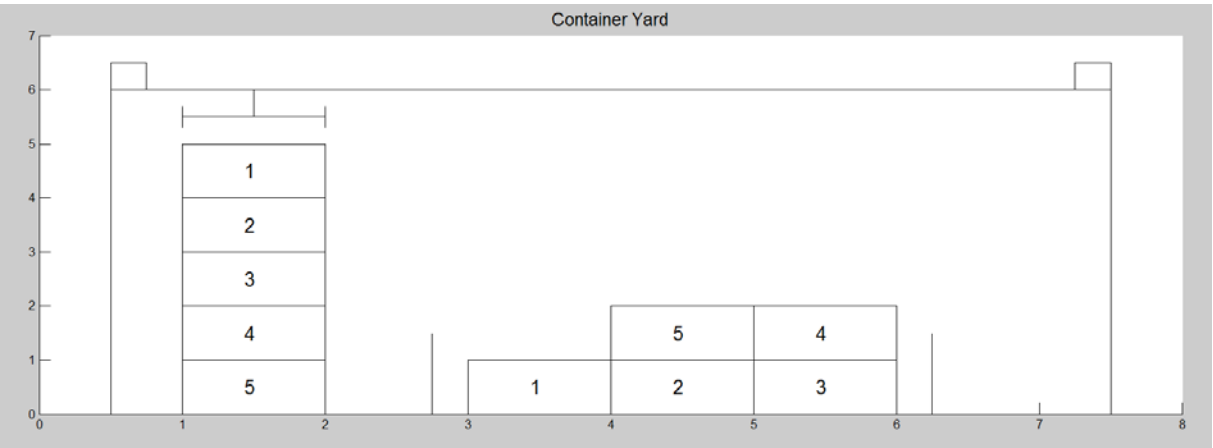
Figure 2



This picture comes from an animation tool I developed in Matlab to simulate solutions to retrieval problems.

This first illustration pictures the situation where the containers arrive in reversed order and no blocks are created. The stack on the left represents the arriving containers, starting with the top container. The three stacks on the right show a possible way to deposit these 5 containers. Here we clearly see that there will be no extra moves required to retrieve these 5 containers, as lower numbered containers are always stacked atop a higher number.

Figure 3



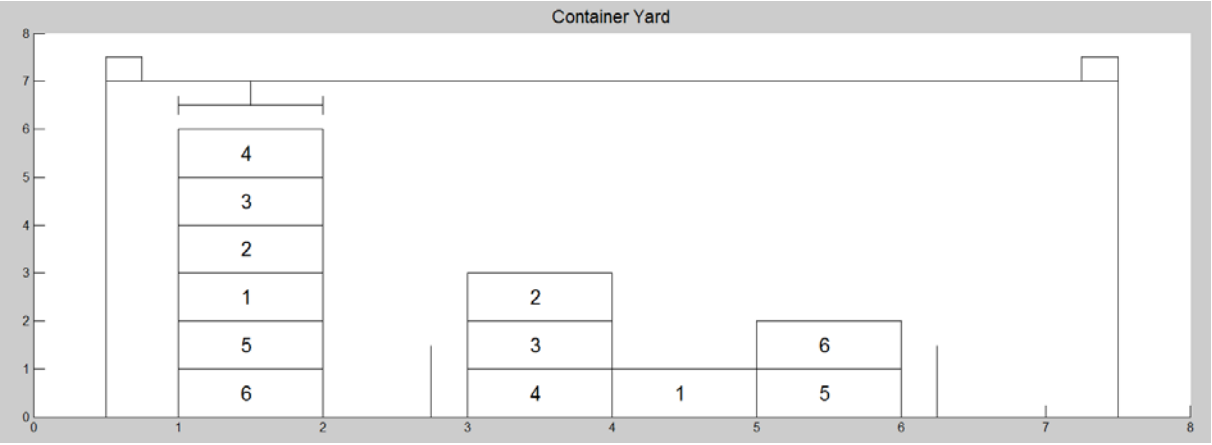
This next situation does require exactly two reshuffling moves. By assigning each container to a separate stack the first three containers can be accommodated without creating the need for a reshuffle. However, each additional container will have to be placed atop a container that will leave earlier and therefore create the need for a reshuffle.

This idea is converted into a minimum function by finding the longest increasing series of arriving container numbers in dataset $D_{i,j}$, without having one of the containers leave before the last container in the series arrives. This is done as follows:

1. move through $D_{i,j}$ step by step and register which containers are present at each step of the problem.
2. At each step calculate the longest list of increasing container numbers that can be made from the arrival sequence of the containers present at that point.
3. Take the maximum of these lists and compare it against the number of stacks. If it is larger than the number of stacks the minimum number of moves is increased by the difference between the length and the number of stacks. The containers in the list are removed from $D_{i,j}$ and we go back to step one. If the maximum length is smaller or equal to the number of stacks, no additional minimum moves are added and we stop.

We can further improve this minimum by taking into account stack height. We consider the arrival sequence [4 3 2 1 5 6], where container 4 arrives first and is the fourth container to leave and we assume again we have 3 available slots to place the containers all with a maximum height of 3 containers. This situation is sketched in the diagram below. The longest increasing sequence in this arrival sequence has a length of 3, which means that we could not assign a minimum move based on the rules described above. However by looking at available spaces, we can deduce that this system needs at least 1 reshuffling move to be solved. For the last two containers to arrive (5 and 6) have higher numbers than the previous 4 arrivals, which means that they depart after the first four leave. This means that they both have to be placed on an empty spot, otherwise they would *block* another container and would have to be moved at some point. This however is impossible as the first 4 containers cannot all be placed in one stack and therefore take up at least 2 spaces, leaving only one space for the new arrivals.

Figure 4



This can be translated into the general rule that new arrivals that arrive in reverse order and have later departure dates than the containers already in the system, are limited by the available space calculated below:

Number of free stacks = Number of stacks – ceil(containers in system/stacks height)

Where the function ceil () means round up.

This always holds true as long as the containers already in the system all have lower departure sequence numbers than the arriving containers and the new containers arrive in the reversed order in which they will depart. Coincidentally the tracking of available slots is also the key to the next problem.

For the question now becomes, what happens when these sequences become very long? We now consider longer arrival and departure sequences that contain for example 40 or more arrivals and have multiple increasing sequences greater than the number of stacks. The question is how do they interact? Can increasing sequences share numbers or overlap and what happens when containers leave in the meantime? We take the following example to illustrate.

4

5

8

9

14

15

4->

5->

16

17

The following set should be read from top to bottom. So container number 4 arrives first, and will leave after container 15 arrives. As shown before any increasing sequence is valid as long as no containers leave in the meanwhile. So here we can make 2 valid maximum increasing sequences of 6 long: [4 5 6 9 14 15] and [8 9 14 15 16 17], both indicate a minimum number of 3 reshuffling moves when we assume there are 3 stacks. Now one possibility is to find the longest sequence in an arrival/departure sequence, count the number of extra moves and then remove all the numbers in that sequence. And then repeat the procedure until we can no longer find a sequence greater than the number of stacks. In our example above it would mean we would take the first sequence and add 3 extra moves and remove the numbers, leaving us with only [16 17] and a total of 3 extra moves. This is a valid technique, however not a very strict approximation of the minimum, as the actual minimum number of containers is in fact higher. But we can also not simply add the number of extra moves found in both sequences as a quick solution of the problem finds a solution with less than 6 extra moves. So somewhere there must be a balance between these two methods.

The answer comes from the previous section, where we would track the number of free spaces. We will approach the problem sequentially, starting at the beginning and then moving our way down. So until our first departure we find a maximum increasing sequence 6, after which two containers will leave. This means that no matter what happens next, we need at least a minimum of 3 reshuffling moves. As the 2 containers arriving after 4 and 5 depart, both have higher numbers and arrive in reversed order the number of extra moves needed for these 2 containers depends on the maximum number of free spaces available when they arrive. This maximum can easily be calculated and depends on the total number of containers left in the system, the number of containers that arrived in reversed order, the number of moves done, and the number of departures. This can fairly easily be calculated for any combination of these values. A part of these results are shown in Appendix A1.

So for our problem we look at the list and see that for 6 arriving containers in reversed order, 3 extra moves and 2 departures, the second row from the bottom in Table A1 indicates that we can have a maximum of one free space. This is easily verified by the fact that we have 4 remaining containers in the system that have to be spread out over at least two stacks. So with one free space and two arriving containers we determine that we need at least one extra move to solve this system. Which equals the best solution found for this problem by a brute force approach, of 4 reshuffling moves.

Now if this problem is longer we can update the number of free spaces depending on how many containers leave next. This way we can determine the minimum number of moves in a linear time frame relative to the arrival sequence length.

2.4 Brute sequence

The brute force approach from section 2 is only an effective solution for problems that have no more than 10 to 12 arrivals and departures. This is due to the fact that the number of possible solutions and therefore the time to solve a problem grows exponentially when the number of departures or number of reshuffling moves is increased by one. Every time a problem is increased by one arrival the calculating time is multiplied by approximately the number of stacks. For an extra reshuffling move this increase is even greater. By using the minimum found in the previous section we can improve this in two ways. It will allow us to save time looking for solutions that are unattainable and it will allow us to stop looking for solutions once we have found one with that minimum.

However we can go even further. A brute force approach faithfully checks every solution. Even if it is clear that the first part of the solution is infeasible, it will continue to check every possible permutation of the second half to no avail. Now it is possible to built in all sort of complicated checks that will make the algorithm skip over certain sections if it finds the early parts infeasible, but this is very difficult because one reshuffle can make an almost impossible solution all the sudden feasible. The time saved by these checks is often offset by the time it takes to run the checks.

Another idea is to run sequential brute force algorithms for parts of the problem. For example, we can calculate all the possible positions we can achieve in the first four steps of a problem. Different combinations of placing containers and reshuffling them can result in the same end solution, so the list should be filtered so it only contains different end positions. It doesn't matter how that end position is reached, so we take the first solution that results in that end result and don't accept others into the list that also end in that configuration.

Now the different configurations also have to be evaluated, as some are acceptable in attempting to find an optimal solution and others are not. In the previous section we found a method that not only finds the total minimum number of extra moves a system will require, it also indicates which specific containers will result in reshuffles. By using those results we will allow a section to try different numbers of reshuffling moves starting at zero and up to the number of moves predicted by the minimum function in that section. Now for each configuration of containers it is possible to predict how many reshuffling moves will be needed. As each container higher in the stack that has a later departure number than a container in a lower part of the stack indicates at least one reshuffling move is necessary to move the latter container. Now for a candidate solution for our list of feasible solutions, we check the number of reshuffling moves predicted by our evaluation function and we add that to the number of reshuffling moves done to reach that position and only if this sum is equal to the minimum number of moves required until that point as predicted by the minimum function, is the solution added to the list.

We now have a list of possible solutions for example 4 steps out. If the minimum function determined that at least one reshuffling move is needed in these four steps, the list can contain solutions that have done one reshuffling move and solutions that have done no reshuffling moves but contain a block in the end position that will require a future reshuffle.

Depending on the size of the stacks and the number containers in the system the number of solutions in this list will vary, but for problems with 3 stacks usually hovers around 20 solutions and rarely exceeds 150 solutions. Now we will calculate the next four steps out to eight positions, starting with each ending position found in the previous section. All these solutions will then be combined into a new list of end positions. This process is repeated until we reach the end of the problem. The time required to calculate all the possible ending solutions from one base position with one reshuffling moves is about a third of a second. Which means that if our list contains on average 20 solutions, each step will take around 7 seconds. But on the far end a step can also take more than a minute. But it does create a solution in a linear relation to the length of the problem, instead of an exponential. Now for shorter step sizes (for example 2 steps ahead) the time required to solve the entire problem decreases, but this also has risks.

This Brute Sequence Solution does not always find an optimal solution, for sometimes it is better to do a reshuffle move a couple steps ahead of the container that causes the need for a reshuffle. Now if the step size is larger more of these optimal solutions are caught than when the step size is small. This creates a tradeoff between better solutions and calculation time. For this reason or the fact that the actual minimum number of extra moves can be higher than the function calculates, sometimes no feasible solution is found in an iteration of the problem. In that case an extra move is awarded and the iteration is run again.

Although the number of possible intermediate positions is fairly limited in problems with only 3 stacks, this number will grow rapidly as the number of available stacks and simultaneously present containers increases. This will make the brute sequence method ineffective in the same way the original brute force method was ineffective, namely it will take too long to evaluate all the possibilities. Therefore a variant of this brute sequence has been implemented, that limits the number of possible of intermediate positions in each generation to a number that can be specified. Although limiting the number of intermediate positions might diminish its result, it will allow us to solve problems that would otherwise take too long to solve.

2.5 Practical Algorithms

The previous Brute Sequence method is an interesting and effective method for finding solutions. However it is not a very practical method for larger problems, as it requires a great amount of time to calculate. In this section we discuss a number of practical algorithms developed from the above results to solve container arrival and departure problems. Here we distinguish between 2 types of solutions. Real time solutions, where the heuristic only knows the current containers in the system and the next arriving or departing container and overall solutions where the entire sequence of arrivals and departures is know beforehand.

Algorithm 1

The First algorithm is a real time solution, that only looks at containers as they arrive. It will initially attempt to place a container on a stack whose minimum number is higher than that of the arriving container and is as close to the container number as possible. If this is not possible it will put the container in empty stack. When no free spots are available it will put the container on the smallest stack. If there are multiple stacks that share the smallest number of containers the stack with the lowest sequence number is chosen. When a container has to be removed because a container below is departing, the same principle is applied. This simple algorithm will be considered our base algorithm.

Algorithm 2

The second algorithm works in the same way as the first algorithm works except that when a container cannot be placed on a stack with a higher departure number or in a free spot, it will be placed on the stack with the smallest container numerical difference between the new container and the minimum container number in the stack. The idea is that this will leave more options open for future containers.

Algorithm 3

The third algorithm works in the same way as the first algorithm except that when a container cannot be placed on a stack with a higher departure number or in a free spot, it will see if it can create a free spot by moving one container on top of another without causing an extra reshuffle. Where it attempts to minimize the difference between the two containers. If this is not possible it will place the container on the stack with the fewest containers.

Algorithm 4

The fourth real time algorithm combines the second and third algorithms. When no free spot is available it will first attempt a reshuffle otherwise it will attempt to minimize the difference between the container it is put on top of.

All of these algorithms perform decently when the problems are *easy,* meaning containers arrive in the reversed order from their departure. But when the problems become difficult these algorithms can run into trouble. The difficulty is foresight or rather lack of foresight. All these algorithms tend to be greedy, they will for example rather fill the last empty slot than incur a block. But sometimes this can be better as illustrated below. We have two situations with three available stacks 2 of whom are already populated by the stacks [5 4] and [7 3] respectively. Now container 6 arrives first in both cases, shown by the arrival stack under the crane on the left. The difference lies in what happens next. If container 6 is followed by container 2 and then 1, the above algorithms are optimal when they fill the empty slot. However, if containers 6 is followed by containers 14 and 13 it would have been better to place container 6 on top of one of the other stacks of containers. This would result in only one extra move instead of the two extra moves, the algorithms incur now.

Figure 5

Situation 1

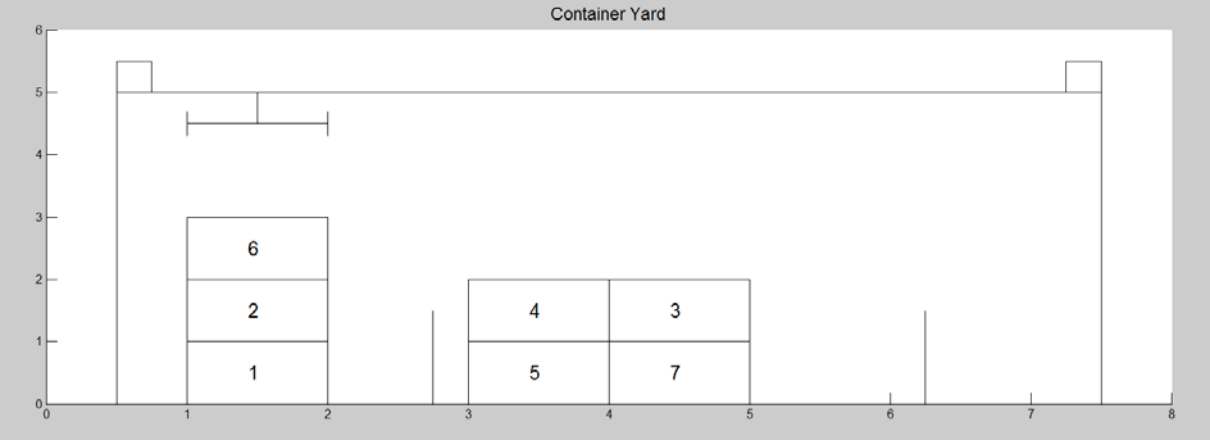
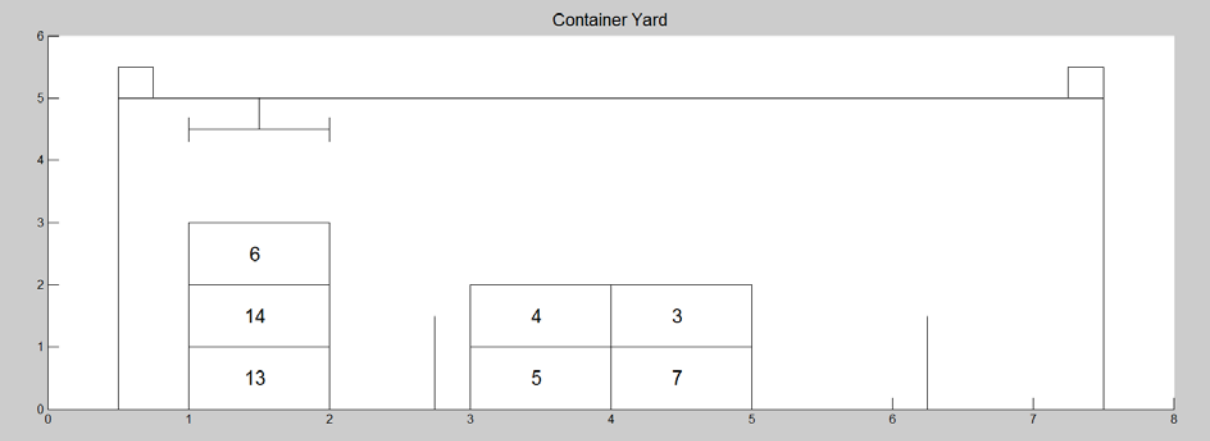


Figure 6

Situation 2



Overall solutions

Next we will consider a set of algorithms that can take advantage of this foresight. Here we assume that the exact sequence of arrivals and departure is know beforehand.

Algorithm 5

The first algorithm that does not operate in real time, simply takes the minimum of the first 4 algorithms. Specifically, it runs all 4 algorithms and then picks the one with the least number of moves. This will serve both as our baseline non-real time algorithm as well as an indication of the variance of the first 4 methods.

Algorithm 6

This algorithm is a variant of the first set of algorithms. It again places containers on top of containers with a later departure time when possible. Otherwise it put the container on an empty spot. However if there are no empty stacks available, it will compare the performance of putting the container in each available stack and choose the one that creates the smallest number of blocks. It does this by simulating for each stack what would happen if the current container was put in that stack and moving forward through all the remaining arrivals and departures algorithm 4 was applied.

Algorithm 10

The final algorithm is an adaptation of the brute sequence approach and is not an online solution. It has the advantage over the other algorithms in that it can try multiple solutions at the same time and later discard unfavorable ones. It again creates a list of possible end configurations after a number of steps that can be specified. However, now it limits the number of configurations in the list to a number that is left as a variable. This way the operator can weigh the balance between quality of the solution and calculation time required. The list is reduced by generally favoring the solutions that have more open spaces and configurations with few blocks.

3 Results

In this section I consider the results of the various parts of my research. The Data generator does not have results of its own, but is checked for level of difficulty of the generated problems in section 3.1. The Brute force approach guarantees the optimal solution so will only be checked for performance in calculation time in section 3.2. In section 3.3 the two minimum functions are compared to a calculated actual minimum. And finally in section 3.4 the different solving algorithms are compared.

3.1 Data generator results

The data generator simply generates a set of arrivals and departures based on the given parameters. These parameters are: the number of arrivals, the maximum number of containers in the system and a difficulty factor. The first two parameters will always be adhered to and do not require testing. The only part worth examining is the difficulty of the problem as these parameters are increased. This will be measured in a couple of ways. First the distribution of the number of extra moves required with the same parameters. Next the difference between different parameter levels. Here we can look again at the minimum number of extra moves required or the average number of containers in the system as an indication of difficulty.

Figure 7

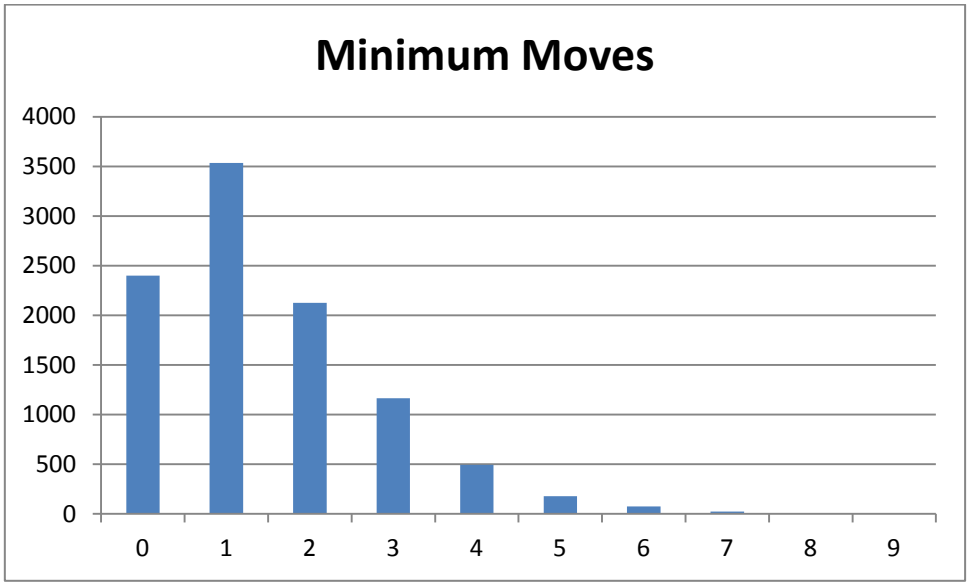


Table 8

	Distribution of extra moves for a given set of parameters									
Extra Moves	0	1	2	3	4	5	6	7	8	9
Frequency	2401	3533	2126	1165	493	178	74	24	4	2

Above is a graph and corresponding table of the distribution the minimum number of extra moves required, generated 10,000 times with the following parameters: number of arrivals: 50, max containers in system: 7, difficulty parameter: .95. There is a little more variation than desired, but if this were an issue datasets could be selected based on the average number of containers in the system and the minimum number of moves needed as calculated by the minimum moves function.

It should be noted that these minimums have been obtained through the minimum function and are therefore an approximation and not an actual minimum. They are however very consistent and parallel simulations created nearly equal results.

Next we look at the progression of the number of extra moves required and the average number of moves in the system, as the value of the difficulty parameter increases.

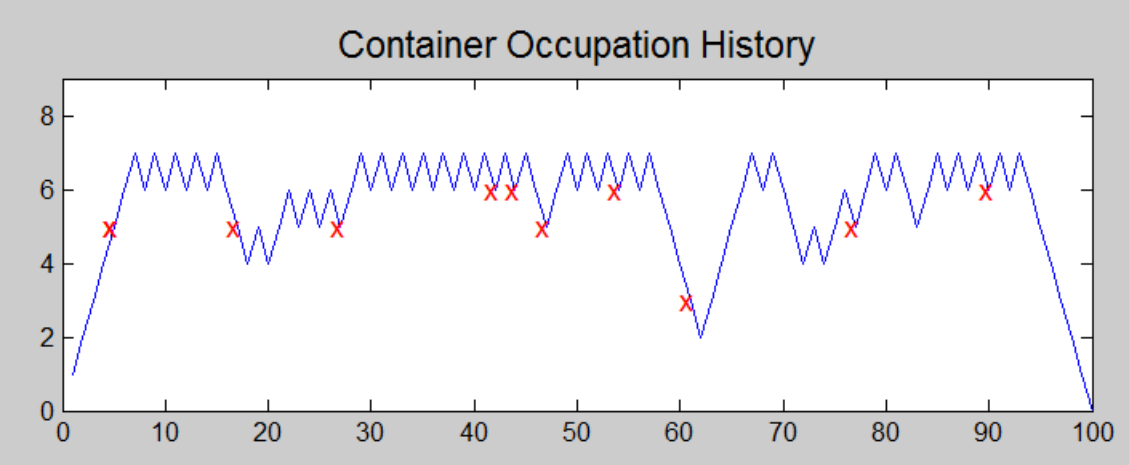
Table 9

	n = 50		1000 simulations					
Parameter	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Average containers	0.82	1.19	1.90	3.12	4.41	5.25	5.67	5.93
Average extra moves	0.06	0.22	0.60	1.11	1.36	1.46	1.45	1.45
Average max containers	3.35	4.49	5.86	6.83	7.00	7.00	7.00	7.00

The table shows a steady increase in difficulty on average as the parameter goes up. This is shown by the average number of containers in the system that increases, as the parameter increases. The predicted minimum extra moves also increases with the parameter. As well as the maximum number of containers present at any point. This is the desired result we were hoping for.

To get another perspective on the generated dataset we can create a graph that show the number of containers present after each event. The dataset behind the graph below, has 50 arrivals and 50 departures for a total of 100 events. The red crosses show the locations of reshuffle moves.

Figure 8



3.2 Brute force results

The brute force method will always find the best result and therefore does not really lend itself to result testing. The only aspect worth investigating is the amount of time required to find a solution.

Table 10

		number of possible solutions				
		0	1	2	3	4
number of arriving containers	1	1	NR	NR	NR	NR
	2	2	24	NR	NR	NR
	3	5	120	1,440	NR	NR
	4	15	540	10,800	155,520	NR
	5	51	2,448	66,096	1,321,920	21,150,720
	6	187	11,220	376,992	9,370,944	190,003,968
	7	715	51,480	2,059,200	60,540,480	1,452,971,520
	8	2,795	234,780	10,866,960	367,061,760	10,084,538,880
	9	11,051	1,060,896	55,697,040	2,119,670,208	65,308,757,760
	10	43,947	4,746,276	278,448,192	11,770,764,480	400,965,396,480
	11	175,275	21,033,000	1,362,938,400	63,300,916,800	2,358,792,057,600
	12	700,075	92,409,900	6,552,702,000	331,465,910,400	13,398,965,049,600

The table above shows the number of different solutions for different numbers of arriving containers on the vertical axis and different numbers of reshuffles on the horizontal axis in a system with 4 stacks. A household desktop running Matlab can check around 20,000 solutions per second, this means that realistically we will only be able to test the solutions of problems containing 10 arriving containers and two reshuffles or less in a system with 4 stacks. The table above is converted into a rough approximation of calculation time using the formula above. It shows that even for reasonable problems of 12 containers needing 4 reshuffles, the calculation time to run through all the possibilities quickly runs into the weeks and years. It is a good illustration of the limitations of this approach and the need for different solutions.

Table 11

Calcultion Time											
		Number of reshuffles									
		0		1		2		3		4	
1	Number of arriving containers	0.0001	S								
2		0.0001	S	0.001	S						
3		0.0003	S	0.006	S	0.07	S				
4		0.001	S	0.027	S	0.5	S	7.8	S		
5		0.003	S	0.12	S	3.3	S	1.10	M	17.63	M
6		0.01	S	0.6	S	19	S	7.81	M	2.64	H
7		0.04	S	2.6	S	1.72	M	50.45	M	20.18	H
8		0.14	S	12	S	9.06	M	5.10	H	5.84	D
9		0.55	S	53	S	46.41	M	1.23	D	5.40	W
10		2.2	S	3.96	M	3.87	H	6.81	D	33.15	W
11		8.8	S	17.53	M	18.93	H	5.23	W	3.74	Y
12		35	S	1.28	H	3.79	D	27.40	W	21.24	Y
		S = Seconds M = Minutes H = Hours D = Days W = Weeks Y = Years									

However, the above results don't mean the brute force algorithm cannot be used for the purposes of this research. Small problems can still be checked for optimal solutions. Furthermore, the above results show the possibilities for 12 *arriving* containers, when *arrivals* are mixed with *departures*, the number of possibilities comes down. Also this table is based on a system with 4 stacks, calculations in systems with 3 stacks (which we use most often in this thesis) are much quicker.

3.3 Minimum function results

When examining the minimum functions described in section 2.3 we are interested in how close the estimates of the minimum number of extra moves approximates the actual number of extra moves.

Table 12

	n = 8 containers		1000 simulations
	Minimum Function	Minimum Function with free space	Actual Minimum
Total extra moves	764	878	1140
Average extra moves	0.764	0.878	1.140
Percentage moves	0.670	0.770	-
percentage correct	0.64	0.71	-
Biggest difference	3	2	-

In this table we compare the two minimum functions with the optimal solution found by the brute force method described in section 2.2. To generate the table above 1000 datasets of 8 arriving containers were generated and their minimum number of extra moves were calculated by the brute force method. At the same time the standard minimum function (column 1), where sequences are removed was run to get an estimate of the minimum number of moves required. As well as the minimum function that utilizes the free spaces (column 2).

The first row of the table shows the Total number of extra moves predicted by the 2 functions and the actual number of extra moves in the last column. We see that the second function performs slightly better than the first function. In the second row these totals are converted to averages per problem. The third row shows what percentage of reshuffling moves was correctly predicted by each function. We see that the first function is able to guess around two thirds of the total number of required reshuffling moves. The second function can predict a little more than three quarters of the actual number of moves. The fourth column represents the percentage of cases where the number of reshuffling moves predicted by the function is right on the money. Again the second function gets it exactly right slightly more often than the first function. The last row shows the biggest difference between the actual number of required reshuffling moves and the predicted number.

The second minimum function clearly performs better. It's guesses are closer to the actual minimum and it is more often exactly right about the number of extra moves. The most important thing is that neither function ever gave a higher estimate of the minimum number of moves required, than the actual result.

3.4 Results Algorithms

Finally we compare the algorithms we developed in section 2.4. In order to test them we generate 100 problems of 25 arriving and departing containers into a system with 3 stacks. Then each algorithm was performed on each problem and the results are calculated. This is repeated ten times, to get a sense for the stability of the results. The results are then combined into the table below. The initial tables are included in appendix A2:A11.

Table 13

Simulation Combined	n = 25 containers				100 simulations					
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	58.2230	56.6450	55.4800	55.0610	54.3280	54.5750	53.1430	52.9880	52.8310	53.0489
Ave moves / container moves	1.1645	1.1329	1.1096	1.1012	1.0866	1.0915	1.0629	1.0598	1.0566	1.0610
stdev number of moves	0.4174	0.3216	0.2812	0.2240	0.1661	0.5931	0.3952	0.3555	0.1107	0.3489
maximum moves	74	68	66	64	61	61	57	57	57	57
Minimum moves	50	50	50	50	50	50	50	50	50	50
Total time	1.34269	1.33915	1.47066	1.77727	5.92976	144.43329	2516	6902	21692	2445
Time per problem	0.00134	0.00134	0.00147	0.00178	0.00593	0.14443	2.516	6.902	21.692	2.445

* Algorithms 1 through 4 are the real time algorithms described in section 2.5

* Algorithm 5 is the minimum of the first 4 algorithms

* Algorithm 6 is the non-real time algorithm described in section 2.5, which evaluates possible moves by simulating their effect on future arrivals.

* Algorithms 7 through 8 are the Brute Sequence algorithm described in section 2.4. run with respectively 2, 3 and 4 step sizes.

* Algorithm 10 is the Brute Sequence variant described at the end of section 2.4, with 4 step size of 4 events and a limit of 20 configurations

The first row shows the average number of moves it took each algorithm to solve the problem. This should be compared against the baseline of 50 moves, which are certainly needed to move the 25 containers into and out of the system. The second row is a performance statistic where the average number of total moves are divided by the 50 essential moves. It shows the average number of extra moves required per essential move. The third row shows the stander deviation of the Average number of total moves of the 10 separate simulation runs. The next two rows show the two solution in the simulation with the most and the fewest number of required moves respectively. The second to last column shows the total time in seconds it took the algorithm do complete all 100 simulations and the last column show the average time per problem

It is clear that all variants of the brute sequence algorithm greatly outperform all the real time solutions. It is obvious that having foresight gives a great advantage, but within the real time algorithms there is also progress being made against the baseline. Especially the maximum number of moves is greatly reduced by the last variant of the real time solutions against the baseline variant. What also should be noted is that the real time algorithms take a fraction of the time to run, compared to the brute force sequence methods. So the solutions are improved, but this comes at a cost.

Within the brute sequence methods we also see a tradeoff between time and quality of the solution, measured in extra moves. Bigger steps generally give better solutions, but also invariably take more time to run. Generally, because due to chance a Brute Sequence algorithm sometimes performs better with a smaller generation size. But in general being able to look further in the future gives an advantage. Although Algorithm 9 with generation size 4 does give slightly better results on average than Algorithm 7 with generation size 2, it comes at a huge cost of a tenfold increase in calculation time.

The adjusted Brute Sequence with the limit on the number of positions per generation performs quite well compared to the other variants. The time is cut in half compared to the regular brute sequence algorithm 8, but the results are only slightly below those of algorithm 8. Especially the more precise ability to adjust the balance between time and result is a nice aspect of this variant. The key in further research will be to select those situations that are most likely to perform well in the remainder of the problem.

Conclusion

In this thesis I examine different strategies for solving deterministic container stacking problems, where the arrivals and departures can be either known or unknown beforehand. I find that it is possible to implement a brute force approach to solving these types of problems, but its effectiveness is limited to very small problems. This could be improved by coding it in a more efficient programming language or running it on a faster computer. However, the size of the problems that can be calculated in this fashion would still be limited and can only be used for research purposes. Furthermore I find that a minimum function can be a useful tool in optimizing these types of problems. It allows us to quickly set a barrier for both entire problems and sections of a problem, beyond which we don't have to search. This can both save time and help in evaluating these strategies. I suspect there is a lot of room for improvement here. Most reshuffles can be readily predicted by looking at the dataset. Especially in the interaction between different sections of a problem I believe there is room for improving the minimum function. But one could also look into the possibility of simplifying problems beforehand. Containers arriving and subsequently departing almost immediately could be taken out of consideration in the optimization process, as they will always fit into a solution. In our comparison of the different algorithm we find that there is a tradeoff between the quality of the solution measured by the number of reshuffling moves needed to solve the problem and the computational time needed to calculate this solution. Quick solution can be achieved by simple algorithms and every incremental improvement of the solution increases the computational time exponentially. I believe that the concept of the Brute Sequence method has been shown both feasible and effective in dealing with larger problems with more arrivals. The Brute Sequence method calculates a number of steps ahead to a number of intermediate positions and then moves forward from the feasible positions, discarding infeasible branches of solutions along the way. It has worked in that it has turned the exponential increasing time it takes to solve longer problems into a linear increase. There is still a great deal of improvement possible here by making less, but more informed guesses in each iteration.

References

Asperen, E. van, Borgman, B. & Dekker, R. (2010). evaluating container stacking rules using simulation. In J. Hugan & E. Yücesan (Eds.), proceedings of the 2010 winter simulation conference proceedings of the winter simulation conferences (pp. 1924-1933). Baltimore.

Borgman, B., Asperen, E. van & Dekker, R. (2010). Online rules for container stacking. OR Spectrum, 32, 687-716

Dekker, R., Heide, S. van der & Asperen, E. van (2010). A Chassis Exchange Terminal to Reduce Congestion at Container Terminals. In Proceedings of the LOGMS conference International Journal of Flexible Manufacturing Systems. Korea.

Dekker, R., Voogd, P. & Asperen, E. van (2007). Advanced methods for container stacking. In K.H. Kim & H.-O. Gunther (Eds.), Container terminals and cargo systems. Design, operations management, and logistics control issues (pp. 131-154). Berlin: Springer.

Duinkerken, M.B., Dekker, R., Kurstjens, S.T.G.L., Ottjes, J.A. & Dellaert, N.P. (2007). Comparing transportation systems for inter-terminal transport at the Maasvlakte container terminals. In K.H. Kim & H.-O. Gunther (Eds.), Container terminals and cargo systems. Design, operations management, and logistics control issues (pp. 37-62). Berlin: Springer.

Vos, S: 2012, A Survey of Recent Developments in Seaport Terminal Management

5 Appendix

Table A1

Maximum number of free spaces for different systems

C	M	D	S
2	0	1	2
2	1	1	2
3	0	1	1
3	1	1	2
3	2	0	2
3	1	2	2
4	1	0	0
4	2	0	1
4	1	1	1
4	2	1	2
4	1	2	2
5	2	0	0
5	2	1	1
5	2	2	2
5	2	3	2
5	3	0	1
5	4	0	1
5	3	1	1
5	3	2	2
6	3	1	1
6	3	2	1
6	3	3	2

C = Number of containers arriving in reversed order

M = Number of moves done

D = number of departures

S = number of free spaces

Tables A2:A11

Individual simulation results from section 3.4

Simulation 1	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	58.75	57.05	55.74	55.17	54.55	54.85	53.91	52.9	52.83	52.93
Ave moves / container moves	1.175	1.141	1.1148	1.1034	1.091	1.097	1.0782	1.058	1.0566	1.0586
maximum moves	69	64	62	61	61	60	56	56	56	56
Minimum moves	51	51	51	51	51	51	50	50	50	50
Total time	0.12681	0.13138	0.13347	0.15202	27.53333	0.27	261.39	706.37	3473.78	407.29
Time per problem	0.00127	0.00131	0.00133	0.00152	0.27533	0.00	2.61	7.06	34.74	4.07

Simulation 2	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	57.74	56.49	55.25	55.19	54.24	54.89	52.92	52.78	52.8	52.81
Ave moves / container moves	1.1548	1.1298	1.105	1.1038	1.0848	1.0978	1.0584	1.0556	1.056	1.0562
maximum moves	70	68	61	64	58	60	57	57	57	57
Minimum moves	50	50	50	50	50	50	50	50	50	50
Total time	0.13047	0.12902	0.13694	0.15484	28.14855	0.26	297.77	507.60	2938.16	203.12
Time per problem	0.00130	0.00129	0.00137	0.00155	0.28149	0.00	2.98	5.08	29.38	2.03

Simulation 3	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	57.98	56.3	55.39	55.35	54.42	54.9	53	53.93	52.87	53.94
Ave moves / container moves	1.1596	1.126	1.1078	1.107	1.0884	1.098	1.06	1.0786	1.0574	1.0788
maximum moves	69	64	64	62	59	60	57	57	56	57
Minimum moves	50	50	50	50	50	50	50	50	50	50
Total time	0.14944	0.14053	0.14943	0.17645	31.39286	0.32	287.81	499.60	2725.36	328.31
Time per problem	0.00149	0.00141	0.00149	0.00176	0.31393	0.00	2.88	5.00	27.25	3.28

Simulation 4	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	58.56	56.88	55.84	55.4	54.54	55.04	53.15	53.08	52.99	53.12
Ave moves / container moves	1.1712	1.1376	1.1168	1.108	1.0908	1.1008	1.063	1.0616	1.0598	1.0624
maximum moves	74	65	61	61	59	61	56	56	56	56
Minimum moves	52	51	51	51	51	51	51	51	51	51
Total time	0.12456	0.12863	0.14049	0.14999	28.32320	0.28	240.61	608.50	1677.96	281.50
Time per problem	0.00125	0.00129	0.00140	0.00150	0.28323	0.00	2.41	6.08	16.78	2.81

Simulation 5	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	57.84	56.7	55.84	54.72	54.21	54.52	53.84	52.87	52.82	52.88
Ave moves / container moves	1.1568	1.134	1.1168	1.0944	1.0842	1.0904	1.0768	1.0574	1.0564	1.0576
maximum moves	69	67	66	62	61	60	57	57	56	57
Minimum moves	51	51	51	51	51	51	50	50	50	50
Total time	0.11966	0.12393	0.13287	0.14680	26.96748	0.25	135.25	346.61	903.44	182.43
Time per problem	0.00120	0.00124	0.00133	0.00147	0.26967	0.00	1.35	3.47	9.03	1.82

Simulation 6	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	58.155	56.695	55.26	55.055	54.275	54.7	52.94	52.825	52.71	52.855
Ave moves / container moves	1.1631	1.1339	1.1052	1.1011	1.0855	1.094	1.0588	1.0565	1.0542	1.0571
maximum moves	69.5	66	61	62	59	60	57	57	56	57
Minimum moves	51	51	51	51	51	51	50	50	50	50
Total time	0.12468	0.12610	0.13733	0.19843	27.74697	0.28707	219.82	420.55	2362.35	193.79
Time per problem	0.00125	0.00126	0.00137	0.00198	0.27747	0.0029	2.20	4.21	23.62	1.94

Simulation 7	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	58.5	56.57	55.59	55.08	54.37	54.66	52.94	52.97	52.88	52.98
Ave moves / container moves	1.17	1.1314	1.1118	1.1016	1.0874	1.0932	1.0588	1.0594	1.0576	1.0596
maximum moves	68	66	64	60	59	59	56	56	56	56
Minimum moves	50	50	50	50	50	50	50	50	50	50
Total time	0.13772	0.14585	0.15541	0.18072	35.57618	0.31	176.03	301.19	1236.35	221.34
Time per problem	0.00138	0.00146	0.00155	0.00181	0.35576	0.00	1.76	3.01	12.36	2.21

Simulation 8	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	57.58	55.99	55.02	54.8	53.98	54.64	52.84	52.83	52.68	52.83
Ave moves / container moves	1.1516	1.1198	1.1004	1.096	1.0796	1.0928	1.0568	1.0566	1.0536	1.0566
maximum moves	66	64	62	61	60	61	57	57	57	57
Minimum moves	50	50	50	50	50	50	50	50	50	50
Total time	0.12742	0.14009	0.13828	0.15938	28.30281	0.27	334.86	994.82	1843.39	304.94
Time per problem	0.00127	0.00140	0.00138	0.00159	0.28303	0.00	3.35	9.95	18.43	3.05

Simulation 9	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	58.14	56.67	55.59	55.06	54.35	54.78	53	53.03	52.92	53.05
Ave moves / container moves	1.1628	1.1334	1.1118	1.1012	1.087	1.0956	1.06	1.0606	1.0584	1.061
maximum moves	67	66	61	60	59	60	56	56	56	56
Minimum moves	51	51	51	51	51	51	50	50	50	50
Total time	0.18880	0.15334	0.20834	0.17302	44.37657	0.30	307.09	817.18	4922.67	331.60
Time per problem	0.00189	0.00153	0.00208	0.00173	0.44377	0.00	3.07	8.17	49.23	3.32

Simulation 10	n = 25 containers					100 simulations				
Algorithm #	1	2	3	4	5	6	7	8	9	10
Average number of moves	58.57	56.9	55.27	54.92	54.31	54.51	52.96	52.87	52.62	52.90
Ave moves / container moves	1.1714	1.138	1.1054	1.0984	1.0862	1.0902	1.0592	1.0574	1.0524	1.058
maximum moves	69	64	61	60	60	60	56	56	55	56
Minimum moves	51	51	51	51	51	51	50	50	50	50
Total time	0.11890	0.12319	0.13772	0.24202	27.34540	0.31	141.87	333.49	1786.55	184.45
Time per problem	0.00119	0.00123	0.00138	0.00242	0.27345	0.00	1.42	3.33	17.87	1.84