Universität Bielefeld

Fakultät Wirtschaftswissenschaften
Lehrstuhl für Decision and Operation Technologies

Bachelorarbeit

im Studiengang Wirtschaftswissenschaften

Deep Reinforcement Learning for the Block Relocation Problem

vorgelegt von
Timo Kubitza

Matrikel-Nr: 2794825

1. Prüfer/in: André Hottung
2. Prüfer/in: Daniel Wetzel

August 9, 2019

Abstract - englisch

The unrestricted block relocation problem (BRP) is an optimization problem, often encountered at terminals, where containers are stacked onto each other. The goal is to minimize the number of relocations needed to remove all containers in a given order. Current solutions include mathematical models, as well as fast heuristics. This solution integrates Deep Learning with tree search algorithms, to compete with state-of-the-art heuristics. The method of Expert Iteration is adapted to the BRP, by training networks on instances with very few containers first, and subsequently using these models to solve slightly harder problems. This process is repeated, until full instances can be solved by the tree search algorithms. The final model can then generate solutions, with similar quality to current state of the art heuristics, with no human input whatsoever.

Keywords: Block Relocation Problem, Tree Search, Deep Learning, Reinforcement Learning

Abstract - deutsch

Das „unrestricted block relocation problem“ (BRP) ist ein Optimierungsproblem, welches oft an Häfen vorgefunden wird. Das Ziel ist es, Container in einer vorgegebenen Reihenfolge von mehreren Stapeln zu entfernen und dabei die Anzahl der benötigten Bewegungszüge zu minimieren. Aktuelle Lösungen umfassen mathematische Modelle zum Finden von optimalen Lösungen sowie schnelle Heuristiken. Diese Lösung ersetzt Heuristiken durch neuronale Netzwerke, welche dabei helfen, den Suchbaum zu verkleinern. Die Methode der „Expert Iteration“ wird adaptiert, um es für das BRP zu benutzen. So werden erst Instanzen mit sehr wenigen Containern gelöst, um mit den generierten Daten die Netzwerke zu trainieren. Diese können dann benutzt werden, um etwas schwierigere Instanzen zu lösen. Dieser Prozess wird wiederholt, bis volle Instanzen gelöst werden können. Das finale Modell ist schließlich in der Lage, Lösungen, vergleichbar mit den besten Heuristiken, zu generieren.

Stichworte: Block Relocation Problem, Tree Search, Deep Learning, Reinforcement Learning

Contents

1 Introduction **1**
- 1.1 Problem Description . . . 1
- 1.2 My solution approach . . . 2

2 Literature **3**
- 2.1 Problem Distinction . . . 3
- 2.2 Restricted Block Relocation Problem . . . 4
- 2.3 Unrestricted Block Relocation Problem . . . 4
- 2.4 Deep Learning Tree Search for CPMP . . . 5

3 Solution Approach **7**
- 3.1 Deep Learning . . . 7
- 3.2 Deep Learning Tree Search . . . 8
  - 3.2.1 Application for CPMP and BRP . . . 8
  - 3.2.2 Value Network . . . 8
  - 3.2.3 Policy Network . . . 9
- 3.3 Policy Iteration . . . 9
- 3.4 Policy Iteration for the BRP . . . 10
- 3.5 Search Strategies . . . 11

4 Computational results **13**
- 4.1 Training Process . . . 13
- 4.2 Comparison with other heuristics . . . 15
- 4.3 Interpretation . . . 17

5 Conclusion **19**

Bibliography **21**

List of abbreviations

BRP	Block Relocation Problem
BFS	Breadth first search
CPMP	Container Pre-Marshalling Problem
DFS	Depth first search
DAG	Directed acyclic graph
EXIT	Expert Iteration

1 Introduction

1.1 Problem Description

In the BRP, n containers (blocks) have to be retrieved from W Stacks in a given order. Container 1 has to be retrieved before container 2, which has to be retrieved before container 3. However, a container can only be retrieved if it is at the top of a stack. If this is not the case, the blocking containers have to be moved to other stacks. The height of a stack is limited by H_{max}. Moving a container from one stack onto another is called a relocation move. The objective is to minimize the number of relocation moves necessary to retrieve all containers in the correct order.

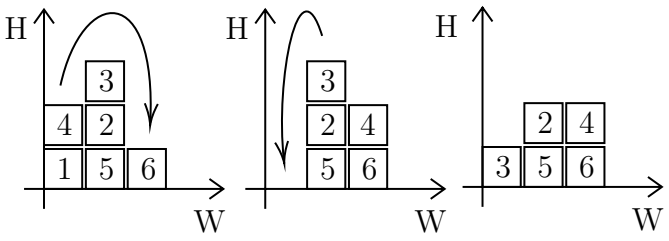

Figure 1.1: Optimal solution for a small instance of the BRP

Moving the 4 on top of the 6 is a relocation move, whereas the removal of the 1 is a retrieval move. Once no container is above a container with a lower retrieval number, the instance is solved, as all containers can be removed without any relocation moves.

The BRP can be represented as a graph, where each configuration is a state and connected to all other configurations that can be achieved by one move with an edge weight of 1. This creates a directed acyclic graph $G = (V, A)$, where the shortest path to an empty state is the optimal solution. Solving the problem by brute-forcing is not feasible due to the high branching factor of the search tree. Caserta et al. (2012) have proven that the BRP is NP-Hard. Thus, heuristics are used to find good solutions in a very short amount of time.

1.2 My solution approach

My solution will try to use neural networks to guide the tree search. A policy network will rank all possible moves on the likelihood that they are the best. A value-network will evaluate the remaining number of relocation moves until the problem is solved. With these two networks, various tree search algorithms can be used. This approach was introduced by Hottung et al. (2017) for the Container Pre-Marshalling Problem, which is similar to the BRP. They trained the neural networks on optimal solutions and used the networks to guide the search, achieving a low gap to optimality. In contrast to the CPMP, the BRP gets smaller, because containers are removed and the problem thus gets easier to solve. I will leverage this fact by solving small and easy problems first and train networks on the resulting data. Once easy problems can be solved, slightly harder problems will be solved using the trained networks and DLTS. This process can be repeated, until we arrive at the full problem. This means that no heuristics or exact methods are used to generate data.

2 Literature

2.1 Problem Distinction

There are two variations of the BRP; the restricted and the unrestricted BRP. In the restricted version of the problem, only containers that block the container to be retrieved next may be moved. In other words, either the next container is already at the top of a stack and can be retrieved, or there is a container blocking it. In case of the blockage, the blocking container has to be moved to another stack and no other moves are allowed. From now on, this assumption is called $A1$, as named by Caserta et al. (2012). The unrestricted version does not have this assumption. This means that a lot more moves are possible and strategic relocations of other stacks can yield better solutions than the restricted version. This also makes the problem much harder to solve to optimality, as the branching factor of the search tree gets much wider. Where for the instance size of 5x5 there are four possible arcs at each state in the restricted version, the search tree for the unrestricted version has a branching factor of 20.

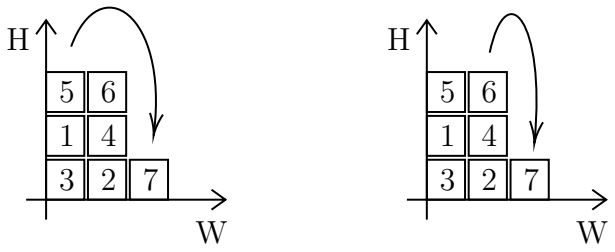

Figure 2.1: Optimal moves for the restricted and unrestricted BRP

Figure 2.1 shows how the unrestricted version can yield better results, because it does not have assumption $A1$. Strategically moving the 6 onto the 7 reduces the total number of moves by one in comparison to the restricted version. It is difficult to find good reasons for assumption A1, other than making the problem more tractable. In this thesis, I will consider the unrestricted version of the BRP. Most published research on the BRP, however, only considers the restricted version.

Another distinction to make is the maximum height of the stacks. The $H_{max} = unlimited$ version, meaning stacks can get as high as they want, does not make

much sense in practice. Both the $H_{max} = H + 2$ as well as $H_{max} = 2H - 1$ versions are applicable in real life. For this thesis I will focus on the $H_{max} = H + 2$ variation. Most heuristics, as well as DLTS, work with all three maximum heights.

2.2 Restricted Block Relocation Problem

The problem was first introduced by Kim and Hong (2006), where they present a branch-and-bound algorithm to solve instances with up to 30 containers in an hour. Additionally, they developed a heuristic to yield good solutions in less than two seconds. They propose a decision rule by using an estimator for an expected number of additional relocations for a stack. The solutions generated by this heuristic have a gap to optimality between 2.9% (3x3) and 16.1% (5x6).
Caserta et al. (2012) prove that the BRP is NP-Hard and present a mathematical model to solve small instances of the BRP. With some assumptions, they define a linear programming model for medium-sized instances.
Caserta et al. (2011) present a corridor method algorithm that explores a limited amount of solutions with a dynamic programming algorithm. Wu and Ting (2010) design a beam search that is similar to the breadth-first branch-and-bound scheme, but only few nodes in each level are kept. A look-ahead heuristic is used to evaluate nodes and select the ones to keep at each level.
Tanaka and Takii (2016) present a new lower bound for the restricted BRP. Using the improved lower bound allows a branch-and-bound algorithm to solve almost all instances to optimality. Whereas small- and medium-sized instances (up to 6x7) can be solved to optimality rather easily and fast, bigger instances cannot all be solved within the time limit of 1800 seconds.

2.3 Unrestricted Block Relocation Problem

Tricoire et al. (2018) propose two new kinds of operations. They define the term *conflict* as a situation where a given item is located above an item with a smaller index, thus blocking at least one item. A **safe relocation** is defined as a move which does not increase the minimum number of moves. The minimum number of moves for a given state can be defined as $NR + LB$, where NR is the number of moves that happened already and LB is the lower bound for the given state. A safe relocation increases the number of moves, $NR + 1$, but decreases the lower bound by one, $LB - 1$, thus keeping the minimum number of moves the same. **Decreasing sequences** are defined as consecutive elements that always have a bigger index than the one below them and also don't include the lowest index item. Consider the example stack of (1,3,4,5), where 1 is at the bottom and 5 is at the top. The decreasing sequence (5,4,3) can be identified. If it is possible to safe relocate the 5

onto another stack, the rest of the decreasing sequence can also be safe relocated.
They also present a new metaheuristic framework called **rake search**. In the BRP, the first few moves are crucial to finding a good or optimal solution. In rake search, the search tree is conducted in a breadth-first-search procedure, generating level after level, until a level with w or more nodes is reached. At that point, all nodes of this level are used as a starting point for multiple fast heuristics. They use four of their own heuristics, thus $4w$ solutions are generated in total. Additionally, they present the **pilot method**, a constructive metaheuristic where at each iteration every possible construction step is evaluated using a greedy construction algorithm, although other construction algorithms may be used.
Comparing their heuristics and frameworks on the Caserta instances, their solutions achieve better or equal quality results compared to other heuristics.

Feillet et al. (2019) propose the first local search type improvement heuristic for the BRP. It relies on dynamic programming to identify the locally optimal sequence of moves of a given container. Applying their method on the rake search solutions of Tricoire et al. (2018), some improvement can be identified. While there are no improvements for instance sizes up to 5x9 and only very few improvements for instance sizes up to 10x10, improvements of up to 50 % can be seen on larger instances. This can be explained by the fact that current state-of-the-art heuristics solve the unrestricted BRP with a very low gap to optimality. However, this also shows that, especially for large instances, unexpectedly high potential for improvement exists.
Tanaka and Mizuno (2018) propose several dominance properties to eliminate unnecessary nodes in the search tree to improve the efficiency of a branch-and-bound algorithm. Additionally, they present a new lower bound of the number of relocations necessary. With a time limit of 1800 seconds, their new dominance rules allow them to solve almost all instances of the Caserta benchmark instances to optimality.

2.4 Deep Learning Tree Search for CPMP

This thesis builds upon the work of Hottung et al. (2017) for the Container Pre-Marshalling Problem. The CPMP has the same setup as the BRP but does not allow retrievals during shuffling. Each container has a retrieval number. The goal is to shuffle the containers in such a way, that containers can then all be retrieved in the correct order, once the shuffling is finished. This makes solving the problem even harder than the BRP, because the CPMP always has the full number of containers at every step of the search tree.
Their work uses neural networks to guide the tree search algorithms. Each state of the CPMP (and the BRP) can be represented with very little data, allowing the state to be easily inserted into neural networks. These can, after being trained on optimal solutions, make predictions about how many moves are left, as well as which

moves of a given state are likely to be the best. Using this information, the search tree can be heavily pruned.
Their benchmarks show that DLTS outperforms current state-of-the-art metaheuristics by finding gaps to optimality between 0-2%, as opposed to 6-15% for other solutions.
Because of the similarity of the two problems, it is very likely that DLTS can be successfully applied to the BRP. The question is rather if the performance compares to current state-of-the-art heuristics, as these already solve most small and medium-sized instances to optimality in a few seconds.

3 Solution Approach

3.1 Deep Learning

Deep Learning is a machine learning method which was developed in the early 1970s, but has gotten a lot more attention in the past 10 years, due to an increasing amount of data and the steady rise in computational power. This section is going to explain briefly how neural networks work and what their strengths are. For reference, this section is based on the popular Deep Learning book by Goodfellow et al. (2016).

A neural network can be seen as a function $y = f(x)$ that maps an input x to an output y and tries to approximate some function f^*. A so-called classifier maps an input, for example a picture of an animal, to an output category, for example the class cats. A regressor tries to map the input, stock market data, to a numeric output, the price of a stock in a month.
This can be achieved by a feedforward network. A feedforward network is a directed acyclic graph that takes in values for x, does some intermediate computations, and then outputs y. The DAG specifies how multiple functions are combined. Each function, for example $f^{(1)}$, or $f^{(2)}$, a so called layer, takes in the output of the previous layer as an input and computes some output, forming a chain of the structure $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$.
Each layer consists of multiple perceptrons. Each perceptron does the computation of $\mathbf{w} \cdot \mathbf{x} + b$, where $\mathbf{x}$ is the output of the previous layer, $\mathbf{w}$ are the weight parameters and b is some trained constant. An activation function, for example the sigmoid function, is applied on the result of the computation. This means that each perceptron has $\mathbf{w}$ and b as trainable parameters. Each added layer adds $layer1_{size} \cdot (layer2_{size} + 1)$ parameters, giving even small networks a lot of flexibility. These parameters can be trained by a method called gradient descent. Each parameter has a relationship with the error, computed by the loss function. The partial derivatives of each parameter measure the impact of change a parameter has on the loss. Each parameter is then nudged into the direction that minimizes the overall error. Doing this process repeatedly allows neural networks to approximate complex functions.
While there are many neural network architectures that excel at certain tasks, for example convolutional neural networks (CNN) at image recognition and recurrent neural networks (RNN) at natural language processing, this thesis will work with regular feedforward neural networks.

3.2 Deep Learning Tree Search

Deep Learning Tree Search for operations research problems was inspired by Alpha Go by Silver et al. (2016) and implemented for the CPMP by Hottung et al. (2017). Silver et al. (2016) used Neural Networks to evaluate board positions and possible moves in the game of Go, allowing a narrower and more accurate Monte Carlo Tree Search. Without heavy pruning of the tree, it would have been impossible to reliably look more than a few steps ahead, because the game of Go allows a lot of different moves at each turn. Using a policy network to imitate a strong Go player allowed the search algorithms to dismiss nodes that would never be reached in a high Elo match.

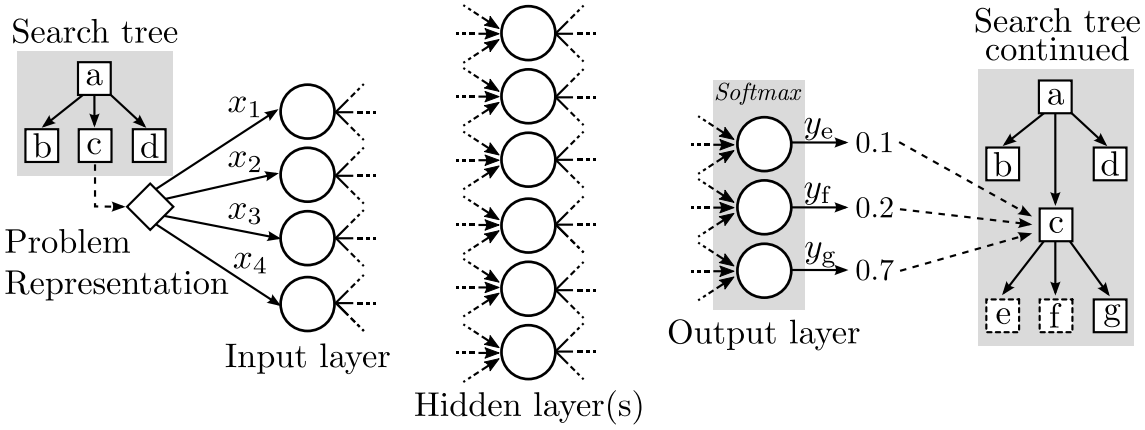

Figure 3.1: Deep Learning Tree Search by Hottung et al. (2017)

3.2.1 Application for CPMP and BRP

Both the CPMP and the BRP can easily be represented as a matrix, where each container is represented by the retrieval number and empty spaces as a zero. This matrix representation can be used as input for Neural Networks. To reduce the total number of parameters for a network, shared weights can be used, meaning that the same layers are applied to each stack, before they all feed into one big layer.

3.2.2 Value Network

The Value Network can be seen as a regressor that tries to guess the number of moves left, that the optimal solution would need to solve a specific instance. Because the problem gets smaller and easier the more containers have been removed, the value network easily predicts the number of moves left for a few containers. However, generally, the more containers are left, the harder the prediction. Search algorithms

can use this information to decide whether a certain state should be cut out of the search. This could be applied during depth first search the following way: A solution with ten relocations has been found, but there are many more nodes that have to be examined. At another node, which can be reached with five relocation moves, the value network predicts that another eight moves are necessary to solve the instance. Because it is highly unlikely that this state can yield a solution with only four relocation moves, the search stops at that node and the DFS algorithm continues with the next node. One important thing to mention is that the network can be wrong in both directions, so there is a distinct difference with a mathematical lower bound. For some states, the network may overestimate the required moves and an optimal solution may be thrown away by the search algorithm.

3.2.3 Policy Network

The policy network is a classifier. Each possible move, for example stack 1 to stack 2, stack 4 to stack 3, can be seen as a class. The network then tries to predict which move leads to the optimal solution. The network assigns each class a probability of being the optimal move.
This information can be used during search to reduce the branching factor of the search tree. Usually, out of all possible moves only a few can be considered useful, while a lot of moves are really inefficient. During DFS, the network output could be used in the following ways:
1. At each node a prediction is made and all moves are sorted in order of their assigned probability. Then, only the best k moves are being pursued.
2. Again, at each node a prediction is made and all moves are sorted in order of their assigned probability. Now, all possible moves that have an assigned probability over a certain threshold are pursued.

3.3 Policy Iteration

DeepMind's Alpha Go, which beat the world champion Lee Sedol, trained exclusively on games played by other world class Go players. Their next iteration, Alpha Go Zero, went on to become a much better Go player than the previous iteration. Silver et al. (2017) did this by letting versions of Alpha Go Zero play against each other and using these games to train a new version. While in the beginning all moves are made randomly, the neural networks learn the moves of the winning side. After the first training, two versions that are slightly better than the first play each other, generating data to be used for the next version. This process can be repeated indefinitely and allowed Alpha Go Zero to reach superhuman Go abilities, without any data generated by humans.
Anthony et al. (2017) presented the same approach, naming it Expert Iteration, or $EXIT$, in short. Their algorithm works very similar to Alpha Go Zero, achieving

very high Elo in the board game Hex.
What Go and Hex have in common, is that in both games random moves yield a winner, and thus data to be used for training. Data, to train the value and policy networks for the DLTS for the BRP, consists of complete solutions of many instances. This, however, cannot be achieved without both a strong policy and value network, because random moves will not yield an appropriate solution. While moving containers around randomly will lead to a solution eventually, the quality of the solution is not good enough to be considered a good training example. Additionally, because of the NP-hardness of the BRP, brute-forcing the problem is also not feasible, even for small instances.

3.4 Policy Iteration for the BRP

In contrast to the CPMP, the BRP has the property that it gets easier to solve the fewer containers are left. This property can be used to adapt the $EXIT$ algorithm and to apply it to the BRP. Instead of trying to solve full instances, instances with only a few containers are solved first. With only a few containers, solutions often only require less than three moves, which makes brute-forcing feasible again. The resulting data can then be used to train the value and policy networks for the first time.
These can now be used to solve instances with an additional container using DLTS. While the prediction quality will suffer under the fact that the training data had one fewer container, the models are still good enough to yield good solutions with DLTS. The process of $EXIT$ can be repeated with each additional container, until the full instance size is reached. Now, the $EXIT$ algorithm can be repeated indefinitely.

Algorithm 1 Adapted Expert Iteration Algorithm

1: $b = min\_containers$
2: $\pi_b = build\_networks()$
3: **while** $b < height^*width$ **do**
4: $\quad I = generate\_instances(b, num\_instances)$
5: $\quad S = treesearch\_solutions(\pi_b, I)$
6: $\quad \pi_{b+1} = train\_networks(\pi_b, S)$
7: $\quad b = b + 1$
8: **for** $i \leftarrow$ desired number of iterations **do**
9: $\quad I = generate\_instances(height * width, num\_instances)$
10: $\quad S = treesearch\_solutions(\pi_b, I)$
11: $\quad \pi_b = train\_networks(\pi_b, S)$

3.5 Search Strategies

Depth-first-search as well as limited-discrepancy-search are described in Hottung et al. (2017). While this thesis primarily works with a DFS algorithm, a breadth-first-search algorithm is used with only a few containers.
A BFS algorithm with a strong reliance on the value network is used during the beginning of the training. The algorithm, assisted by neural networks, is implemented the following way:
At each level, all nodes are expanded. The policy network can be used to reduce the number of expansions at each node. However, with only a few containers, this is not always needed. At the new level, each node is then evaluated by the value network. Now, the worse half of nodes gets cut out of the search. This strategy, combined with other pruning methods, can heavily reduce the search tree.

Algorithm 2 BFS with DLTS

```
1: function BFS − DLTS(s, k, p, m)
2:     current_level = [s]
3:     next_level = [ ]
4:     while no solution found do
5:         for state in current_level do
6:             new_states = best_next_states(state, k)
7:             next_level.append(new_states)
8:         if solved(next_level) then
9:             return solution
10:        new_width = max(m, len(next_level * p))
11:        current_level = best(next_level, new_width)
12:        next_level = [ ]
```

Parameter	Impact during BFS
s	Starting state of BRP
k	Only pursues best k moves at each state
p	Keeps p percent at each level
m	Minimum number of states to keep at each level

Table 3.1: Parameters of BFS with DLTS

4 Computational results

4.1 Training Process

Once all parameters are set, the program should be able to iteratively improve the capabilities of the networks and produce solutions to the benchmark instances at the end. This process can be tuned by many hyperparameters, which heavily influence the duration of the training, as well as the performance of the final model. In this section, I am going to present these hyperparameters and explain the impact each of them has on the final result.

Retraining Frequency

The $EXIT$ algorithm retrains a whole new model with every iteration. This, however, is not always necessary in this problem setting. Retraining from scratch takes significantly more time than improving the current model. Especially for small instances, retraining is not needed, because the problem is not that hard.
With medium-sized instances, a pattern can be seen in many cases. Without completely retraining the model every few iterations, performance may suffer at some point during the training. This often happens around the time where strategic relocation moves are required to get a good solution. With only a few containers, there is almost always a safe relocation for the container that's blocking the next item, as described by Tricoire et al. (2018). However, because not many examples of strategic relocations will be in the training set, learning this strategy is hard for an already trained model.
A good trade-off in regards to the retraining frequency depends on the instance size. For instances with more than 15 containers, completely retraining with every third additional container can boost the performance.

Instances per iteration

During the process of training the models, instances with an additional container are solved to generate training data. At each iteration of the $EXIT$ algorithm, n instances are solved by DLTS. While small instance sizes don't need that many examples, more training examples are always beneficial. Figure 4.1 compares the average solving time per instance during the training with the $EXIT$ algorithm. It shows that not many instances are needed for the training of the networks. While for the instance size of 4x4 5.000 instances per iteration show an increased performance over 2.000 instances, solving 40.000 instances does not yield any improvements. To

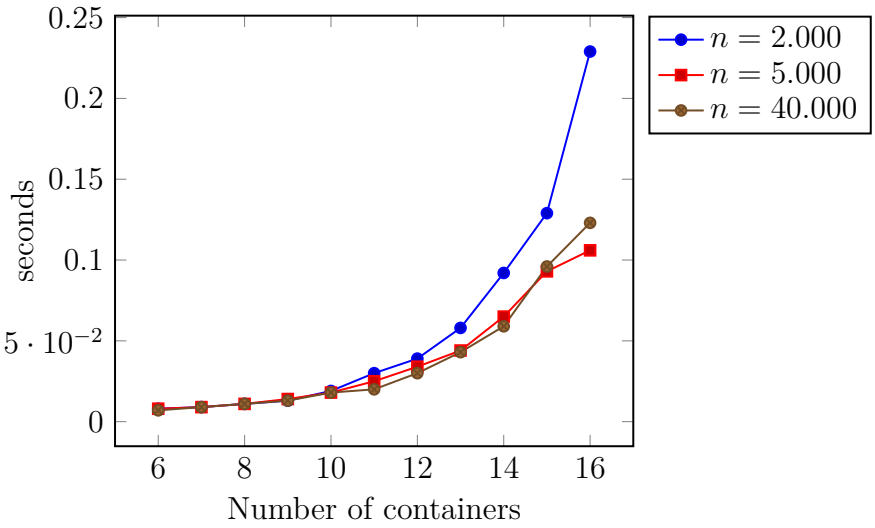

Figure 4.1: Comparison average solving times by number of instances per iteration for 4x4

additionally inflate the number of data points, permutations of each state can be used to generate new data. Figure 4.2 shows all possible permutations of an instance with width 3. Generally, there are $W!$ permutations. Using permutations allows the algorithm to generate a lot of data very fast.

Search Parameters

Neural networks need a lot of data to learn complex functions. During the training process, a trade-off between data quality and data quantity has to be made. Search parameters decide how thoroughly the algorithm searches the tree. Generally, the better the solutions, the more time it takes to compute them. Therefore, solving each instance to optimality and thus heavily reducing the amount of data generated is not the best strategy. Keeping the average solving time at less than 0.1 seconds for small instances and less than 1 second for medium-sized instances ensures that the training process is fast. Additionally, a time limit per instance can be set, depending on the instance size. This guarantees that the search algorithm doesn't spend most of its time on very few instances.

Training Duration

The training duration of the entire process depends on the instance size, the search parameters, how many instances are solved at each stage and on what hardware the training is run. Generally, most time is spent solving instances, with up to 85% of the total time. Because a DFS algorithm is used most of the time, using a GPU

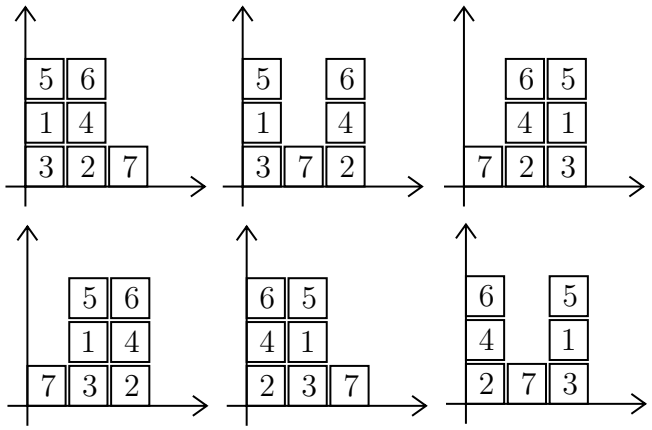

Figure 4.2: All possible permutations of an instance with width of 3 stacks

won't speed up the process much, because evaluating each state on a single GPU call is slower than on a CPU.
Table 4.1 shows how training time increases with each additional container. While instance sizes should not be compared, because they use different neural network architectures, search parameters and may run on different CPUs, it is easy to see that with each additional container, solving time per instance generally increases. Especially the last few containers heavily increase the solving time per instance.

4.2 Comparison with other heuristics

Table 4.2 shows how DLTS compares with current fast heuristics. These solve instances in less than a second, but most of the time finish instantly. LA-S1 are solutions provided by Petering and Hussein (2013). SM-1, SM-2, SmSEQ-1 and SmSEQ-2 are the heuristics based on safe relocations and decreasing sequences by Tricoire et al. (2018). Lastly, results by Jin et al. (2015) are included as JZW. We can see that DLTS beats those heuristics in terms of solution quality for multiple instance sizes. Often times, DLTS can beat the fast heuristics, even before the $EXIT$ algorithm has finished. Benchmarks during the training process, with two containers still missing, were already able to beat the fast heuristics a lot of times.

Table 4.3 shows how DLTS compares with current metaheuristic frameworks. Rake search, as well as the pilot method, have been mentioned earlier in this thesis. The CPU effort of both of these methods averages below 1 second on the Caserta instances, however, maximum CPU time may be higher, as there is a lot of variability in solving time. Some instances are really easy to solve to optimality and finish instantly, while others may take a couple of seconds. A time limit of 10 seconds is set

C	N	3x3	3x4	3x5	4x4	4x5
6	40.000	0.027	0.025	0.026	0.007	0.006
7	40.000	0.036	0.034	0.029	0.009	0.008
8	40.000	0.046	0.040	0.033	0.011	0.010
9	40.000	0.065	0.049	0.042	0.013	0.013
10	40.000		0.065	0.060	0.018	0.016
11	40.000		0.076	0.076	0.020	0.018
12	40.000		0.078	0.071	0.030	0.022
13	40.000			0.095	0.043	0.032
14	40.000			0.104	0.059	0.039
15	40.000			0.123	0.096	0.045
16	40.000				0.114	0.069
17	40.000					0.095
18	40.000					0.125
19	40.000					0.106
20	40.000					0.133

Table 4.1: Average time in seconds for one instance of C containers throughout the training process

H	S	LA-S1	SM-1	SM-2	SmSEQ-1	SMSEQ-2	JZW	DLTS
3	3	5.075	5.100	5.125	5.175	5.200	5.000	**4.975**
3	4	6.275	6.250	6.225	6.275	6.225	6.150	**6.075**
3	5	6.975	6.925	7.000	6.925	6.950	6.925	**6.850**
3	6	8.475	8.450	8.375	8.400	8.350	8.400	8.650
3	7	9.250	9.225	9.250	9.225	9.250	9.150	9.950
4	4	10.575	10.400	10.325	10.750	10.725	10.275	**9.750**
4	5	13.200	13.150	12.925	13.150	13.150	12.875	**12.475**
4	6	14.150	14.125	13.925	14.050	14.025	13.775	14.550
5	4	16.425	16.325	16.125	16.825	16.700	15.755	**15.175**
5	5	20.050	19.600	19.400	19.825	19.875	19.325	**18.500**
5	6	22.875	22.525	22.250	22.650	22.675	22.400	24.550

Table 4.2: Comparison with fast methods on Caserta instances, $H_{max} = H + 2$, Average number of relocations

for each instance, however the average CPU times lies far below that and rarely gets used during the benchmark. We can observe that, for small instance sizes, DLTS often achieves the performance of current state-of-the-art metaheuristics. While DLTS beats fast heuristics many times, current metaheuristic frameworks achieve slightly better results on the benchmarking instances. Having the best neural network architecture can speed up the training process tremendously.

H	S	Rake Search	Pilot Method	DLTS
3	3	**4.975**	**4.975**	**4.975**
3	4	**6.025**	**6.025**	6.075
3	5	**6.850**	**6.850**	**6.850**
3	6	**8.275**	**8.275**	8.650
3	7	**9.125**	**9.125**	9.950
4	4	**9.750**	9.975	**9.750**
4	5	**12.325**	12.350	12.475
4	6	**13.250**	13.375	14.550
5	4	**14.925**	14.975	15.175
5	5	**17.700**	17.825	18.500
5	6	**21.125**	21.150	24.550

Table 4.3: Comparison with advanced methods on Caserta instances, $H_{max} = H+2$. Average number of relocations

4.3 Interpretation

For larger and wider instances, performance of DLTS with $EXIT$ is not as good as rake search and the pilot method. This, however, does not necessarily mean that DLTS does not work as well on the BRP as it does on the CPMP. Rather, more tuning of the $EXIT$ algorithm is most likely required to achieve similar performance to results on the CPMP. Hottung et al. (2017) use optimal solutions, meaning that there is no bias in the training data. With this approach however, solutions that are used as training examples are not always optimal, leading the value network to be biased into predicting more moves, as well as giving the policy network a higher variance. It is very likely that once solutions are worse than the optimal solutions by some margin, performance decreases with each additional container.
Increasing the training duration for each experiment may allow this approach to achieve similar results to the state-of-the-art. For example, after reaching the full instance size during training, the models were not able to solve instance 38 for the size of $W = 4, H = 5$, even with a time limit of 20 seconds. However, after training with the $EXIT$ algorithm for an additional 7 hours, the instance could be solved in less than 0.5 seconds. A combination of improved neural network architecture, as well as an increased training time could allow this apporach to generate solutions similar to rake search.

5 Conclusion

In this thesis I have presented an adapted version of the $EXIT$ algorithm and applied it to the Block Relocation Problem. The computational results show that without any human input, the models are able to provide strong solutions in a few seconds or less. While current state-of-the-art heuristics and metaheuristics perform similarly well or a little better, these results are still noteworthy.
Results in this thesis are limited by computational resources. Letting the $EXIT$ algorithm run for a longer period of time and optimizing the neural network architectures for each instance size may improve the final results.
Even for the BRP, there still is some room for improvement. Many search algorithms and pruning techniques can be built upon the information the networks provide. Additionally, different neural network architectures could be applied.
Future research could try to apply DLTS to very large instances of the BRP. However, it stands to question whether networks can generalize among many containers, as well as very few. Results have shown that these large instances still have a large room for improvement.
Lastly, Reinforcement Learning has the ability to reach a superhuman skill level. This was shown by DeepMind's AlphaGo Zero. For optimization problems, there often exists an optimal solution, so the focus lies on reducing the time for constructing a good or optimal solution. Neural networks have the ability to understand complex patterns, which may not be easy to implement as heuristics, or haven't been discovered by researchers yet. Future research may apply DLTS to other optimization problems, where states can be abstracted by neural networks.

Bibliography

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

Marco Caserta, Stefan Voß, and Moshe Sniedovich. Applying the corridor method to a blocks relocation problem. *OR spectrum*, 33(4):915–929, 2011.

Marco Caserta, Silvia Schwarze, and Stefan Voß. A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, 219(1):96–104, 2012.

Dominique Feillet, Sophie N Parragh, and Fabien Tricoire. A local-search based heuristic for the unrestricted block relocation problem. *Computers & Operations Research*, 108:44–56, 2019.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

André Hottung, Shunji Tanaka, and Kevin Tierney. Deep learning assisted heuristic tree search for the container pre-marshalling problem. *arXiv preprint arXiv:1709.09972*, 2017.

Bo Jin, Wenbin Zhu, and Andrew Lim. Solving the container relocation problem by an improved greedy look-ahead heuristic. *European Journal of Operational Research*, 240(3):837–847, 2015.

Kap Hwan Kim and Gyu-Pyo Hong. A heuristic rule for relocating blocks. *Computers & Operations Research*, 33(4):940–954, 2006.

Matthew EH Petering and Mazen I Hussein. A new mixed integer program and extended look-ahead heuristic algorithm for the block relocation problem. *European Journal of Operational Research*, 231(1):120–130, 2013.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676): 354, 2017.

Shunji Tanaka and Fumitaka Mizuno. An exact algorithm for the unrestricted block relocation problem. *Computers & Operations Research*, 95:12–31, 2018.

Shunji Tanaka and Kenta Takii. A faster branch-and-bound algorithm for the block relocation problem. *IEEE Transactions on Automation Science and Engineering*, 13(1):181–190, 2016.

Fabien Tricoire, Judith Scagnetti, and Andreas Beham. New insights on the block relocation problem. *Computers & Operations Research*, 89:127–139, 2018.

Kun-Chih Wu and Ching-Jung Ting. A beam search algorithm for minimizing reshuffle operations at container yards. In *Proceedings of the International Conference on Logistics and Maritime Systems, September*, pages 15–17, 2010.

Versicherung

Name: Kubitza

Vorname: Timo

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen, als die angegebenen, Quellen benutzt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen habe ich als solche kenntlich gemacht. Diese Versicherung gilt auch für alle gelieferten Datensätze, Zeichnungen, Skizzen oder grafischen Darstellungen. Des Weiteren versichere ich, dass ich das Merkblatt zum „Umgang mit Plagiaten“[1] gelesen habe.

Bielefeld, den August 9, 2019 _____ Unterschrift

[1] www.wiwi.uni-bielefeld.de/organisation/pamt/~/organisation/pamt/uploads/PlagiatInfo-BlattStudenten.pdf