F.C.T. Universidade do Algarve

Departamento de Engenharia Eletrónica e Informática (DEEI)

AI Heuristic Approach to Solving a Minimization Problem "Containers"

Diogo Fonseca

(no. 79858, grp. 3)

Professor(s): José V. Oliveira and Pedro M. S. V. Martins

A report submitted in partial fulfilment of the requirements of
UAlg for the degree of Engenharia Informática in *Artificial Intelligence*

(Problem 2: Containers)

October 24, 2024

Abstract

This is an undergraduate project to solve the minimization problem "Containers" with a heuristic state space search, where we want the lowest cost path to a solution. One of the main objectives is to compare it to an uninformed state space search: Best-First in this case. The algorithm used was a heuristic state space search is A*, which was (on the average case) much, much faster than the uninformed search, which is easy to conclude from the fact that unlike the uninformed state space search, it has a general idea of where the solution might be (thus being informed).

Contents

Chapter 1

Introduction

1.1 Background

The purpose of this project is simply to implement a heuristic space state search, for learning purposes, and also to show the comparison between an uninformed space state search, and an informed one.

1.2 Problem statement

The problem is to get the lowest cost possible to go from a configuration of containers to another configuration of containers.

A container c can be thought of as a square box with a given weight w. A container can also be stacked on top of another.

From a given configuration of containers, on the action/move a_i a single container can be placed on top of a different stack of containers or it may be placed on the ground. The a_ith move has the cost w_i of the moved container.

If N is the number of moves to reach the goal, it could then be said that the cost from the start configuration to the goal configuration is:

$$cost = \sum_{i=0}^{N-1} w_i$$

Our goal then is to find the **minimum** possible value for $cost$, value which generally also means finding a path that **minimizes** $cost$.

The problem then has some additional restrictions: A container may be identified only by their alphabetic character (with clear separation from lowercase and uppercase), furthermore a weight must be represented by a single digit, so the following restrictions apply

$$(1 \leq c \leq 52)$$

$$(1 \leq w \leq 9)$$

INPUT: Two lines, the first represents the initial configuration, the second represents the goal configuration. Both of the lines have the following format: each stack of containers is separated by a white space. Each stack has one or more containers, containers are described by the letter they are identified by, followed by their weight. The second line (representing the goal configuration) does not necessarily need to contain weights, as these are immutable.

OUTPUT: The goal configuration, each stack is a single line within square brackets with the character of the container from bottommost to topmost separated by commas. The stacks should be ordered by the lexicographic order of the bottommost container's character. It should then have an empty line followed by the minimum cost from the initial configuration to the goal configuration in yet another line

EXAMPLE:

INPUT:
A1a2D4 C1b8
b a DCA

OUTPUT:
[D, C, A]
[a]
[b]

16

EXPLANATION: The path that gets us the least cost is moving "D" to the ground (cost of 4), moving "b" to the ground (cost of 8), moving "C" on top of "D" (cost of 1), moving "a" to the ground (cost of 2) and finally moving "A" on top of "DC" (cost of 1), which has a total cost of 16.

1.3 Solution approach

The problem was approached using an **A* heuristic state space search**, which was thought to be the most appropriate for the problem at hand, given it was a minimization problem with a weighted path involved.

Chapter 2

Methodology

2.1 Why A*?

A* was chosen, simply put, because it is the best tool for the job. For the current problem we want to find the path that minimizes cost, more specifically what we really want is just the minimum cost, but in such a problem finding the minimum cost also means finding the path of least cost.

We want an algorithm that assures the optimal solution is always found, this rules out using something like hill climbing or beam search. It should be noted that simulated annealing or genetic algorithms could both be good candidates, but despite being stochastic, not assuring a global optima, they also don't offer any considerable performance improvements over A*, especially for the given problem where their fitness function would be heavily tied or equal to the A* heuristic found, not even giving them any performance benefits there.

Following this line of thought, there is another algorithm that could very well make as much sense to implement as A*: IDA*. The reason A* was chosen over IDA* is because the priority was performance and not memory, thus not making sense incurring the performance overhead of IDA* given we don't care how much memory gets used.

Furthermore, the heuristic found in general cuts enough branches so that the memory used is very minimal, causing the computation time to be the biggest bottleneck. The input would have to be near the upper limit (52) to make the current heuristic run out of memory on most modern computers, further making an argument against IDA*. For cases where we *do* want (possibly even more than) 52 containers (which the current implementation does not support), and computation time isn't a concern, for such cases IDA* might make more sense, but for the purpose of this problem we're only interested in the solutions that can be computed in a very reasonable time frame, thus putting IDA* out of the picture.

2.2 Heuristic deduction

The first and most important deduction to reach an admissible heuristic for this problem is to understand that a given container can **only** be moved between 0 and 2 times. So the cost from a given goal to the end goal is **exactly** the sum of the weights of all containers that must be moved once: h_1 plus the *doubled* sum of the weights of all containers that must be moved twice: h_2.

A useful way to put the problem moving forward is to define h_1 as all the containers that must be moved ***at least*** once, and thus h_2 would be defined as only the sum of the weights of all containers that must be moved twice (their first movement is already accounted for in

h_1). The total cost would then simply be $h_1 + h_2$.

2.2.1 Why any container must be moved between 0 and 2 times in total

0 times

A container must be moved 0 times if and only if it is already in the goal state, which means all the containers below it must also be in their goal state.

1 time

A container must be moved exactly once if and only if at some point in the steps to solve the problem, all that is necessary is for the container to be moved from where it is to it's goal configuration.

2 times

It can be proven that any container needs at ***most*** 2 moves to get to it's goal configuration. Any container can be placed on the ground (1 move), using it as a buffer to then be placed in it's goal configuration (1 move), thus being placed in it's goal configuration in a total of 2 moves at most.

Something that can either be used to visualize this, or also as a byproduct of this rule: The worst case scenario possible is for $2n - 2$ moves to occur, which happens when we have to keep placing the topmost container on the ground (except for the bottom one) and then reconstruct the stack to the goal configuration. This directly proves that:

1. Any initial-goal pair with the same containers has a solution.
2. Any configuration can be solved in $2n - 2$ moves (or less).

2.2.2 How to get h_1

Getting h_1 is really simple, all that's needed is to count every container that *isn't* in it's goal state and make the sum of all of their weights. We then have how many containers need to be moved **at least** once.

2.2.3 How to get h_2

Getting h_2 in polynomial time for any set of configurations is impossible. That said we can get a pretty good approximation while maintaining the heuristic admissible. All we have to do is sum the weights of all the containers we have total certainty of having to be moved twice.

What's left is to figure out what containers do we know for sure have to be moved twice.

- Any container ***has*** to be moved twice if there's a container below it **and** in it's goal configuration that container is also below the current one.

This ensures that the container must be placed on the ground first (conceptually, it technically could also be placed somewhere else), acting as a buffer until when it has to be placed somewhere in the stack on top of the container that was below it.

All containers that need to be moved twice for the optimal solution, but don't follow this pattern (in other words, we can't be completely sure without exploring more nodes if they really do need to be moved twice) will be counted as only moved once by h_1, making it so that $h_1 + h_2$ will always give a value under the true value, logically making this heuristic admissible.

2.2.4 h_2's blind spot

h_2's blind spot, more formally, the configuration for when h_2 underestimates, is when two containers must be swapped. This swap might not be direct, it could be any kind of swap-cycle. For example something like this would make h_2 not detect that one of them has to be moved twice: A moves to B, B then moves to C, C moves to A.

The detection of these "swap-cycles" can be extremely hard to detect, especially when there are several of them (possibly intertwined), needing an exponential algorithm to get an accurate value for these cases. And thus this should not be taken into account by the heuristic.

2.3 Code interpretation of the problem

This problem can have many different code implementations, but it's not trivial to think of one that makes every operation we need efficient. This section explains the implementation thought most appropriate.

NOTE: even though the problem has single-digit weights, the code implementation allows for any integer value as a weight parameter in the input.

2.3.1 Containers and weights

Both the existence of a container and it's weight was implemented as a single array, note this is only possible because we're working with a finite number of containers (52), for a pseudo-infinite number of containers, a hashmap might be more appropriate.

The "key" or index is simply a 0 based index of the position of the character in the alphabet where 0 is "A", 1 is "B" and 26 is "a". Because weights cannot be negative, we can define the absence of a container as having (any) negative weight. If the container does have a weight, that would mean not only that it exists but that we can get it's weights. This means we have **insertion**, **checking for containment** and **weight lookup** in $\boldsymbol{O(1)}$.

2.3.2 Stacks

We can represent "stacking", or placing a container on top of another using two arrays: One array represents the index of the container below it, the other the index of the container above it.

Because indexes cannot be negative, we can represent the lack of a container below it (a.k.a being on the ground) as a negative number in the array that holds the container below it. Analogously we can represent not having a container above it (a.k.a being the topmost container on it's stack) as having a negative number in the array that holds the containers above it.

This way we can **check the container below and above another** and even **check if a container is on the bottom or on top** of it's stack all in $\boldsymbol{O(1)}$.

Checking in **what stack** a container is in is performed in $\boldsymbol{O(n)}$, where n is the number of containers below it (51 at worst case, a constant number, so it could be argued that the true cost is $O(1)$). This is because we need only iterate over container always checking what the container below it until we reach a negative number.

Because the containers are already ordered by the way they get placed in the arrays, this makes the printing operation as simple as iterating over all 52 elements of the array, checking if they are at the bottom, and printing the whole stack following the containers on top for every bottom container found. The worst case scenario is performing 52 iterations for all the

containers plus 52 iterations of a stack, thus making the worst case a total of 104 iterations (arguably $O(1)$, if the number of possible containers remains constant).

2.3.3 Bottom containers and top containers

Because iterating over the bottom containers and top containers is a common occurrence, we can save in two lists the top containers and the bottom containers. Such lists would have to be updated when each configuration generates all possible moves from that configuration, which is not a exactly a fast operation even though at most the lists would have 52 elements each, which would also have to be replicated for each of the 52 possible moves, causing $52x52$ operations in the worst case.

To avoid such a cost, which would arguably not be worth the gain in performance for merely being able to efficiently iterate over the bottom and top containers, lazy removal from these lists is employed, which is only updated when the configuration has to be copied. As these lists get iterated and copied, we can check if the top and bottom containers they have are really on the top and bottom positions respectively.

2.3.4 Hashing

Because hashing a configuration of containers is a crucial operation for the A* to be efficient, besides caching the hash we can make hash generation even more efficient than having to calculate from the start for each configuration of layouts.

An efficient hashing of a configuration would be to have 3 distinct prime numbers for each array. The hash h would then be calculated by cumulatively adding the index of each container c that exists multiplied by it's array prime p to the power of it's index i:

$$h = h + c_i \times p^i$$

This then allows us to as we generate configurations originating from moves of a configuration (successors of a given configuration), automatically calculate the hash of the successor without having to calculate it's whole hash from the start by applying the following difference:

$$h_{sucessor} = h_{base} + ((c_{i,base} - c_{i,sucessor}) \times p^i))$$

2.4 Algorithm comparison

To compare both algorithms, the following parameters and metrics will be compared and/or taken into consideration:

- Rough time estimate.
- Nodes expanded.
- Nodes generated.
- Solution length.
- Penetrance.

2.5 Object oriented philosophy

The given interpretation described in Section 2.3 could be implemented in many different ways. This section explains the reasons as to why things are structured as they are in the source code.

Admittedly the interpretation described is very efficient, but it isn't without its mental overhead for someone reading, modifying or appending the code.

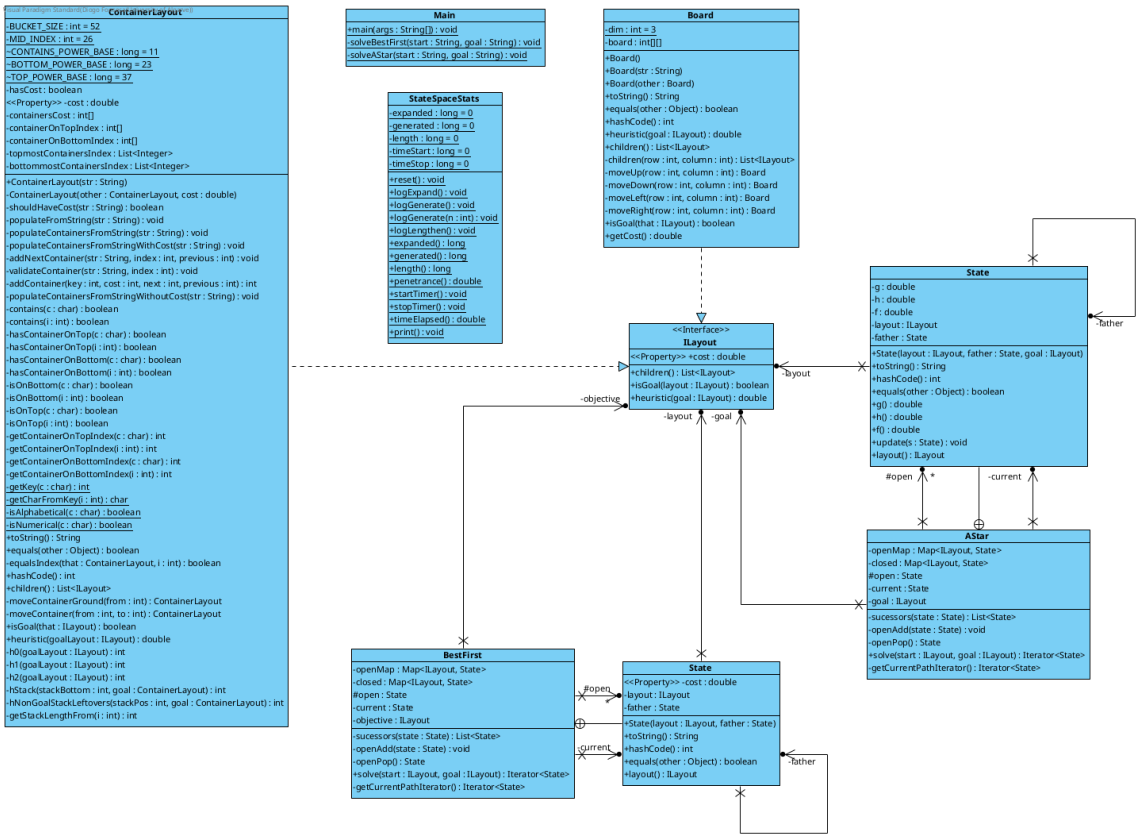
2.5.1 utility of implementing best-first algorithm

The implementation of the best-first algorithm has no meaning. It is only present due to a previous assignement, such an algorithm could be achieved using A* by simply making a heuristic that is always 0.

It could very well be implemented as a derived class of A* that overrides the heuristic to always return 0.

2.5.2 UML

The following UML was automatically generated using the assistance of the software Visual Paradigm and slightly adjusted for visual clarity.



2.5.3 Statistics

A small static class "StateSpaceStats" was made to save all the statistics such as the number of nodes expanded or number of nodes generated.

2.5.4 Strategy pattern

The code follows the strategy pattern, any configuration that follows the ILayout interface can then be solved by an algorithm that solves problems specified by the ILayout interface.

2.5.5 Duplicate "State" inner class

The code contains two algorithms: best-first and A*. Both of them contain an inner class called "State" thus having BestFirst.State and AStar.State.

The philosophy behind having two distinct "State" inner classes comes from their differences as much as the scalability of the code. If we had a dedicated "State" class, best-first would have unnecessary information such as the variables "h" and "f" used in A*.

Even if given the argument that it wouldn't hurt to have a little overhead, there's another key point that makes this not a desirable philosophy: The A* State class calculates it's heuristic and "f" (increasing it's principle of locality), both operations aren't supported by best-first as it doesn't need nor use a heuristic.

By far the greatest argument however, is that when this scheme is scaled to other algorithms, such as simulated annealing or even an adversarial search algorithm, the state once again changes it's "shape", and thus this project is done with the philosophy that each algorithm should implement it's own State class (or not, or even something completely different).

2.5.6 A* returns a path

Given the problem statement, it is sufficient for the A* to simply get the minimal cost, even if it does know the path, it need not return it which adds a slight performance overhead.

Still the code is returning a path. That is because the A* algorithm implemented is made with re-usability in mind, and if the algorithm is then again needed, and a path is also needed, the algorithm already does so.

An A* that returns a path instead of only the cost is obviously a superset of an A* that returns only the cost, and the performance loss for most problems is negligible.

2.6 Unit Testing

Unit tests were developed for every class, every edge case or pitfall that could be thought of has been made into a test.

Chapter 3

Results and Discussion

Some considerations when interpreting the following data:

- initial node counts as an expanded node
- initial node counts as a generated node
- initial node counts as a solution length

It should also be taken into consideration that the time elapsed is a rough time estimate, and should by no means be taken as empirical data. It is only present to get a rough estimate.

3.1 Heuristic best case

The best case for the heuristic is when all containers need **at most** one move to reach their goal position **and** when they do need more than one move, this second move is not cyclic by nature (like swapping two containers) and goes only "one way". In this case $h = h*$ when $h*$ is the real cost to the goal.

This does not mean that it will have the exact same nodes expanded as there is solution length, this is because there might be generated nodes of equal weight.

3.1.1 Uniform weights

INITIAL CONFIGURATION
A1B1C1D1E1F1G1H1I1

FINAL CONFIGURATION
A B C D E F G H I

Note how in this example, A* has the same value for nodes expanded and solution length. This means that it went directly to the solution.

This is the best case scenario and will always happen as long as there are not many valid solutions, making the amount of paths to explore greater, nor there are any cyclic moves.

As explained in Section 2.2.3, we know exactly how many containers must be moved at least once, and as such we know exactly what the path of least weight is in this case.

Table 3.1: **A***

Time elapsed (ms)	5
Nodes expanded	9
Nodes generated	170
Solution length	9
Penetrance	0.052941

Table 3.2: **Best-first**

Time elapsed (ms)	102
Nodes expanded	3 408
Nodes generated	50 344
Solution length	9
Penetrance	0.000179

In contrast, the best-first search has to search every single node up until it finds the goal, which makes for a very inefficient search in comparison.

These two metrics is all we really need to compare the two, but the rough time estimate, nodes generated and the penetrance also show a huge gap in performance from the A* to the best-first search. The penetrance in particular is 2 orders of magnitude greater, and there are $\sim$29614% more nodes being generated in the best-first search.

3.1.2 Ascending weights (best-first's best case)

INITIAL CONFIGURATION
A1B2C3D4E5F6G7H8I9

FINAL CONFIGURATION
A B C D E F G H I

Table 3.3: **A***

Time elapsed (ms)	3
Nodes expanded	9
Nodes generated	170
Solution length	9
Penetrance	0.052941

Table 3.4: **Best-first**

Time elapsed (ms)	54
Nodes expanded	2 058
Nodes generated	39 443
Solution length	9
Penetrance	0.000228

Just like the case in 3.1.1, this case is going to be solved in exactly the same manner, since each container needs to be moved at most once, and thus $h = h*$, when $h*$ is the real cost to the goal.

In other words, if all containers must be moved **at most** once, then the implemented heuristic (h_1 alone) will always perform the same, going straight for the solution, which we can again confirm by this example.

The best-first search performed relatively better when comparing with the results from 3.1.1, this is to be expected as the path with least cost is *generally* to remove a container from the initial stack. It could also be said that because of this, this case is generally **best-first's best case**.

Even at it's best, however, it cannot even begin to be compared to the A* algorithm, which essentially solved the problem in a linear fashion (Note that this is not true linearity, and such is not being claimed, as it will still generate the nodes in an exponential fashion).

3.1.3 Descending weights

INITIAL CONFIGURATION
A9B8C7D6E5F4G3H2I1

FINAL CONFIGURATION
A B C D E F G H I

Table 3.5: **A***

Time elapsed (ms)	5
Nodes expanded	9
Nodes generated	170
Solution length	9
Penetrance	0.052941

Table 3.6: **Best-first**

Time elapsed (ms)	497
Nodes expanded	44 729
Nodes generated	576 665
Solution length	9
Penetrance	0.000056

The A* as explained in 3.1.1 and further developed in 3.1.2, will still have the exact same result.

As for the best-first search, this is it's worst case in configurations where containers need to be moved at most once *to the ground*. This is because inversely to 3.1.2 it will prioritize moving the lower-cost containers on the ground than moving the containers from the big stack to the ground. As we can see by the results, it performed **considerably** worse in this case when comparing with the prior configurations.

For a sense of scale, this simple case that was solved in as many steps as there is solution length by the A* algorithm, the best-first search had to generate ∼339215% more nodes than it's A* counterpart. For a sense of scale, that would be roughly the same ratio as a baseball when compared to the titanic.

3.1.4 Several stacks

INITIAL CONFIGURATION
A1B1 C1 D1E1 F1G1H1I1

FINAL CONFIGURATION
CBGEAD F IH

Table 3.7: **A***

Time elapsed (ms)	5
Nodes expanded	10
Nodes generated	176
Solution length	8
Penetrance	4.455E-2

Table 3.8: **Best-first**

Time elapsed (ms)	2 310
Nodes expanded	197 987
Nodes generated	3 044 188
Solution length	8
Penetrance	2.628E-6

This is an interesting example to discuss, since it is not obvious what containers should be moved where first, although there *are* several solutions where we need only move a container *at most once*. As per shown in all the cases prior to this one, this means that the heuristic will know accurately the cost from any one node to the end goal.

If such is the case, then why do we have 10 nodes expanded? Where did the 2 extra expansions came from? This is easy to deduce.

At some point in the algorithm, even though it knows exactly how much a node will cost to get to the goal, two nodes had exactly the same cost, and thus there was no choice but to explore both to know which would lead to a lower weight path.

Even in these "worst cases of the best cases", it still completely outperforms the best-first, which had to generate 3 *million* nodes to finally find the solution.

In this example the best-first performs about 6 times worse than in case 3.1.3! This is because the initial branching factor is much worse, as it has to start with 4 distinct stacks.

3.2 Best-first worst case

3.2.1 Maximizing stacks

INITIAL CONFIGURATION
A1 B1 C1 D1 E1 F1 G1 H1 I1

FINAL CONFIGURATION
ABCDEFGHI

Table 3.9: **A***

Time elapsed (ms)	5
Nodes expanded	9
Nodes generated	241
Solution length	9
Penetrance	3.734E-2

Table 3.10: **Best-first**

Time elapsed (ms)	25 743
Nodes expanded	4 594 622
Nodes generated	33 223 249
Solution length	9
Penetrance	2.709E-7

When maximizing stacks, we are generating the worst case for Best-first state. This is due to best-first having the largest possible initial branching factor. This example also minimizes the best-first search's penetrance, having the highest amount of nodes generated in proportion to it's solution length.

By no surprise, we get a *huge* difference in performance, as Best-first generates ∼13 785 580% more nodes than A*. Again for visualization sake, that is the same ratio as the diameter of a baseball when compared to the Burj Khalifa (plus a large margin).

3.2.2 Max length descending weight

INITIAL CONFIGURATION
A9B8C7D6E5F4G3H2I1

FINAL CONFIGURATION
IABCDEFGH

Table 3.11: **A***

Time elapsed (ms)	12
Nodes expanded	100
Nodes generated	1 802
Solution length	17
Penetrance	9.43E-3

Table 3.12: **Best-first**

Time elapsed (ms)	26 955
Nodes expanded	4 592 974
Nodes generated	33 223 249
Solution length	17
Penetrance	5.117E-7

First note how this example maximizes the number of moves m. As $m = L - 1$, L being the solution length (because we don't need to make a move to get the initial node, but it is counted in the solution length). Then for this case $m = 16$. It was proven in Section 2.2 that if n is the number of containers, $m_{max} = 2n - 2$. Which then leads us to conclude that in this configuration $n = 9$ and $m_{max} = 16$! Which means we successfully maximized the number of steps to reach the goal ($m = m_{max} = 16$).

This is also a perfect visualization as to why the maximum number of moves must be $2n - 2$, which can then be interpreted as: "every container will be moved twice, except for two, which will be moved only once", the two containers that won't be moved twice are the one at the bottom of the goal state, "I", which will only have to be moved once to the ground, and the second will be the first container of the initial state, "A", which will only have to be moved once from the ground to the top of "I".

In such a configuration we maximize the amount of nodes with the same cost, resulting in the outcome we see where A* needs to a bit more to find the solution, as further explored in 3.3.2.

Even with A* in it's worst case, the best-first search is still nowhere close to it's performance, being in it's second worse case (albeit close to the worst case).

3.3 Heuristic worst case

3.3.1 Worst heuristic estimation

INITIAL CONFIGURATION
A1B1C1D1 E1F1G1H1I1

FINAL CONFIGURATION
AIHGF EDCB

The heuristic's worst estimation is when it can't discern that a container that will be moved twice, will be moved twice. This happens when in it's initial configuration a container below it won't be below it in the goal configuration, even though it'll be moved twice.

The only way to reproduce that condition, is to have a configuration where two containers need to swap places. This can even be extended to a "chain" of containers that need to be swapped (i.e. A needs to be swapped with B, B needs to be swapped with C, C needs to be

Table 3.13: **A***

Time elapsed (ms)	32
Nodes expanded	41
Nodes generated	466
Solution length	11
Penetrance	2.361E-2

Table 3.14: **Best-first**

Time elapsed (ms)	10 304
Nodes expanded	1 156 958
Nodes generated	14 238 148
Solution length	11
Penetrance	7.726E-7

swapped with A). Such a pattern cannot easily be exploited in a heuristic, as it would have exponential time.

In this case in particular, all the containers (except the base ones) in a stack need to be swapped out with the other, this causes one of the stacks to have to be "dismantled", or in other words, all it's containers need to be placed in the ground, then the other stack can be moved to the first one, and finally the containers on the ground can then be reassembled in the second stack. This causes a small heuristic underestimation, as it doesn't take into account that the containers in the first stack need to be moved twice.

Even though the heuristic underestimates the cost to the goal state in this scenario, this is not where it performs the worst.

3.3.2 Worst performance overall

INITIAL CONFIGURATION
A1B1C1D1E1F1G1H1I1

FINAL CONFIGURATION
IABCDEFGH

Table 3.15: **A***

Time elapsed (ms)	7
Nodes expanded	100
Nodes generated	1 802
Solution length	17
Penetrance	9.434E-3

Table 3.16: **Best-first**

Time elapsed (ms)	27 073
Nodes expanded	4 508 035
Nodes generated	33 223 249
Solution length	17
Penetrance	5.117E-7

The worst performance overall comes from solution length, as the heuristic does not increase/decrease performance with weights. So naturally, the worst case for the heuristic is when we maximize the solution length as this leads to there being several solutions to be explored with the same cost.

3.4 Random case

INITIAL CONFIGURATION
A2F7E9 D2 G1K4B6J2

FINAL CONFIGURATION
J D GB EKFA

Table 3.17: **A***

Time elapsed (ms)	5
Nodes expanded	21
Nodes generated	403
Solution length	8
Penetrance	1.985E-2

Table 3.18: **Best-first**

Time elapsed (ms)	12 406
Nodes expanded	122 788
Nodes generated	1 145 784
Solution length	8
Penetrance	6.982E-6

This case has no other purpose than to serve as a point of reference to what a random layout's statistics would look like. It is neither a best case or a worst case for either algorithm.

3.5 Benchmarking

A very crude, non-empirical benchmark was done to have a clearer boundary of the limits of the current heuristic implementation. A random 28-container pair was generated and fed into the A*. The results lie below.

INITIAL CONFIGURATION
C5A1R3c2 B9r2d5b5G9h6g4H1a8 D5t2j3y4u6 z3T1J7Y2U3 k2l3P1O3K7

FINAL CONFIGURATION
CcAaBbRrdHhGg zDJjYyTUktuKlPO

Table 3.19: **A***

Time elapsed (ms)	447 478
Time elapsed (min)	7.46
Nodes expanded	39 676
Nodes generated	3 353 238
Solution length	49
Penetrance	1.461E-5

It took around $\sim$7.5min for such a problem to be solved. All with 3M nodes being generated, which further justifies the position not to use IDA* for this problem *with the given constraints*.

Chapter 4

Conclusions

This project aimed to show the difference between informed and uninformed searched, by comparing their relevant statistics. A second goal of the project was to explain and justify the ideas behind the code in the project.

We can conclude from the results and their respective analysis in Chapter 3 that for the problem solved (and in general, for any admissible heuristic that does give some information about the problem) A* massively outperforms a best-first search, even when comparing the A*'s worst case 3.3.2 with the best-first's best case 3.1.2.

The A*'s performance comes from the fact that it has some information about the problem, and so it can get to it's goal without having to go through paths it knows aren't as good. Even if evaluating a node has some extra performance overhead (which very much is the case in this project), the fact that it has a rough idea where to go makes it vastly more performant than an uninformed search, as the overhead of heuristic calculation is usually much more efficient than testing every possible path. It should be noted however that even with this massive gap in performance, both scale exponentially with input size.

It was also explained 3.3.1 that the heuristic does have an underestimation problem in very specific cases (cyclic container dependencies), but the performance loss is not significant. The characteristic that does impact the heuristic's performance however, is solution length, especially when there are multiple solutions present, as the heuristic will have to go through all of them until it finds a solution.

The best-first's worst case was shown to be generally when we have a large number of stacks with uniform weight 3.2.1, as it will have the highest initial branching factor.

Chapter 5

Reflection

The project as a whole and it's key observations went exactly as expected. Due to time constraints it was not possible, but if the project were to be expanded on, it would also be interesting to see and compare other statistics such as branching factor.

The code overall is very efficient, and although the code could be refactored to be more performant (mostly in heuristic calculation), for the purpose of this report such was not required.

Something that could also be of interest for further analysis but mostly as a learning exercise, would be to implement the simulated annealing algorithm, even if explained to be as good or worse than A* for this case in 2.1. In the original plan for this report, the three algorithms were to be compared, but due to time constraints simulated annealing had to be cut from this report.

Another thing that could greatly improve this report, and perhaps the most important one, is the inclusion of diagrams and graphs explaining the logic in order for the reader to visualize what is being explained. The explanations sometimes are very abstract, and without previous knowledge of the problem it might also be very hard to decipher what is being explained.

References