UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JEAN PERSI BOELTER

Learning Deadlocks in Sokoban

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. André Grahl Pereira

Porto Alegre
December 2018

ABSTRACT

In this thesis, we present an approach for deadlock detection in Sokoban based on neural networks. Sokoban is a challenging state space problem in artificial intelligence due to many characteristics, being the presence of deadlocks one of them. A deadlock is a state reachable from the initial state which cannot reach any goal state. An informed search algorithm aims to find in a state space an ordered sequence of actions that transform the initial state into a goal state. Deadlock detection is essential to increase the performance of an informed search algorithm. Pattern databases are the current state of the art heuristic for deadlock detection in Sokoban. We present methods to generate a training set and train a neural network to detect deadlocks. Our approach has a similar performance to a pattern database. When compared to the standard heuristic function of Sokoban we solved two more instances while exploring an order of magnitude fewer states.

Keywords: Sokoban. Deadlocks. Neural Networks. Heuristic Search.

Aprendendo Deadlocks em Sokoban

RESUMO

Nesta tese, apresentamos uma abordagem para detecção de deadlocks em Sokoban baseada em redes neurais. Sokoban é um problema de espaço de estados desafiador na inteligência artificial devido a muitas características, sendo a presença de deadlocks uma delas. Um deadlock é um estado alcançável a partir do estado inicial que não consegue atingir nenhum estado de objetivo. Um algoritmo de busca informada visa encontrar em um espaço de estados uma sequência ordenada de ações que transformam o estado inicial em um estado objetivo. A detecção de deadlocks é essencial para aumentar o desempenho de um algoritmo de busca informada. Pattern databases são a atual heurística estado da arte para detecção de deadlock em Sokoban. Apresentamos métodos para gerar um conjunto de treinamento e treinar uma rede neural para detectar deadlocks. Nossa abordagem tem um desempenho semelhante a pattern databases. Quando comparado com a função heurística padrão do Sokoban resolvemos duas instâncias a mais enquanto exploramos uma ordem de grandeza menos estados.

Palavras-chave: Sokoban. Deadlocks. Redes Neurais. Busca Heurística.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ANN	Artificial Neural Network
BFS	Breadth-First Search
CPU	Central Processing Unit
EMM	Enhanced Minimum Matching
GBFS	Greedy Best-First Search
RGBFS	Reverse Greedy Best-First Search
GPU	Graphics Processing Unit
IPDB	Intermediate Pattern Databases
MPDB	Multiple Goal State Pattern Databases
NN	Neural Network
PDB	Pattern Databases
REMM	Reverse Enhanced Minimum Matching

CONTENTS

1 INTRODUCTION

We can define a large number of problems as *state space* problems. The solution for these problems is an ordered sequence of *actions* that transform the given *initial state* into a *goal state*. Each of the actions transforms a state into another with an associated cost. The cost of a solution is the sum of the costs of all its actions. An optimal solution has the minimum cost among all solutions.

There are two classes of algorithms that search for a solution to a state space problem: *uninformed* and *informed search*. Uninformed search operates in a brute-force way, it has no additional information about the states and needs to expand all states, from the initial states, until it finds the goal. It is inefficient in larger problems, where the number of states in the state space is prohibitive for brute-force search. Informed search algorithms speed up the search by pruning states using additional information about the problem, usually provided by a *heuristic function*.

A heuristic function, or simply a heuristic, is a function that usually returns the remaining cost to reach a goal state from any given state. This information is used by the informed search algorithm to expand the most promising states, that are closer to the goal, first. However, computing the exact remaining cost is, in general, as hard as solving the problem, so heuristics provide estimates of the remaining cost. The quality of the heuristic directly impacts the performance of the search algorithm. In general, a search algorithm with more informed heuristics expands fewer states. The task of designing informed heuristics is an active research topic. A recent approach to automate the process of finding good heuristics is the use of machine learning algorithms.

Many state space problems are reversible, that is, for every action that transforms a state s to a state s', there exists an ordered sequence of actions that transform the state s' to the state s. When a problem is not reversible, an action that cannot be undone can lead to a state that has no solution, a *deadlock*. Deadlocks are states that cannot reach any goal state, while states that can still reach a goal are called *alive*. Good heuristics should be able to detect deadlocks, pruning them from the search and significantly reducing the number of expanded states.

Sokoban is a PSPACE-Complete (CULBERSON, 1999) problem and an example of a state space problem. In this problem a *man* needs to push all *stones* in a *maze* to certain marked positions called *goal squares*. The movement of the man is limited to *free squares*. Free squares are squares within the walls of the maze and not occupied by

stones. He can move to any adjacent free square or push adjacent stones if the square the stone is being pushed to is free. Deadlocks also play an important role in Sokoban. There are nontrivial placements of subsets of stones that can generate deadlock states. Solving Sokoban is a hard task which requires informed heuristics and deadlock detection for an efficient search algorithm.

Inspired by previous works on learning heuristics this thesis explores the potential of machine learning, namely neural networks, to detect deadlocks in Sokoban. The objective is to train a neural network that detects deadlocks and then use it together with a heuristic function to improve the performance of an informed search algorithm. After giving some background in the related areas, we discuss how to create the training set and the model for the neural network, then we present experiments that compare the neural network with Pattern Databases, the state of the art for efficient deadlock detection in Sokoban (PEREIRA; RITT; BURIOL, 2014).

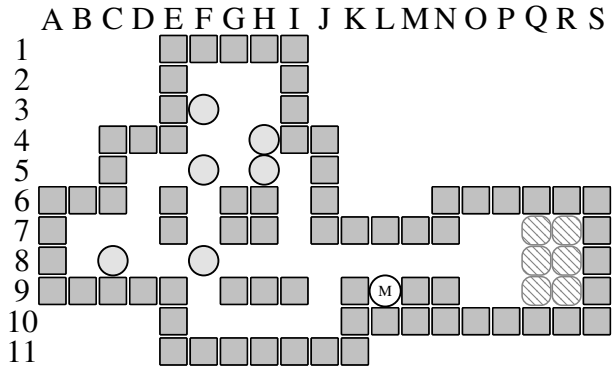
2 BACKGROUND

This chapter provides an introduction to the concepts and techniques used throughout this thesis.

2.1 Heuristic Search

This section gives an introduction to *heuristic search*. Most notations and definitions are based on the book *Heuristic Search: Theory and Applications* (EDELKAMP; SCHROEDL, 2011).

Figure 2.1: Instance #1 of xSokoban. The man is shown at L9, a stone at C8, a wall at A6, and a goal square at R8.



2.1.1 State Space Problems

A *state space problem* P can be defined as $P = (S, A, s, T)$ and consists of a set of states S, a finite set of actions $A = \{a_1, ..., a_n\}$ where each $a_i : S \rightarrow S$ transforms a state into another state, an initial state $s \in S$ and a set of goal states $T \subseteq S$. Using the Sokoban instance in Figure 2.1 as an example, the state from the figure is the initial state, the set of goal states are the states where all six stones occupy the six goal squares and the man is in any of the remaining free squares. An action in Sokoban corresponds to the man to push a stone to an adjacent free square. For example, in Figure 2.1 the man can push the stone at H5 to G5 because its reachable component includes square I5. However, the man cannot push the stone at F8 to G8 because its reachable component does not include square E8. A solution to a state space problem is an ordered sequence of

actions that transforms the initial state s into one of the goal states $t \in T$, it can be defined by $\pi = (a_1, ..., a_n)$ with $a_i \in A, i \in \{1, ..., n\}$. This solution also generates an ordered sequence of states $(s_0, ..., s_n)$ with $s_0 = s$, $s_n = t$, and applying a_i to s_{i-1} will result in s_i for $i \in \{1, ..., n\}$. State space problems can also be weighted, that is, every action has a cost, but for this thesis, we are only interested in unweighted state space problems. In unweighted state space problems, the cost of each action is the same, usually represented as one, making the cost of a solution π the number of actions in it.

2.1.2 Heuristics Functions

A *heuristic function* $h(u)$ is a state evaluation function that estimates, for a state $u \in S$, the remaining cost to a goal state $t \in T$. This is used to direct the search to the goal by expanding more promising states first.

A heuristic h is called *admissible* if it never overestimates the optimal cost of reaching the goal, and it is called *consistent* if, for a state u and its neighbor v, $h(u)$ is less than or equal to the cost from u to v plus $h(v)$. All consistent heuristics are admissible but not all admissible heuristics are consistent.

2.1.3 A* Algorithm

A* is an informed search algorithm that uses the evaluation function $f(u) = g(u) + h(u)$, where $g(u)$ is the cost of the path from the initial state to u and $h(u)$ is the heuristic function estimating the cost from u to a goal $t \in T$. This evaluation function is used to determine the order in which the states are expanded. The algorithm starts at the initial state, adding its successor states to the open set. Next, the state with the lowest f-value is expanded and has all of its successors added to the open set; this process continues until a termination condition is met. A* using an admissible heuristic is guaranteed to return an optimal solution if one exists. The algorithm terminates once it finds an optimal solution, or if there is no solution the algorithm expands all states before terminating.

2.1.4 Pattern Databases

Pattern Databases(PDB), introduced by Culberson and Schaeffer (1996), are heuristics implemented as lookup tables that store the optimal solution cost of an abstraction space $S' = \phi(S)$ from a state s' to a closest abstraction goal state t'. An abstraction function ϕ defines the abstract space. ϕ maps the state space S into a smaller abstract state space S' in which the optimal cost between abstract states u' and v' does not exceed the optimal cost between states u and v in S. The PDB is constructed in the preprocessing phase by a reverse search from the set of abstract goal states storing the cost for each visited state in the lookup table. Later, during the search, the states are mapped to their respective abstract spaces using the function ϕ and the values stored in the lookup table are used as the heuristic function.

2.2 Sokoban

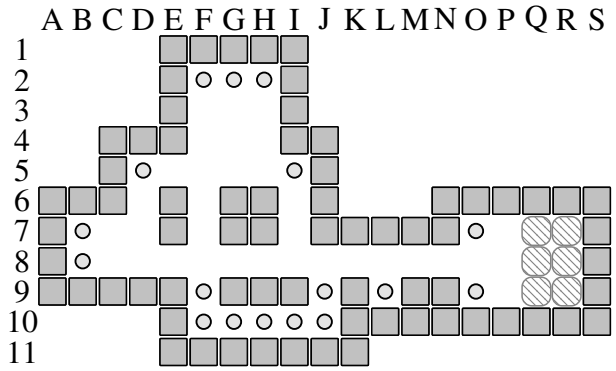
Sokoban is a PSPACE-Complete (CULBERSON, 1999) problem that is challenging for both humans and computers due to its large state space with deadlock states, long solutions, and great branching factor. Its similarities with general robot motion planning make it an interesting problem for research in artificial intelligence.

Deadlocks are extremely important in Sokoban, they are very common and greatly impact performance during the search. *Dead squares* are squares from which a stone cannot be pushed to the goal, Figure 2.2 shows the dead squares in instance #1. The subset of stones that defines a deadlock can be simple, such as a stone in a dead square, or complicated, involving multiple stones. An informed heuristic for Sokoban should be able to detect deadlocks, pruning them from the search and greatly increasing the performance.

2.2.1 Enhanced Minimum Matching

Enhanced Minimum Matching (EMM) is the standard heuristic function for Sokoban, it is consistent and was introduced in the Rolling Stone solver (JUNGHANNS; SCHAEFFER, 2001). The idea behind EMM is to match each stone to a different goal square. It is computed using three components: backout conflicts, a minimum cost perfect matching, and linear conflicts. The heuristic initially computes the minimum number of pushes

Figure 2.2: Instance #1 of xSokoban, a dead square is shown for example at F2.

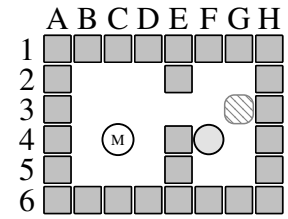


needed to move a stone from each square to every other square considering all the other stones were removed from the maze, this considers backout conflicts. A backout conflict occurs when the optimal path of a stone considers the current position of the man. An example is shown in Figure 2.3a, where the only way to push the stone to the goal is through the C3 square. Then, a minimum cost perfect matching in a bipartite graph composed of nodes for the stones in one half and nodes for the goals in the other half computes a lower bound estimating the minimum number of pushes needed to bring all stones to goal squares. The weight of each edge is the number of pushes needed to move the stone to the goal according to the previous computation. Linear conflicts add a cost of two if two adjacent stones are in the optimal path of each other, an example is shown in Figure 2.3b. Detecting deadlocks is extremely important for a Sokoban heuristic, and EMM can detect simple deadlocks, such as stones in dead squares or when a matching does not exist, for example, when there are multiple stones that can only reach a single goal square. EMM will be used as the main heuristic in this thesis, together with neural networks for deadlock detection.

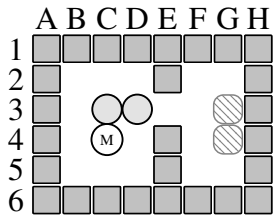
2.2.2 PDBs in Sokoban

PDBs were introduced in Sokoban by Pereira, Ritt and Buriol (2013) using intermediate PDBs (IPBD), a technique that decomposes the instance to allow multiple abstract goal states to be abstracted into one single intermediate abstract goal state. This cuts the instance into two zones, the maze zone, were the PDB is used as the heuristic to reach the intermediate goal state, and the goal zone, where a standard minimum matching

Figure 2.3: Example of backout and linear conflicts.



(a) The stone can not be pushed to the goal directly, it must first be pushed back to square C3.

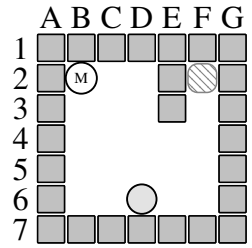


(b) The stones are in the optimal path of each other, the stone on C3 must be pushed down in order to push the stone on D3 to the goal.

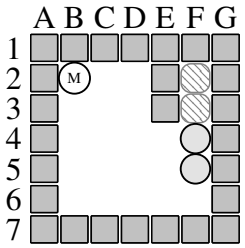
heuristic is used to reach the goal states. IPDB was proven to produce great lower bounds for Sokoban, but it is effective at detecting deadlocks only in the maze zone.

Later on Pereira, Ritt and Buriol (2014) introduced multiple goal states PDBs (MPDB), a technique that uses PDBs for deadlock detection in Sokoban. MPDBs have different sizes based on the number of stones used in the abstraction, an MPDB-4, for example, uses only four stones. This limits the MPDB capability to detect deadlocks of up to its order, an MPDB-2 can detect deadlocks of order two or lower, and an MPDB-4 can detect deadlocks of order four or lower. Figure 2.4a shows a deadlock of order one, the stone can only be moved left or right following the wall. Deadlocks of order one consist of a stone on dead squares. Figures 2.4b, 2.4c and 2.4d show deadlocks of order two, three and four respectively. Pereira, Ritt and Buriol (2014) experimented with MPDBs as a heuristic function, but it provided worse estimates than both EMM and IPDB. Later they used MPDBs only for deadlock detection. In this experiment, they ran an A* guided by EMM that used MPDBs only for pruning deadlock states. For every generated state, A* checks with an MPDB if that state is a deadlock. If it is the state is pruned. If the MPDB classify the state as alive A* inserts it into the open set. A* algorithm guided by EMM and pruning states with MPDB-4 solves two more instances from xSokoban, twelve in total, while expanding an order of magnitude fewer states. We will use an MPDB-4 to create the training set for the neural network and this configuration of A* as a comparison for our approach.

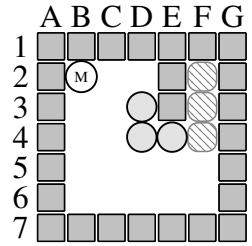
Figure 2.4: Sokoban deadlocks of different orders. The order of a deadlock is defined by how many stones are needed for the state to be a deadlock. So if any of the stones is removed the state will be alive.



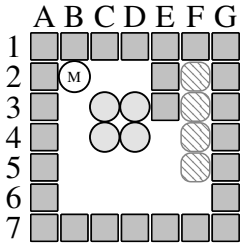
(a) Deadlock with one stone.



(b) Deadlock with two stones.



(c) Deadlock with three stones.



(d) Deadlock with four stones.

2.3 Learning Heuristic Functions

This section gives an introduction to machine learning and neural networks. See (GEISSMANN, 2015) for a more detailed explanation.

2.3.1 Machine Learning

Machine learning is a field of study in Computer Science where the algorithm learns based on a dataset to be able to make predictions or act based on new data. There are three main types of learning algorithms. Supervised learning is when the learning algorithm receives a dataset where the input values have a corresponding output value, the goal is to learn a function that maps the input data to the output values. Unsupervised learning is when the learning algorithm receives a dataset with only the input values and has to find patterns in the dataset. Reinforcement learning is when the learning algorithm takes actions in a dynamic environment to perform a specific task based only on some notion of rewards. This thesis only uses supervised learning. In supervised learning, there are two possible tasks: classification and regression. Classification maps the input

values to different class labels while regression maps the input values to output values. We will use a binary classification model to learn whether a Sokoban state is a deadlock or not.

2.3.2 Artificial Neural Networks

Artificial Neural Networks, or simply *Neural Networks* (NN) are computing systems inspired by biological neural networks such as the human brain. A NN is an interconnected net of *artificial neurons*, each neuron calculates its output based on its inputs and their weights using an *activation function*. The simplest form of NN consists of two layers of neurons: the input layer, and the output layer. The neurons in the input layer do not use the activation function, they simply provide the input values to the next layer. The neurons on the output layer calculate their outputs based on the values received from the previous layer. *Hidden layers* are extra layers added between the input and the output layer, increasing the complexity of the NN.

NNs are particularly interesting because there are algorithms to learn the weights of the neurons based on the *training set*, a set of inputs with their expected output values. These algorithms run the network with their current weights and then use the error produced to update the weights backward throughout the network's layers. The weights can be updated for each element of the training set or in *batches*. Each batch can contain the whole training set, only one element, or any number in between. An *epoch* is when each element of the training set was used once to update the weights, the *learning rate* controls how much the weights are updated each batch.

2.4 Related Work

Next, we present approaches that use algorithms to learn heuristics that estimate the cost of state space problems. They use machine learning to learn heuristics that can be used to guide an informed search algorithm.

When a problem has multiple existing heuristics, taking their maximum is an effective way of combining them, but Samadi, Felner and Schaeffer (2008) introduced a new technique where the heuristics are treated as features of the problem domain and an NN is used to combine them. The NN learns a function that combines k heuristics to get

as close as possible to the optimal solution. The experimental results showed that this technique greatly reduced the search effort with a small cost in the quality of the solution.

Arfaee, Zilles and Holte (2011) investigated the use of machine learning to create effective heuristics for search algorithms or heuristic-search planners in a single domain. Their method generated a sequence of heuristics from a given weak heuristic h_0 and a set of unsolved instances using a bootstrapping procedure. The bootstrapping procedure uses the training instances that can be solved using h_0 to create a training set for a learning algorithm that produces a heuristic h_1 that is expected to be stronger than h_0. The bootstrapping procedure is repeated until a sufficiently strong heuristic is produced. They presented experiments for the 24-Sliding-tile puzzle, the 35-Pancake puzzle, Rubik's Cube, and the 20-Blocks world, in all cases the heuristic produced was strong enough to solve random instances quickly with solutions close to optimal. The total time to create strong heuristics was in the order of days, for a more efficient way to solve a single instance they presented a variation in which the bootstrap learning of new heuristics is interleaved with problem-solving using the initial heuristic and whatever heuristics have been learned so far. This substantially reduced total time with solutions that were still close to optimal.

Lelis et al. (2012) presented an algorithm to predict the optimal solution cost for a space state problem called Bidirectional Stratified Sampling(BiSS). BiSS is based on ideas of bidirectional search and stratified sampling that produces accurate estimates of the optimal solution cost without finding the solution itself. Their method makes accurate predictions in several domains, is guaranteed to return the optimal solution cost in the limit as the sample size goes to infinity, and is much faster than actually finding the solution. Later on, Lelis et al. (2016) used BiSS to improve the bootstrapping procedure by Arfaee, Zilles and Holte (2011). Instead of using search to solve problem instances to generate a training set, they generate a set of problem instances and then use BiSS to label these instances to form the training set. They were able to reduce the time needed for the bootstrapping procedure to train the heuristic from days to minutes, while still keeping the quality of the learned heuristics roughly the same.

Based on the bootstrapping procedure approach by Arfaee, Zilles and Holte (2011), Geissmann (2015) introduced a framework to learn heuristic functions that can be used in classical domain-independent planning. Their approach reduced the number of states expanded when compared to blind search, but because the heuristic needs to be learned the total time was increased.

3 LEARNING DEADLOCKS IN SOKOBAN

Deadlocks are extremely important in some state space problems such as Sokoban, but, usually, there is no obvious way to detect them. We explore the possibility of using neural networks for deadlock detection, comparing them to the current approach, MPDBs (See Section 2.2.2). One of the main problems when using neural networks is how to generate a good training set. If the training set is too small or contains states that are too much alike, the capability of the network to learn can be severely limited. To generate a good training set, we present different methods to generate alive states, and we use MPDB to generate deadlocks states. Then, using our best method, we train a neural network for a specific Sokoban instance with the goal to emulate the behavior of an MPDB. Afterwards, we run an A* guided by EMM (See Section 2.2.1) that uses the NN only for pruning states. For every generated state A* will use the NN to detect if the state is alive or a deadlock and prune them accordingly. The following sections explain how we generated the training set, the model used for the neural network.

3.1 Training Set

We generate a training set composed of two classes of states: deadlock states and alive states. The training set contains the same number of alive states and deadlock states, and this is done to prevent the network from favoring the class that is more frequent in the training set. The next sections explain the methods used to generate alive states and to generate deadlock states. Chapter 4 shows experiments with these methods and their combinations to obtain the best approach.

3.1.1 Generating Alive States

We generate alive states using a reverse search from a goal state. Given a state space problem $P = (S, A, s, T)$ the problem for the reverse search can be defined as $P' = (S, A', t, s)$, where the initial state is a goal state $t \in T$ and the goal state is the original initial state s. A' is the set of reverse actions, that is, given an action $a \in A$ with $a(u) = v$ there exists an action $a' \in A'$ with $a'(v) = u$ and $u, v \in S'$. We use a reverse search to ensure the generated states are indeed alive. Since we only generate states that

are reachable from the goal state with the reverse set of actions A', and every action in A' can be reversed, then all generated states can reach the goal in a forward search. Thus, a reverse search generates only alive states.

Many search algorithms can be used for the reverse search, the ones we chose are Breadth-Firs Search, Greedy Best-First Search, and A*. The objective is to use these different algorithms to generate a diverse set of states.

3.1.1.1 Breadth-First Search

Breadth-First Search(BFS) is an uninformed search algorithm that expands the states in the order of their depth. In general, using BFS in the reverse search produces states that are close to the goal.

3.1.1.2 A* Algorithm

The motivation to use of a reverse A* is to produce states that are more likely to be seen on the forward search. A* starts at the goal and move in the direction of the initial state, searching for the optimal solution. A* algorithm requires a heuristic that estimates the cost from the current state to the initial state. For this, we use the Reverse Enhanced Minimum Matching heuristic. REMM is similar to EMM but for the reverse state space problem. We compute backout conflicts and linear conflicts using the reverse moves and the minimum matching from the current positions of the stones to the squares that contain stones in the initial state. REMM is the same as EMM but estimating the costs from the current state to a goal state of the reverse. A* also tends to produce states close to the goal but in the direction of the initial state.

3.1.1.3 Greedy Best-First Search

Greedy Best-First Search (GBFS) only uses the value of the heuristic to choose which state expand next. GBFS always expands the generated state with the best h-value. We have used two different heuristics for GBFS, maximizing EMM and minimizing REMM. GBFS guided by EMM, in general, generates states that are as far from the goal as possible. GBFS guided by REMM, what we call *Reverse GBFS* (RGBFS), in general, produces states that are as close to the initial state as possible.

3.1.2 Generating Deadlocks

We propose a simple method to generate deadlocks. Our algorithm places all stones of the instance at random free square. Then, an MPDB-4 is used to detect if the generated state is a deadlock, if the state is classified as a deadlock it is added to the training set. This is done until there is the same number of deadlocks states as there are alive states. Using an MPDB-4 limit the order of the generated deadlocks to four.

We also force deadlocks of each order, that is, we try to generate the same number of deadlocks for each order up to four for a time limit. If the time limit is reached and there are not enough deadlocks, we generate them freely until we complete the training set. To find out the order of a deadlock we generate four MPDBs, one for each order, the first one to detect a state as a deadlock represents its order, for example, a state is a deadlock of order three if MPDB-1 and MPDB-2 classify it as alive but MPDB-3 classify it as a deadlock. Deadlocks with order higher than four are not included in our training set.

3.2 Neural Network Model

We use a fully connected neural network with hidden layers. The input is a *flat state*: an array of zeros and ones that has twice the size of the number of free squares, the first part contains the positions of the stones and the second part the position of the man. The output is the two classes: alive and deadlock. To decrease the number of false positives in the search we only consider the state a deadlock if the network predicts it with 99% certainty. Mistakenly pruning an alive state from the search could render the problem unsolvable or the solution sub-optimal, while ignoring all deadlocks degrades the heuristic back to the original one, in our case EMM.

4 EXPERIMENTS

This chapter presents the results of the experiments. First, we choose the parameters for the network. Then, we use the network to compare the different algorithms to generate the training set. In the end, we compare our best method with MPDBs when detecting deadlocks for random states and for pruning states during an informed search with A* guided by EMM. The instances tested were the ones solved when using an MPDB-4 to detect deadlocks in one hour, instance #1 was kept out for being too small.

During this chapter we reference a neural network as *NN(*$A_1, A_2, \ldots, A_n$*)*. This notation means that the alive states in the training set were generated by algorithms $A_1, A_2, \ldots, A_n$. The number of alive states generated by each algorithm is always the same. In tables below, when we present experiments for NN($A_1, A_2, \ldots, A_n$) we are running an A* guided by EMM which uses NN($A_1, A_2, \ldots, A_n$) only for deadlock detection.

The framework used for the neural network implementation was TensorFlow (ABADI et al., 2015), it is a widely used open source framework for machine learning that was developed by Google. Tensorflow supports the use of GPUs for optimization, and it has APIs for multiple programming languages, the main one being Python. In our case, the network was modeled with the Python API but trained and executed with the C++ API using GPU. The API version is 1.10.1. Experiments run on a computer running Ubuntu with an Intel Core i7 6700k CPU, 16 GB of RAM, and an NVIDIA GTX 1070 GPU. We limit the number of expanding nodes to 5 million and the time to one hour.

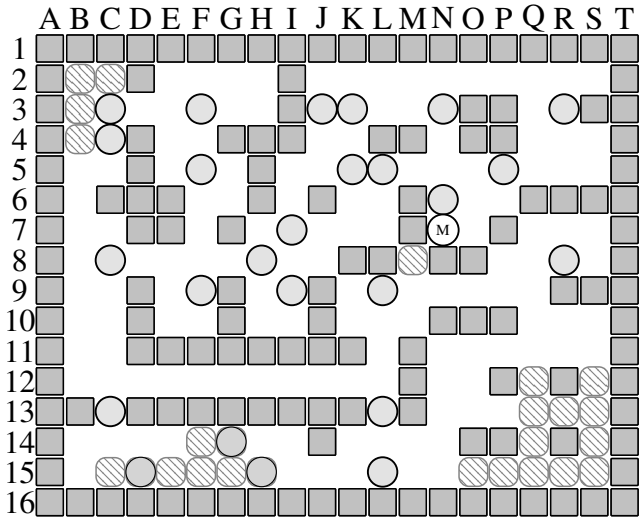
4.1 Instances

The instances used throughout this thesis are taken from the benchmark xSokoban (MYERS, 1995). The 90 instances have varying levels of difficulty, from easier ones (Figure 2.1) to harder ones (Figure 4.1), and are ordered roughly in difficulty for humans to solve.

4.2 Neural network parameters

There are many parameters that can be adjusted in a neural network: how the input and output are mapped, the number of hidden layers, the number of nodes in each layer,

Figure 4.1: Instance #90 of xSokoban. D15, G14 and H15 are goal squares.



the activation functions, the weights initial values, the bias initial values, the learning rate, the loss function, the optimizer, number of epochs, the batch size, and the size of the training set. Since we want a total time of less than one hour, we limit some parameters to have a training time of fewer than 30 minutes. This section does not present all experiments made to arrive in the final values, but, as an example, Table 4.1 compares different numbers of hidden layers, other parameters are at their final values. The column *Nodes* shows the number of expanded states, instances that were not solved within the time limits are marked with the $>$ symbol. The column *Time* shows the time taken for the search in seconds, excluding training time. We can see that more hidden layers provide the best results, expanding fewer states in less time. Instance #51 is the best example, expanding less than half the number of states for each additional hidden layer.

The NN used in the rest of the thesis has four hidden layers, 1024 nodes in each hidden layer, a flat state as the input and two classes as the output. The activation function used in the hidden layers is rectifier(ReLU), and the activation function for the output layer is softmax. The weights are initialized using the truncated normal function with a standard deviation of 0.1 and the bias values are initialized at 0.01. The Adam optimizer minimizing the cross-entropy loss is used to update the network weights and learning rate, that starts at 0.00001. The network trains for 10 epochs with batches of 2048 elements and a training set consisting of 10 million unique alive states and the same number of unique deadlock states, if the instance does not contain 10 million unique alive states then all alive states are used in the training set. The time to generate the training set and to

Table 4.1: Comparing different number of hidden layers for NN(BFS,GBFS,RGBFS).

#	2 Hidden layers		3 Hidden layers		4 Hidden layers	
	States	Time	States	Time	States	Time
2	105,414	251	84,356	226	109,204	287
3	348,180	382	135,056	205	154,499	232
6	559,775	600	384,721	496	296,311	436
7	>4,115,756	3600	>3,910,386	3600	>3,645,610	3600
17	645,261	370	344,020	237	482,362	343
38	31,979	41	22,843	33	17,995	28
49	1,580,374	1795	1,554,011	1909	1,436,325	1847
51	>1,404,784	3600	842,433	2508	306,711	902
78	392	1.36	20,863	63	27,076	91
80	27,705	28	27,694	30	27,694	32
81	>1,403,714	3600	>1,075,350	3600	>945,112	3600
83	7,624	15	2,634	6.25	13,239	32
Mean	852,580	1191	700.364	1077	621.845	952
Solved	7		8		8	

training the NN takes at most 27 minutes.

4.3 Training Set

With the NN model fixed we test which method for generating the training set produces the best results. We start by comparing the different algorithms to generate alive states and then test if deadlocks of higher order produce a better training set.

4.3.1 Generating Alive States

We proposed four different methods to generate alive states for the training set: BFS, A*, GBFS and RGBFS. We compare them running A* guided by EMM using the NN only for deadlock detection. Since BFS and A* produce similar states that are close to the goal they are not combined. All other possible combinations are shown in Tables 4.2, 4.3 and 4.4. Due to the high number of tests a lower limit of 10 minutes was used in the search. The tables show the number of expanded states and the time in seconds for the search, excluding the training, if no solution was found in the given limits the symbol $>$ is used, if the solution found was not optimal the symbol + is used. All methods, except for NN(BFS), found the optimal solution in all solved instances. The time to generate the training set is at most 12 minutes, with BFS being the fastest and A* the slowest.

In table 4.2 we see that NN(BFS) performed the worst, solving only two instances

and being the only one to find solutions that are not optimal. For instance #38 NN(BFS) found a solution with cost 83, instead of the optimal 81, and for instance #17 it found a solution with cost 217, instead of the optimal 213. NN(A*) performed better than NN(BFS), solving five instances, but not as good as NN(GBFS) and NN(RGBFS) which solved eight instances each. NN(RGBFS) solved the same number of instances than NN(GBFS), but in less time and expanding fewer states.

Table 4.3 show combinations of two algorithms, NN(BFS,GBFS) solves seven instances and NN(BFS,GBFS), NN(A*,GBFS) and NN(A*,RGBFS) solves eight instances. Table 4.4 show combinations of three algorithms, NN(BFS,GBFS,RGBFS) solves seven instances and NN(A*,GBFS,RGBFS) solves eight instances. Comparing the results from the three tables we see that many solved the same number of instances, NN(RGBFS) expanded the lowest number of states and NN(A*,GBFS,RGBFS) was the fastest overall.

Table 4.2: Comparing methods to generate alive states with one algorithm.

#	NN(BFS)		NN(A*)		NN(GBFS)		NN(RGBFS)	
	States	Time	States	Time	States	Time	States	Time
2	>830,212	600	>582,198	600	39,795	112	77,025	210
3	>927,984	600	>331,403	600	127,477	224	129,664	210
6	>927,369	600	344,225	495	111,292	192	164,860	274
7	>808,279	600	>643,042	600	203,816	365	>354,466	600
17	+446,954	273	434,742	342	454,164	324	426,374	329
38	+50,495	57	19,251	33	20,118	32	16,831	28
49	>719,266	600	>380,325	600	>691,842	600	>349,425	600
51	>493,723	600	>158,636	600	>203,509	600	>196,116	600
78	>796,995	600	>263,115	600	852	3	1,217	4
80	>650,001	600	9,820	20	27,743	39	27,773	35
81	>644,689	600	>135,817	600	>149,943	600	>184,176	600
83	>884,527	600	448	2	>294,267	600	940	3
Mean	681,708	527	275,252	424	193,735	308	160,739	291
Solved	2		5		8		8	

To better compare the best methods we ran another experiment using a higher time limit of one hour only with harder instances. We chose instances #7, #17 and #49 and compared methods that solved eight instances previously, except for GBFS since it was unable to solve instance #83, which should be easy and is solved by EMM with 559 expanded states. Table 4.5 show these results. NN(A*,GBFS), NN(A*,RGBFS) and NN(BFS,RGBFS) were only able to solve two instances, while NN(A*,GBFS,RGBFS) and NN(RGBFS) solved all three instances. We confirmed the results from the previous tests that NN(RGBFS) is the best method, solving the instances faster and with fewer states expanded. It is interesting that using a single algorithm to generate the alive states performed better than more complex combinations.

Table 4.3: Comparing methods to generate alive states with two algorithms.

#	NN(BFS,GBFS)		NN(BFS,RGBFS)		NN(A*,GBFS)		NN(A*,RGBFS)	
	States	Time	States	Time	States	Time	States	Time
2	31,013	98	71,653	195	41,089	116	94,309	251
3	381,877	501	152,898	219	169,879	290	169,254	263
6	109,931	188	131,366	213	220,300	361	344,343	483
7	>533,160	600	>547,245	600	>511,343	600	>532,611	600
17	425,460	314	462,511	329	520,034	359	519,130	358
38	17,310	29	22,317	34	19,017	29	15,693	25
49	>731,608	600	>416,607	600	>452,950	600	>430,914	600
51	>202,766	600	>165,053	600	>149,768	600	>156,335	600
78	1,141	3	7,586	12	2,007	3	8,329	14
80	177,691	600	43,384	69	23,976	29	27,639	31
81	>177,005	600	>189,202	600	>185,289	600	>173,575	600
83	>306,227	600	3,388	9	921	2	23,541	66
Mean	257,932	394	184,434	290	191,381	299	207,973	342
Solved	7		8		8		8	

4.3.2 Generating Deadlocks

We also tested how the NN would perform if given a training set contains a higher number of more complex deadlocks. To generate this training set we forced the same number of deadlocks of each order for 30 minutes. We trained the network and ran A* guided by EMM using the NN only for deadlock detection.

Table 4.6 compares how many deadlocks of each order were generated when forcing for 1 minute and for 30 minutes. Forcing for 1 minute has a small impact in most cases, on average 91% of the generated deadlocks have order one, 8% of order two, 0,7% of order three and 0,3% of order four, while forcing for 30 minutes has only 58% of order one, almost 25% of order two, 12% of order three and 5% of order four. The only case the algorithm finished in less than 30 minutes is instance #1, for all other instances 30 minutes was not enough to produce the same number of deadlocks for higher orders. Overall we see that forcing for a longer time resulted in more complex deadlocks in the training set.

Table 4.7 show how a training set with more complex deadlocks affects the performance in the informed search. We can see that the number of states expanded and the time for the search decreased, but the total time increased significantly. Using a limit of one hour for the total time when forcing for 30 minutes the NN would not be able to solve instances #7, #49, #51 and #80. Even though the performance improved when forcing deadlocks for a long time it is not enough to compensate the extra preprocessing time.

Table 4.4: Comparing methods to generate alive states with three algorithms.

#	NN(BFS,GBFS,RGBFS)		NN(A*,GBFS,RGBFS)	
	States	Time	States	Time
2	77,234	209	100,104	274
3	>447,153	600	50,090	85
6	150,654	251	>429,895	600
7	>481,793	600	>492,071	600
17	443,646	322	427,258	344
38	17,578	29	18146	28
49	>425,730	600	>433,500	600
51	>165,552	600	19,490	97
78	3,516	12	3,384	10
80	28,701	34	27,702	31
81	>200,165	600	>183,162	600
83	50,731	56	2,567	7
Mean	207,704	326	182,281	273
Solved	7		8	

Table 4.5: Comparing best methods to generate alive states for harder instances.

#	NN(A*,GBFS,RGBFS)		NN(A*,GBFS)		NN(A*,RGBFS)		NN(BFS,RGBFS)		NN(RGBFS)	
	States	Time	States	Time	States	Time	States	Time	States	Time
7	1,505,647	2029	>3,493,845	3600	>3,186,357	3600	1,268,034	1263	600,757	836
49	1,027,959	1497	759,668	1053	943,056	1346	1,024,895	1336	566,021	1020
51	35,649	170	182,718	672	227,649	940	>1,083,221	3600	187,982	602
M.	856,418	1232	1,478,744	1775	1,552,354	1762	1,125,383	2066	451,586	819
S.	3		2		2		2		3	

4.4 Comparing to MPDBs

In our final experiments, we use NN(RGBFS) forcing deadlocks for one minute. NN(RGBFS) was the best method when generating alive states and forcing deadlocks for one minute has almost no impact in the total time while still having a small improvement in the deadlock states for the training set.

Table 4.8 compares the deadlock detection capabilities of EMM, MPDB-2, MPDB-4 and NN(RGBFS). We used the same 10,000 randomly generated states from Pereira, Ritt and Buriol (2014) adding a column for the NN, each column shows how many of the 10,000 states were classified as deadlocks by that method. EMM on average detects only 1,318, MPDB-2 does a lot better with 8,168 deadlocks and MPDB-4 improves a bit more with 8,981. The network considers even more states as deadlocks, with an average of 9,988, but they are not necessarily correct.

While the other methods never classify an alive state as a deadlock, the network does. We try to prevent this by only considering a deadlock when the network is almost

Table 4.6: Deadlocks generated when forcing deadlocks.

#	Order of deadlocks when forcing one minute				Order of deadlocks when forcing 30 minutes			
	1	2	3	4	1	2	3	4
1	2,010,974	1,734,525	209,942	66,504	1,005,487	1,005,487	1,005,487	1,005,484
2	9,436,515	4,756,82	73,643	14,162	5,513,207	2,520,172	1,654,844	311,779
3	9,341,456	578,223	61,347	18,976	5,886,983	2,598,386	1,150,933	363,700
6	9,683,306	255,064	42,632	18,999	5,374,751	2,519,547	1,451,474	654229
7	9,572,769	306,949	71,283	49,001	4,849,442	2,510,475	1,566,699	1,073,386
17	7,969,700	1,965,710	48,176	16,416	5,804,694	2,566,137	1,216,955	412,216
38	7,603,961	2,161,236	167,467	67,338	3,503,361	2,662,488	2,512,578	1,321,575
49	9,366,810	568,956	53,944	10,292	5,777,451	2,517,751	1,434,757	270,043
51	9,837,068	134,653	19,416	8,865	6,834,054	2,532,870	435,438	197,640
78	9,356,747	563,918	64,351	14,986	5,941,759	2,643,573	1,151,637	263,033
80	9,915,158	76,951	3,870	4,022	8,310,465	1,527,451	82,612	79,474
83	9,757,914	221,644	16,665	3,779	6,902,231	2,527,605	466,671	103,495
Mean	8,654,364	753,626	69,395	24,445	5,475,323	2,344,328	1,177,507	504,671

certain, but it can still make mistakes. It is also possible that the NN is correct and all generated states are deadlocks. For example, in instance #48 MPDB-4 detects 9,964 deadlocks, the remaining 36 states could be deadlocks of a higher order than four that NN(RGBFS) classified correctly.

Table 4.9 shows an A* guided by EMM using MPDB-4 for deadlock detection. For each state generated we use both MPDB-4 and NN(RGBFS) to detect if the state is alive or deadlock, comparing their answers. The first two columns show the number of states that were detected as alive by MPDB-4. Column "NN(RGBFS) Alive", shows the number of states that both MPDB-4 and NN(RGBFS) detected as alive while column "NN(RGBFS) Deadlock" shows states that MPDB-4 detected as alive but NN(RGBFS) detected as deadlocks. The states in the second column can either be a deadlock with order higher than four that MPDB-4 is unable to detect or alive states that were mistakenly detected as deadlocks by the NN. We can see that in average NN(RGBFS) agrees with MPDB-4 in only 44% of the alive states, detecting the majority of the MPDB-4 alive states as deadlocks. The last two columns are states that MPDB-4 detected as deadlocks, these states are guaranteed to be deadlocks. States in the third column are deadlocks that NN(RGBFS) missed, detecting them as alive, while states in the fourth column are states the NN(RGBFS) correctly detected as deadlocks. We see that NN(RGBFS) correctly detects 73% of the deadlocks detected by MPDB-4, missing 27%.

Our final experiment compares the performance of an A* guided by EMM using MPDB-2, MPDB-4 and NN(RGBFS) only for deadlock detection. In this experiment, we use the complete set of instances of xSokoban and only instances solved by at least

Table 4.7: Forcing deadlocks for NN(RGBFS).

#	Forcing 1 minute			Forcing 30 minutes		
	States	Time		States	Time	
		Search	Total		Search	Total
1	158	0	764	159	1	2003
2	50,284	132	1997	52,396	136	3580
3	165,503	232	1833	79,654	119	3568
6	97,599	157	1771	41,315	63	3477
7	486,226	701	2510	246,357	368	3814
17	134,745	124	1974	50,993	56	3500
38	18,854	31	1763	15,933	27	3318
49	299,967	487	2470	539,433	1030	4619
51	231,144	772	2493	124,903	404	3997
78	4,436	13	2029	3,846	11	3573
80	27,878	35	2054	31,252	41	3808
83	2,272	8	1653	1,352	4	3636
Mean	126,589	225	1943	98,966	188	3574
Solved		12			12	

one of the methods are shown. We limit the number of expanded states to five million and the time to one hour. Results are shown in table 4.10, for NN(RGBFS) there are two columns for the time, one for the total time and one for the time taken only in the search. For the other methods, we only show the total time. The neural network improves on EMM significantly, EMM solved 10 instances and the NN solved 12, expanding one order of magnitude fewer states for harder instances while maintaining a similar performance for easier instances. It also performs better than MPDB-2, which solved 11 instances expanding more states overall. MPDB-4 was still the best, solving 13 instances. NN improved on MPDB-4 on instances #2, #49 and #78, but was unable to solve instance #81 and expanded more states on the other instances. The total time for the NN is higher due to the preprocessing time to train the network.

Table 4.8: Comparing deadlock detection over 10,000 randomly generated states. The states and values for EMM, MPDB-2 and MPDB-4 are from Pereira, Ritt and Buriol (2014), we generated the values for NN(RGBFS).

#	EMM	MPDB-2	MPDB-4	NN(RGBFS)	#	EMM	MPDB-2	MPDB-4	NN(RBGFS)
1	76	5,669	6,434	9,812	46	2,063	7,754	8,752	9,999
2	142	6,545	7,764	9,998	47	0	8,365	9,119	10,000
3	9	7,153	8,173	9,995	48	0	9,808	9,964	10,000
4	2,463	8,612	9,194	10,000	49	3,848	9,079	9,707	9,998
5	4	7,619	8,456	10,000	50	8,268	9,616	9,768	9,672
6	2,491	6,897	7,928	9,985	51	0	6,792	8,296	9,997
7	0	5,779	8,079	9,969	52	62	9,479	9,765	10,000
8	37	7,056	8,377	10,000	53	1	7,485	8,278	10,000
9	3,935	8,685	9,106	10,000	54	0	8,147	8,900	10,000
10	1,315	8,218	9,405	10,000	55	8,072	9,087	9,520	10,000
11	0	7,959	8,792	10,000	56	7,922	9,626	9,819	10,000
12	205	7,199	8,380	10,000	57	0	5,643	6,845	10,000
13	0	7,564	8,887	10,000	58	2,853	8,070	8,683	10,000
14	3,518	9,575	9,867	10,000	59	1,979	7,758	8,890	10,000
15	0	7,985	8,922	10,000	60	5,180	8,274	8,917	10,000
16	2,623	8,506	9,642	10,000	61	4,218	9,044	9,591	10,000
17	3,890	8,254	8,387	9,937	62	8,721	9,763	9,877	10,000
18	4,930	7,503	8,757	10,000	63	13	5,820	7,152	10,000
19	3,245	9,020	9,467	10,000	64	1,671	8,327	8,861	10,000
20	0	8,380	9,044	10,000	65	0	6,479	8,311	10,000
21	0	8,196	8,923	9,998	66	2,003	7,170	8,400	10,000
22	0	8,826	9,569	10,000	67	2,554	8,168	9,219	10,000
23	0	9,262	9,658	10,000	68	35	6,986	8,736	9,681
24	3,460	9,613	9,875	10,000	69	29	7,435	8,700	10,000
25	4,004	9,290	9,775	10,000	70	2,271	9,006	9,585	10,000
26	444	8,426	9,303	9,996	71	2,370	8,706	9,601	10,000
27	64	8,869	9,587	10,000	72	1,100	8,326	9,405	10,000
28	2,687	9,215	9,809	10,000	73	2,958	7,376	8,313	10,000
29	0	9,524	9,952	9,999	74	2,116	8,341	9,441	10,000
30	0	9,651	9,927	10,000	75	28	8,324	9,487	10,000
31	0	8,802	9,645	10,000	76	1,906	9,059	9,651	9,999
32	2,526	8,457	9,359	10,000	77	1,701	9,147	9,810	10,000
33	0	8,439	9,508	10,000	78	195	5,134	5,863	10,000
34	14	7,140	8,774	10,000	79	153	4,005	4,935	10,000
35	0	7,639	8,641	10,000	80	1,939	5,656	6,080	9,998
36	28	8,968	9,724	10,000	81	130	6,402	7,320	10,000
37	39	8,252	9,186	10,000	82	17	7,016	7,854	9,999
38	0	7,942	8,775	9,946	83	426	6,227	6,828	10,000
39	3,625	9,498	9,896	10,000	84	1	5,597	6,122	9,999
40	9	9,185	9,543	10,000	85	5,701	9,040	9,749	10,000
41	0	8,826	9,740	10,000	86	68	7,315	8,338	9,989
42	2,465	9,066	9,638	10,000	87	3	8,223	8,907	10,000
43	0	6,874	8,030	10,000	88	0	9,413	9,775	10,000
44	3,030	7,595	8,414	9,996	89	2,453	9,529	9,876	10,000
45	0	8,167	9,044	10,000	90	959	9,665	9,952	10,000
					Mean	1,318	8,168	8,981	9,988

Table 4.9: Comparing NN(RGBFS) and MPDB-4 during search.

#	MPDB-4 Alive		MPDB-4 Deadlock	
	NN(RGBFS) Alive	NN(RGBFS) Deadlock	NN(RGBFS) Alive	NN(RGBFS) Deadlock
1	637	0	94	30
2	191,384	108,762	74,464	105,574
3	22,019	25,912	7,561	8,582
6	2,939	2,021	2,356	2,793
7	85,807	226,330	22,565	79,238
17	1,615	0	1,196	674
38	21,005	1,371	6,155	11,527
49	1,607,843	898,527	235,680	370,884
51	5,525	956,825	2,695	349,142
78	6,945	22,851	70	15,657
80	1,888	3,072	424	223
81	315,224	666,097	226,067	597,768
83	594	1,028	269	206
Mean	174,110	224,061	44,584	118,638

Table 4.10: Comparing Neural networks with MPDBs.

#	EMM		EMM+MPDB-2		EMM+MPDB-4		EMM+NN(RGBFS)		
	States	Time	States	Time	States	Time	States	Time	
								Search	Total
1	160	0	160	0	**153**	0	158	0	764
2	161,835	5	86,841	18	68,928	52	**50,284**	132	1997
3	1,187,486	21	950950	76	**22,268**	10	165,503	232	1833
6	1,354,432	25	366,955	26	**1,641**	1	97,599	157	1771
7	>5,000,000	93	223,957	24	**127,841**	64	486,226	701	2510
17	1,471,533	16	19,633	0	**775**	1	134,745	124	1974
38	93,423	1	24,196	1	**8,919**	1	18,854	31	1763
49	1,596,896	39	1,381,596	226	964,810	922	**299,967**	487	2470
51	>5,000,000	210	>5,000,000	1279	**223,106**	709	231,144	772	2493
78	8,544	0	8,387	1	7,646	9	**4,436**	13	2029
80	27,708	1.08	10,594	2	**480**	27	27,878	35	2054
81	>5,000,000	266	>5,000,000	1236	**198,935**	534	>1,169,441	3600	5772
83	559	0	362	0	**305**	9	2,272	8	1653
Mean	1,607,890	52	1,005,664	222	125,062	180	206,808	484	2238
Solved	10		11		13		12		

5 CONCLUSION

We presented an approach based on machine learning for deadlock detection in Sokoban. We proposed methods to generate training sets of alive and deadlock states which were used to train an artificial neural network. We applied the resulting network to prune deadlocks with an informed search algorithm. Even though training the network introduced a substantial preprocessing time, our best network solved more instances than two of the previous methods and the solution found was optimal for all solved instances.

We have shown that neural networks are an effective alternative for the task of deadlock detection. When comparing our results with EMM and MPDB-2 we significantly decreased the number of expanded states during the search and solved more instances. Our performance was close to an MPDB-4 which solved one more instance.

However, there are open problems: training requires a high preprocessing time and finding the best model and parameters is challenging. We used a simple multilayer perceptron, and different models or parameters could provide better results, both in training time and in its deadlock detection capabilities. Another possible improvement would be to use a model such as Convolution Neural Networks that is independent of the instance, amortizing the preprocessing time among all instances.

REFERENCES

ABADI, M. et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org, Accessed: 2018-11-17. Available from Internet: <https://www.tensorflow.org/>.

ARFAEE, S. J.; ZILLES, S.; HOLTE, R. C. Learning heuristic functions for large state spaces. **Artificial Intelligence**, v. 175, n. 16-17, p. 2075–2098, 2011.

CULBERSON, J. Sokoban is PSPACE-complete. In: **International Conference on Fun with Algorithms**. [S.l.: s.n.], 1999. p. 65–76.

CULBERSON, J. C.; SCHAEFFER, J. Searching with Pattern Databases. In: **Canadian Artificial Intelligence Conference**. [S.l.: s.n.], 1996. p. 402–416.

EDELKAMP, S.; SCHROEDL, S. **Heuristic Search: Theory and Applications**. [S.l.]: Elsevier, 2011.

GEISSMANN, C. **Learning Heuristic Functions in Classical Planning**. [S.l.], 2015.

JUNGHANNS, A.; SCHAEFFER, J. Sokoban: Enhancing General Single-Agent Search Methods using Domain Knowledge. **Artificial Intelligence**, Elsevier, v. 129, n. 1-2, p. 219–251, 2001.

LELIS, L. et al. Predicting Optimal Solution Costs with Bidirectional Stratified Sampling in Regular Search Spaces. **Artificial Intelligence**, v. 230, p. 51–73, 2016.

LELIS, L. et al. Predicting Optimal Solution Cost with Bidirectional Stratified Sampling. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2012. p. 155–163.

MYERS, A. **xSokoban**. 1995. Accessed: 2018-11-17. Available from Internet: <http://www.cs.cornell.edu/andru/xsokoban.html>.

PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Solving Sokoban Optimally using Pattern Databases for Deadlock Detection. **Encontro Nacional de Inteligência Artificial**, 2014.

PEREIRA, A. G.; RITT, M. R. P.; BURIOL, L. S. Finding Optimal Solutions to Sokoban using Instance Dependent Pattern Databases. In: **Symposium on Combinatorial Search**. [S.l.: s.n.], 2013.

SAMADI, M.; FELNER, A.; SCHAEFFER, J. Learning from Multiple Heuristics. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2008. p. 357–362.

APPENDIX A — NN(RGBFS) ON ALL INSTANCES

Note that while NN(RGBFS) was able to solve instance #43 this was an isolated case, we ran the test multiple times and it was never solved again.

Table A.1: Results for all instances using NN(RGBFS).

#	States	Time		#	States	Time	
		Search	Total			Search	Total
1	158	0	764	46	>1,011,739	3600	5654
2	50,284	132	1997	47	>3,530,893	3600	5808
3	165,503	232	2102	48	>653,362	3600	4994
4	>1,105,355	3600	5669	49	299,967	487	2470
5	>1,055,263	3600	5617	50	>2,010,329	3600	3752
6	97,599	157	1771	51	231,144	772	2493
7	486,226	701	2510	52	>1,741,871	3600	5742
8	>742,366	3600	5744	53	>1,059,168	3600	5685
9	>1,258,439	3600	5669	54	>1,334,225	3600	5681
10	>859,355	3600	5996	55	>968,036	3600	5678
11	>1,598,098	3600	5729	56	>602,548	3600	5668
12	0	0	0	57	>1,081,152	3600	5722
13	>989,835	3600	5682	58	>1,062,735	3600	5732
14	>1,098,070	3600	5685	59	>1,318,568	3600	5686
15	>1,931,383	3600	5721	60	>1,196,413	3600	5689
16	>2,269,023	3600	5638	61	>2,187,609	3600	5675
17	134,745	124	1974	62	>773,664	3600	5677
18	>1,109,961	3600	5599	63	>1,066,815	3600	5801
19	>1,810,420	3600	5709	64	>1,082,640	3600	5690
20	>973,786	3600	5764	65	>2,344,266	3600	5690
21	>2,730,205	3600	5594	66	>3,414,937	3600	5681
22	>2,021,888	3600	5441	67	>1,442,566	3600	5681
23	>2,045,786	3600	5755	68	>1,869,573	3600	5692
24	>1,355,357	3600	5934	69	>2,064,173	3600	5651
25	>1,860,117	3600	5713	70	>960,345	3600	5665
26	>2,920,417	3600	5574	71	>1,365,980	3600	5658
27	>3,081,117	3600	4988	72	>3,591,538	3600	5691
28	>3,598,128	3600	5052	73	>1,077,835	3600	5712
29	>3,446,820	3600	5652	74	>3,099,144	3600	5579
30	>3,403,857	3600	5647	75	>2,823,173	3600	5548
31	>3,071,542	3600	5176	76	>806,526	3600	5671
32	>1,073,985	3600	5650	77	>3,858,312	3600	5676
33	>1,361,511	3600	5622	78	4,436	13	2029
34	>2,223,820	3600	5642	79	>909,647	3600	5747
35	>2,840,992	3600	5799	80	27,878	35	2054
36	>3,590,473	3600	4954	81	>1169,441	3600	5772
37	>1,412,976	3600	5728	82	>1947,255	3600	5413
38	18,854	31	1763	83	2,272	8	1653
39	>1,157,307	3600	5824	84	>1,457,982	3600	5583
40	>1,371,116	3600	5674	85	>1,092,317	3600	5659
41	>2,043,810	3600	5642	86	>1,917,479	3600	5405
42	>590,108	3600	5815	87	>3,381,787	3600	5666
43	2,979,748	3270	5046	88	>1,797,888	3600	5403
44	>4,747,423	3600	5404	89	>1,083,628	3600	5915
45	>2,030,897	3600	5668	90	>1,787,636	3600	6100
				Mean	1,591,389	3106	5060