FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Application of Artificial Neural Networks in Solving the (N^2-1)-Puzzle
Student:	Vojtěch Cahlík
Supervisor:	doc. RNDr. Pavel Surynek, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2019/20

Instructions

Many algorithms exist that solve the (N^2-1)-puzzle. These algorithms guarantee either optimal or sub-optimal solutions. Examples of important optimal search-based algorithms include A*, CBS and ICTS. There is usually a trade-off between the speed of algorithms and the quality of solutions. Relaxations towards near-optimal solutions usually lead to improved speed.

The goal of the thesis is to investigate the potential of Artificial Neural Networks (ANNs) for finding interesting trade-offs between the speed and optimality in solving (N^2-1)-puzzle by search-based solvers. The following steps are expected to be done:

(1) analyze potential applications of ANNs in solving the (N^2-1)-puzzle in a near-optimal manner with selected search-based methods,
(2) integrate ANNs into selected methods and implement a prototype,
(3) evaluate suggested method theoretically and experimentally on a relevant set of benchmarks.

References

[1] Pavel Surynek, Petr Michalík. The Joint Movement of Pebbles in Solving the (N2-1)-Puzzle Suboptimally and its Applications in Rule-Based Cooperative Path-Finding. Autonomous Agents and Multi-Agent Systems (JAAMAS), Volume 31, Issue 3, pp. 715-763, IFAAMAS/Springer, 2017, ISSN: 1387-2532.

[2] Ian Parberry: A Real-Time Algorithm for the (n^2-1)-Puzzle. Inf. Process. Lett. 56(1): 23-28 (1995)

[3] Guni Sharon, Roni Stern, Meir Goldenberg, Ariel Felner: The increasing cost tree search for optimal multi-agent pathfinding. Artif. Intell. 195: 470-495 (2013)

[4] Guni Sharon, Roni Stern, Ariel Felner, Nathan R. Sturtevant: Conflict-based search for optimal multi-agent pathfinding. Artif. Intell. 219: 40-66 (2015)

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 8, 2019

Acknowledgements

I would like to thank my supervisor, Mr. Surynek, for being very patient with me and for motivating me to do my best. Without his wisdom and experience, writing this thesis would have been impossible. I would also like to thank my family for their support and love.

Declaration

Citation of this thesis

Cahlík, Vojtěch. *Application of Artificial Neural Networks in Solving the (*$N^2\!-\!1$*)-Puzzle*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Práce se zaměřuje na využití umělých neuronových sítí při hledání řešení hlavolamu (N^2-1), která jsou blízká řešením optimálním. V první části práce je provedena analýza možností využití umělých neuronových sítí při řešení hlavolamu, a je zjištěno, že nejefektivnější je použít umělou neuronovou síť jako heuristiku pro algoritmy prohledávání stavového prostoru. Později se práce zaměřuje na natrénování několika heuristik založených na hlubokých umělých neuronových sítích, jejichž výkonnost je následně experimentálně změřena. Při využití heuristik spolu s algoritmem A* jsou nalezená řešení nejčastěji optimální, a počet expandovaných stavů je výrazně nižší než při použití srovnatelných přípustných i nepřípustných heuristik.

Klíčová slova umělé neuronové sítě, hlavolam (N^2-1), heuristické prohledávání, řešení blízká optimálním, hluboké učení

Abstract

The thesis focuses on the usage of artificial neural networks in near-optimal solving of the (N^2-1)-puzzle. In the first part of the thesis, possible applications of artificial neural networks in solving the puzzle are analyzed, and it is found that the most effective way is to use them as a heuristic for state-space search algorithms. Later in the thesis, several heuristics based on deep artificial neural networks are trained, and they are evaluated on a set of benchmarks. When used together with the A* algorithm, the solutions obtained with the new heuristics are optimal most of the time, while the number of expanded nodes is significantly lower than that in comparable admissible and non-admissible heuristics.

Keywords artificial neural networks, (N^2-1)-puzzle, heuristic search, near-optimal solving, deep learning

Contents

List of Figures

List of Tables

Introduction

Artificial neural networks (ANNs) are a relatively old domain of research, but have gained most of their popularity only recently. ANNs are very powerful computing systems which are able to learn how to perform tasks by being given examples, without the need to be explicitly programmed. ANNs are commonly used to classify images, perform speech recognition, recommend items to users, and play games. At the time of writing, new applications of ANNs emerge every day. It is interesting to analyze the potential for applications of ANNs in the (N^2-1)-puzzle, a field in which ANNs, and especially deep learning, has had only little attention.

The (N^2-1)-puzzle is a sliding tile puzzle in which the goal is to rearrange N^2-1 pebbles on a square board of size $N \times N$ into a goal configuration, where the numbered pebbles are ordered from 1 to N−1. The pebbles are rearranged using a single empty position called the blank: a pebble can move by sliding horizontally or vertically into the adjacent blank position. There are many versions of this game based on choosing different N, of which the most famous is the 15-puzzle with a 4×4 board.

A solution is a sequence of moves which leads to the goal configuration. Many different solutions of various lengths exist to every solvable initial configuration of the puzzle. A solution is called optimal if no other solution, which is composed of a smaller number of moves, exists. It is fairly easy to solve the (N^2-1)-puzzle suboptimally: this is usually approached by systematically moving more and more pebbles into their goal positions, which gradually lowers the complexity of the remaining problem. However, finding optimal solutions is a computationally challenging problem, as it is NP-hard [1]. Finding optimal solutions is so complicated that up to now, only a handful of optimal solutions of random 24-puzzle instances have been found [2], and optimally solving a random instance of the 35-puzzle remains an open problem.

Problem domains such as the (N^2-1)-puzzle are often called “toy domains”. This is somewhat unfair, as the research of these problems is far from useless:

it has contributed various ideas back into other areas of computer science and artificial intelligence. Moreover, these puzzles serve as important benchmarks for various types of heuristics and heuristic search algorithms [2]. Any progress in these toy domains may quickly be applied elsewhere, which is the reason why the research of these problem domains is important.

Most of the literature about artificial intelligence focuses on discovering optimal solutions, yet many real-world problems, such as pathfinding in computer games, robotic navigation, and planning, are so challenging that obtaining optimal solutions is often too slow. [3] Because of the complexity of finding these optimal solutions, a focus of research may be to find suboptimal solutions of good quality in satisfactory time. This is the focus of this thesis: to quickly obtain solutions of the 15-puzzle that are often optimal or near-optimal, while utilizing artificial neural networks.

The thesis is structured as follows. The first chapter describes the necessary background of solving the (N^2-1)-puzzle and defines the essential terms and concepts of heuristic search. The first chapter also analyzes other approaches to solving the (N^2-1)-puzzle besides heuristic search. The second chapter gives an overview of the current state of research on the topic of solving the (N^2-1)-puzzle with the use of artificial neural networks. The third chapter describes the ideas behind designing heuristics based on artificial neural networks, and discusses the training of these heuristics. The last chapter focuses on experimental evaluation of the new heuristics.

Goal of Thesis

There are two main goals of this thesis.

The first goal is to perform research of the current state-of-the-art methods for solving the (N^2-1)-puzzle, with the focus on methods employing artificial neural networks. This will be covered in the first two chapters of this thesis. Research will explore various philosophies and techniques which have previously been used for solving the (N^2-1)-puzzle, and will present them in a clear and informative way to the reader. This part of the thesis will also analyze the history and potential for the use of artificial neural networks in solving the (N^2-1)-puzzle.

The second goal of this thesis is to improve upon the existing applications of artificial neural networks in solving the (N^2-1)-puzzle in a near-optimal manner. This will be the focus of the third and fourth chapter of this thesis. A state-of-the-art method utilizing artificial neural networks will be selected and a prototype will be implemented, which will then be evaluated using various benchmarks.

Chapter 1

Background

1.1 The (N^2-1)-puzzle

1.1.1 History

The (N^2-1)-puzzle has a long and rich history, with many contradictory claims and false popular beliefs. While it is known that the 15-puzzle was invented in the 1870s, it is not known for certain who it's creator was: while most of literature states the chess player Sam Loyd, Slocum and Sonneveld disagree and have stated postmaster Noyes Palmer Chapman as the puzzle's inventor [4]. In any case, the 15-puzzle started a world-wide craze in the year of 1880 [4]. This craze initiated partly because of Sam Loyd's $1000 cash prize offer for solving a particular state of the puzzle, namely the goal state with the tiles 14 and 15 reversed. However, William Johnson and William Story proved that this assignment was impossible, as the state space of the 15-puzzle is divided into even and odd permutations which can't be transformed into each other. Solving Loyd's assignment would require switching between these two types of permutations and is hence impossible to solve [5].

The popularity of the (N^2-1)-puzzle has not faded much over time, as it remains one of the most popular puzzles in the world. Recently, the puzzle has appeared in a digital version on computers and smartphones, and even as a mini-game in larger computer games, such as in The Legend of Zelda (Nintendo, 2003) and Machinarium (Amanita Design, 2009) [6].

1.1.2 Problem definition

The (N^2-1)-puzzle consists of a set of pebbles $P = \{1, 2, \ldots, N^2 - 1\}$, which are placed on a square board of size $N \times N$ with positions numbered from 1 to N^2. A configuration can be expressed as an assignment $c : P \to \{1, 2, ..., N^2\}$, which maps the pebbles to the positions on the board. [7]

Definition. *The (N^2-1)-puzzle is a quadruple* $\Phi = [N, P, c_0, c_g]$*, where* N

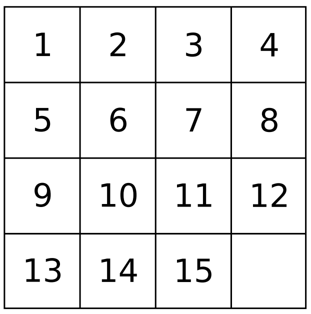


Figure 1.1: A solved 15-puzzle

is the puzzle's size, $P = \{1, 2, \ldots, N^2 - 1\}$ is a set of pebbles, $c_0 : P \rightarrow \{1, 2, \ldots, N^2\}$ is an initial configuration of tiles and $c_g : P \rightarrow \{1, 2, \ldots, N^2\}$ is a goal configuration (an identity with $c_g(p) = p$ for $p \in \{1, 2, \ldots, N^2 - 1\}$). [7]

One position on the board always remains empty and is called the blank. This blank position allows to rearrange the puzzle by sliding a pebble into it. The objective of the (N^2-1)-puzzle is to rearrange the pebbles from the initial configuration c_0 to the goal configuration c_g using a set of moves: left, right, up and down. A move is valid for a given configuration only if the target position adjacent to the moved pebble is blank.

The solution sequence can be expressed as $\sigma = [m_1, m_2, ..., m_l]$ where $m_i \in \{Up, Down, Left, Right\}$ represents a single valid move. We then call l the solution length or the solution cost and say that the solution is *optimal* if there exists no other solution sequence from c_0 to c_g whose solution length is shorter. [7]

There is a limited number of the different solvable configurations of the (N^2-1)-puzzle. This set of solvable configurations is called the state-space: the state-space for the 8-puzzle contains over 10^5 states (or configurations), the 15-puzzle space contains over 10^{13} nodes, and the 24-puzzle space contains almost 10^{25} nodes [2]. This is exactly half of the total number of configurations, as half of them is unsolvable.

1.1.3 Obtaining random configurations

There are two straightforward methods for obtaining random configurations of the (N^2-1)-puzzle: the first method is to randomly shuffle the board, and the second method is to obtain random permutations of the pebble positions.

1.1.3.1 Random shuffling

Random shuffling is done by randomly performing n moves, starting from the goal state. In some implementations, the move opposite to the last move may be disallowed. In any case, n should be set to a relatively high number, preferably several thousands of shuffles. Otherwise, the optimal solutions of the obtained configurations will be very short.

1.1.3.2 Random permutations

Random permutations may be used to directly generate the list of pebble positions on the board. However, a check must be made to see if such a generated configuration is actually solvable, as half of the puzzle configurations obtained with random permutations are unsolvable. This check consists of counting the number of inversions in the configuration permutation and possibly (based on N) of counting the distance of the blank from the bottom row [8]. If a generated configuration is found to be unsolvable, it is discarded and the process is iterated until a solvable configuration is obtained.

Average and maximum optimal solution costs can be established: for the 15-puzzle, the average optimal solution cost is about 52.6 moves, and the maximum optimal solution cost is 80 moves. For the 24-puzzle, the average optimal solution cost is over 100 moves, while the maximum optimal solution cost is unknown. [2, 5]

1.1.3.3 Convergence of the two approaches

Interestingly, experiments show that these two approaches to obtaining random configurations converge, that is, the distributions of optimal solution lengths become more and more equal as n in random shuffling increases. For n of 10,000, the distribution of optimal solution lengths obtained by random

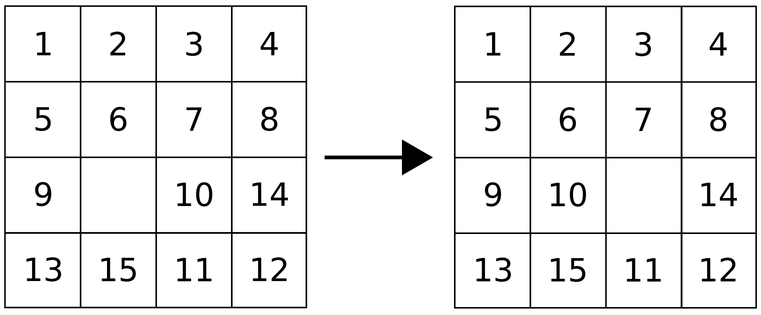


Figure 1.2: Boards before and after moving pebble 10 to the left

shuffling is almost identical to the distribution obtained with random permutations.

1.1.4 Similar problem domains

The (N^2-1)-puzzle is quite similar to two other famous problems: the Rubik's cube (invented by Erno Rubik in 1974, 100 years later than the 15-puzzle [5]) and the Towers of Hanoi puzzle. Both of these problems have a discrete state-space and contain only a predefined set of actions. While the state-space of the Towers of Hanoi is much smaller than that of the (N^2-1)-puzzle (for the respective N), the state-space of the Rubik's cube is much larger, with about 4.3×10^{19} states for the $3 \times 3 \times 3$ Rubik's cube [5]. The Rubik's cube is also much more difficult for humans to solve: even once a part of the puzzle is completed, it must be temporarily dismantled in order for the remaining portions to be solved.

1.2 Rule-based methods for solving

The most straightforward way to solve the (N^2-1)-puzzle is to use a rule-based algorithm. These algorithms are typically defined by a set of rules, which define only the next move (or several moves) at a time. Unlike other algorithms, rule-based algorithms perform very little to no search. Thanks to this, some rule-based algorithms are *real-time*, which means that $O(1)$ computation time is required before computing the next move. [9] On the other hand, some rule-based algorithms have polynomial complexity. All rule-based algorithms are sub-optimal, which results in the vast majority of solutions being far from optimal. Another big disadvantage of rule-based algorithms is that they are often very domain specific, that is, they can't be applied to other problem domains. On the other side, these algorithms are fast and require very little memory.

An example of a rule-based algorithm for the (N^2-1)-puzzle is the real-time Parberry's algorithm, which works by recursively arranging pebbles to their goal positions one by one, starting in the left-most column and the top-most row, then moving on to do the same on the remaining smaller square area. Movement of a pebble is enabled by vacating the position next to it in the direction along a path towards the pebble's goal. The algorithm handles many special cases by introducing a number of rules. [9, 10] Some enhancement techniques exist, such as moving the pebbles in pairs or triplets called snakes, which slightly pushes the algorithm towards better optimality of solutions. [10]

Rule-based algorithms may be successfully used when any solution to the problem is satisfactory, that is, when even solutions which are far from optimal are sufficient. On the other hand, most algorithms that produce optimal solutions run much slower than rule-based algorithms. Clearly there is a trade-off between speed and quality of solutions. Because of the high sub-optimality and limited space for the use of artificial neural networks, the rest of this thesis does not consider rule-based methods for solving, and instead focuses on other methods, which produce optimal or near-optimal solutions.

1.3 MAPF approach to solving

Multi-agent path finding (MAPF), sometimes called *cooperative path finding* (CPF), is a very different approach to solving the (N^2-1)-puzzle. In MAPF, instead of an N×N board with pebbles, the problem consists of a graph $G = (V, E)$, k entities called agents and labeled $a_1, a_2 \dots a_k$ (in the (N^2-1)-puzzle, the agents play the role of pebbles), and a start and goal vertex $start_i \in V$ and $goal_i \in V$ for each agent. The vertices V represent the positions on the puzzle's board, and the edges E represent the neighboring positions. At each time step, an agent can either stay idle or move to a neighboring vertex. The goal is to find a sequence of actions for each agent such that each agent gets from its $start_i$ to its $goal_i$ vertex without any conflicts, while minimizing a global cost function. The cost function is usually defined as the number of time steps required to move all agents into their goal positions. A conflict occurs when two agents share a vertex at the same time, or when two agents switch their respective vertices in a single time step (in the (N^2-1)-puzzle, this would seem as two pebbles sliding through each other).

As MAPF is a generalization of the (N^2-1)-puzzle [10], many different kinds of algorithms exist for solving very different kinds of MAPF problems (with various topologies and sizes of graphs and different placements of agents). There is no universally dominant algorithm which would excel at solving all MAPF problems.

MAPF algorithms can be classified by the computing setting, which can either be *centralized* (a single CPU which has complete information of all the agents) or *decentralized* (each agent having its own computing power and only limited information of the environment). It is much easier to work in the centralized computing setting, to which two main approaches exist - *decoupled* and *coupled*. In the decoupled approach, paths are planned for each agent separately. This results in relatively fast algorithms, but optimality is usually not guaranteed. The coupled approach formalizes MAPF as a global, single-agent-like search problem. [11]

There are two modern centralized, coupled MAPF algorithms: ICTS [12] and CBS [11]. Both of these algorithms are guaranteed to produce optimal solutions. The following sections analyze them and show that none of these algorithms is suitable for solving the (N^2-1)-puzzle.

1.3.1 The increasing cost tree search algorithm

The Increasing Cost Tree Search (ICTS) algorithm works on two levels: the high-level and the low-level. The algorithm is based on the understanding that an optimal solution is built from the individual paths of agents, each of which has a specific (optimal) length.

In the high-level, a tree called *Increasing Cost Tree* (ICT) is searched in a breadth-first manner. In ICT, each node s consists of a vector of individual path costs, $s = [C_1, C_2, \dots C_k]$, with one cost per agent. Each of these nodes represents all possible solutions to the problem in which the individual cost of the path of agent a_i is exactly C_i. Each node of the ICT has exactly k child nodes, each of which is generated by adding a unit cost to one of the path costs. The root node of ICT is defined as $[opt_1, opt_2, \dots opt_3]$, where opt_i is the cost of an optimal individual path of agent a_i, which assumes that this agent is the only agent present in the graph (in the ($\mathrm{N}^2{-}1$)-puzzle, this would be the Manhattan distance of the pebble from its goal position). A node $[C_1, C_2, \dots, C_k]$ is a goal node if the solution, in which the individual cost of the path of agent a_i is exactly C_i, is without any conflicts.

It is clear that a breadth-first search of the ICT will produce the optimal solution, given a correct goal function. This goal function, which checks if a non-conflicting solution composed of individual paths of length C_i exists, is defined by the low-level of the ICTS algorithm. More details can be found in the original ICTS paper [12].

The ICTS algorithm introduces room for the application of artificial neural networks. Artificial neural networks could produce a lower-bound estimate of the costs for the root node of the ICT, given the encoded initial positions of the agents. Unfortunately, the ICTS algorithm seems to be unsuitable for solving the ($\mathrm{N}^2{-}1$)-puzzle. Experimentally, ICTS has outperformed the A* algorithm only in problems where the depth of the goal node in ICT was lower than the number of agents, and vice versa. [11] Clearly, in case of the ($\mathrm{N}^2{-}1$)-puzzle, the depth of the goal node would be much greater than the number of agents. However, if a precise artificial neural network predicted exactly the correct numbers for the estimate of the costs for the root node of the ICT, the search would degrade to running only the low-level of the algorithm. Unfortunately, the authors of the original ICTS paper do not state the time complexity of the low-level. More research is needed to analyze this, however, the analysis is out of the scope of this thesis.

1.3.2 The conflict-based search algorithm

Like ICTS, the Conflict-Based Search (CBS) algorithm also works at two levels. The general idea is to decompose the MAPF problem into a large number of single-agent path-finding problems. In the high-level, a *constraint tree* is searched. Each node in the constraint tree carries a set of constraints, each of which prohibits a specific agent from occupying a specific vertex at a specific time. Given the list of constraints for a node, the low-level is run, which finds a path for each agent such that no constraints of that node are violated. These paths are then pair-wise searched for conflicts. If no conflict is found, the paths are returned as the solution, otherwise, a new constraint must be added to the constraint tree in order to prevent the conflict. It is clear that in order to resolve a conflict in which two agents share a vertex at the same time, a constraint must be added which prohibits one of the agents from occupying that vertex at that time. In order to ensure optimality, two child nodes are created, each inheriting the parent node's constraints, and each adding a new constraint for one of the conflicting agents. Adding both constraints independently to the constraint tree ensures optimality of the algorithm. The low-level and other details of the CBS algorithm can be found in the original paper [11].

Unfortunately, like the ICTS algorithm, CBS is also not suitable for solving the (N^2-1)-puzzle. The algorithm runs in exponential time with respect to the number of encountered conflicts. This can be useful for processing sparsely populated graphs, but not for the dense environment of the (N^2-1)-puzzle, where an enormous amount of conflicts would occur. Experiments have shown that the A* algorithm dominates CBS even on graphs with a much lower room for conflicts than in the (N^2-1)-puzzle. [11]

1.3.3 Discussion of the MAPF approach

Overall, the MAPF approach to solving the (N^2-1)-puzzle with pebbles as agents is slow, as almost all of the vertices are occupied by an agent — which leaves very little room for manipulation with the individual agents at each time step. It seems to be much more efficient to operate with just the blank as "an agent" (instead of the pebbles), as this enables working with just four actions per time step. Because of this, this thesis does not follow the topic of MAPF any further and instead focuses on other methods.

1.4 State-space search approach to solving

Instead of working with every pebble as an agent, solving the (N^2-1)-puzzle can be approached by working with only one agent, the blank. This "agent" has a set of actions, which can be used to change the state of the world. This is the basic idea behind state-space search. In state-space search, each state represents a configuration of the (N^2-1)-puzzle. States are connected by actions, which come from the set of $\{Up, Down, Left, Right\}$ and represent moving the blank between two configurations (or states). States and actions together create a state-space, or a solution graph, which can then be searched from the starting position with the intention of obtaining a sequence of actions that reaches the goal.

Algorithm 1 The tree search algorithm [13, p. 77]

```
function Tree-Search(initial_node)
    open ← empty data structure
    add initial_node to open
    loop
        if open is empty then return failure
        node ← node chosen by search strategy, removed from open
        if node contains a goal state then return Solution-Sequence(node)
        expand node, add resulting nodes to open
    end loop
end function
```

With a given initial state, the possible action sequences together create a search tree, whose root is at the initial state. The branches of the tree represent actions and the nodes represent the states in the state-space of the problem. [13, p. 95] While searching, this sequence of actions is repeated: first, a node is selected (starting with the initial node), and is checked whether the underlying state is the goal state. If it is not, the node is "expanded", meaning that each legal action is applied, which generates a new set of nodes. These generated nodes are put aside for later expansion. The set of all nodes which have already been generated, but not yet expanded, is called the *frontier*, or the *open list* (this is a misnomer as the underlying data structure does not have to be a list). This general algorithm is called *tree search*, and virtually all search algorithms share its structure; they mostly differ only by the method of selecting a node for expansion (this is called the *search strategy*).

The main problem of the tree search algorithm is encountered when there is more than one way to get from one state to another, as is the case of the

Algorithm 2 The graph search algorithm [13, p. 77]

```
function Graph-Search(initial_node)
    open ← empty data structure
    closed ← empty set
    add initial_node to open
    loop
        if open is empty then return failure
        node ← node chosen by search strategy, removed from open
        if node contains a goal state then return Solution-Sequence(node)
        closed ← closed ∪ {node}
        expand node, add resulting nodes to open
            only if not already in open or closed
    end loop
end function
```

(N^2-1)-puzzle. These *redundant paths* make the resulting complexity of the algorithm much higher than necessary, and can sometimes even cause the algorithm to run forever (e.g. when there are loops in the state-space and an unsuitable search strategy is used). In case of the (N^2-1)-puzzle, a search tree of depth d has about 4^d leaves, but the number of distinct states within d steps from an initial state is only about $2d^2$ [13, p. 97]. Luckily, it is easy to improve the tree search algorithm by introducing the *closed list*, which is a data structure that stores every expanded node. Upon expanding a node, each generated node is checked against the closed list (and sometimes also against the open list) and if it is found there, the generated node is discarded instead of being added to the open list. This improved algorithm is called *graph search*, and is the basis of most of the algorithms discussed later in this thesis.

Once the goal state is found by a search algorithm, getting the solution sequence is easy. Each node must simply remember its parent, so that when the goal state is reached, the sequence of actions is reconstructed backwards.

There are two main classes of search strategies: uninformed and informed. Uninformed strategies have no special knowledge about the problem states beyond just the problem definition. All the uninformed search strategies can do is expand nodes in order to generate successor nodes, and distinguish goal states from non-goal states. On the other hand, informed search strategies use problem-specific knowledge, beyond just the problem definition. They can decide whether one non-goal state is "more promising" than another, so that the corresponding nodes can be expanded in the right order. [13, p. 81] Informed search algorithms can tell which parts of the state-space are far away from the solution, so they often expand less nodes than their uninformed counterparts, which leads to shorter run-time.

The most common way additional knowledge is added to the informed search strategies is through *heuristic functions*. A heuristic function, usually denoted h, estimates the cost of the optimal solution from a node to the goal state. The value of $h(node)$ can then be useful to the search strategy of an algorithm. Heuristics can be categorized by two important properties, **admissibility** and **consistency**; optimality of informed search algorithms usually strongly depends on these two properties. An admissible heuristic always underestimates or matches the true cost of reaching the goal. This means that an admissible heuristic is "optimistic": it never overestimates the true optimal solution cost. A heuristic h is consistent if "*for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n': $h(n) \leq c(n, a, n') + h(n')$*" [13, p. 95]. All consistent heuristics are also admissible. In algorithms like IDA*, admissibility is a sufficient condition for guaranteeing optimality, but it is not a necessary condition, as there are cases in which heuristic search with a non-admissible heuristic produces an optimal solution [14].

Many heuristic functions of different qualities exist for the (N^2-1)-puzzle problem. These heuristic functions can be compared by the values they return for different nodes. Ideally, the value returned by a heuristic function would be admissible and as close as possible to the optimal solution length [3]. A heuristic function h_1 is said to dominate h_2 if for any node n, $h_1(n)$ is greater or equal to $h_2(n)$. Dominating heuristics are preferred as they provide tighter lower-bound estimates, which generally results in fewer states being expanded by informed algorithms such as A* and IDA*. However, this is only a rule of thumb. As Robert Holte pointed out in [15], there are some cases in which using a more accurate heuristic results in a larger number of states being expanded.

1.4.1 The breadth-first search algorithm

Breadth-first search (BFS) is a very simple uninformed search algorithm. It works by first expanding the initial node, then all the successors of the expanded node, then all the successors of the successors, and so on. This effectively means that first the nodes at depth 1 from the initial node get expanded, then the nodes at depth 2, etc., which ensures that when a state is first encountered, it must be on the shortest path from the initial node. This behavior is made possible by using a *first in, first out* (FIFO) queue as the data structure for the open list. Because of these properties, the BFS algorithm is guaranteed

Algorithm 3 The breadth-first search algorithm [13, p. 82]

```
function BFS(initial_node)
    if initial_node contains a goal state
        then return Solution-Sequence(initial_node)
    open ← empty FIFO queue
    closed ← empty set
    push initial_node to open
    loop
        if open is empty then return failure
        node ← pop a node from open
        closed ← closed ∪ {node}
        for each child_node in Expand(node) do
            if child_node not in open or closed then
                if child_node contains a goal state
                    then return Solution-Sequence(child_node)
                push child_node to open
            end if
        end for
    end loop
end function
```

to produce optimal solutions to the (N^2-1)-puzzle.

Algorithm 3 shows a slightly augmented version of the original BFS, which applies the goal test when a node is first generated, rather than when it is expanded. This improvement lowers time complexity of the algorithm from $O(b^{d+1})$ to $O(b^d)$, where b is the *branching factor* (the maximum number of successors of a node) and d is the optimal solution depth. [13, p. 82] However, this time complexity is still not satisfactory and only allows the use of BFS for solving random instances of the 8-puzzle, or very simple instances of the 15-puzzle. In order to solve random instances of the latter, more advanced algorithms are necessary.

Another disadvantage of the BFS algorithm is its space complexity, which is also $O(b^d)$. This often results in the algorithm running out of memory early on during search.

1.4.2 The depth-first search algorithm

Depth-first search (DFS) is another simple uninformed search algorithm. DFS works by always expanding the deepest node in the open list. The behavior is as follows: the algorithm first dives directly to the deepest nodes of the state space, where the nodes have no unexpanded successors. After expanding these last nodes and removing them from the open list, the algorithm backs up to the deepest node that still has unexpanded successors. [13, p. 85] This behavior is made possible by using a *last in, first out* (LIFO) queue as the open list of the tree-search or graph-search algorithm.

The properties of DFS depend on whether the tree-search or the graph-search version is used [13, p. 86]. The tree-search version, shown in Algorithm 4, may run forever when applied to the (N^2-1)-puzzle problem, as it would possibly get stuck in a loop (this depends on the initial state and exact implementation). However, its space complexity is only $O(bm)$, where m is the maximum depth. In contrast, the graph-search version is guaranteed to find a solution, but most likely a highly sub-optimal one and with space complexity

Algorithm 4 Tree version of the depth-first search algorithm

```
function DFS(initial_node)
    open ← empty LIFO queue
    push initial_node to open
    loop
        if open is empty then return failure
        node ← pop a node from open
        if node contains a goal state then return SOLUTION-SEQUENCE(node)
        expand node, add resulting nodes to open
    end loop
end function
```

of $O(b^m)$. [13, p. 87] Also, because the algorithm searches the nodes in a somewhat random manner, it is likely to take a very long time before finding a solution. The reason for including the definition of DFS in this thesis, even though the algorithm is slow and produces highly suboptimal solutions, is that its modified version, *depth-limited search*, is part of the IDA* algorithm, which is discussed later in section 1.4.4.

Depth-limited search is a variant of the tree version of the DFS algorithm. Depth-limited search introduces a depth limit, l, which forces nodes at depth l from the initial node to behave as if they had no successors. This can be implemented by introducing a new variable l to the DFS algorithm shown in Algorithm 4, and not adding new nodes to the open list when expanding a node with depth larger than l.

1.4.3 The A* search algorithm

A* belongs to the family of *best-first search* algorithms, that is, algorithms which always expand the most "promising" node first, and arguably is the most popular informed search algorithm. It evaluates nodes by the function f,

$$f(n) = g(n) + h(n), \tag{1.1}$$

where $g(n)$ is the cost of the current path from the start node to node n, and $h(n)$ is the heuristic estimate of the cheapest path from n to the goal node. A* always expands the node with the smallest $f(n)$ first. This is achieved by using a *priority queue* as the open list.

Properties of A* depend on whether the tree search or the graph search version is used. The tree search version is guaranteed to always return the optimal path from the start to the goal if the heuristic function used is admissible. On the other hand, the graph search version, shown in Algorithm 5, is guaranteed to return the optimal solution only if its heuristic function is consistent. The disadvantage of the tree search version is that it tends to recalculate large subtrees if a shorter path to an already processed node is found.

The time complexity of the algorithm strongly depends on the heuristic function used. In practice, however, A* search coupled with a reasonably powerful heuristic usually achieves much better performance than uninformed search. Once again, a strong disadvantage of the graph version of the A* search algorithm is its space complexity, as the algorithm must hold all the generated nodes in memory.

1.4.4 The iterative deepening A* algorithm

The *iterative deepening A** (IDA*) algorithm is another informed search algorithm. It works on two levels. First, the high-level establishes a lower bound of the solution cost, which is provided by the admissible heuristic h. This

Algorithm 5 Graph version of the A* search algorithm

```
function A-STAR($initial\_node$)
    $open \leftarrow$ empty priority queue
    $closed \leftarrow$ empty set
    $open$.ENQUEUE($initial\_node, h(initial\_node)$)
    loop
        if $open$ is empty then return failure
        $node \leftarrow$ pop the node with lowest priority from $open$
        if $node$ contains a goal state then return SOLUTION-SEQUENCE($node$)
        for each $child\_node$ in EXPAND($node$) do
            if $child\_node$ not in $closed$ then
                $child\_estimated\_cost = \text{COST}(node) + 1 + h(child\_node)$
                if $child\_node$ not in $open$ then
                    $open$.ENQUEUE($child\_node, child\_estimated\_cost$)
                else
                    $prev\_estimated\_cost = open.\text{GET-PRIORITY}(child\_node)$
                    if $child\_estimated\_cost < prev\_estimated\_cost$ then
                        $open$.UPDATE-PRIORITY($child\_node, child\_estimated\_cost$)
                        UPDATE-PARENT($child\_node, node$)
                    end if
                end if
            end if
        end for
        $closed \leftarrow closed \cup \{node\}$
    end loop
end function
```

lower bound is used as a temporary limit of the solution cost. After establishing the cost limit, the low-level is run. The low-level is basically the tree version of the depth-limited search algorithm. Unlike depth-limited search, however, the low-level uses the $f(n)$ value from Equation (1.1) as the cut-off value — whether or not a node is included in the search is determined by its estimated cost, which must be smaller or equal to the current limit [16]. If the low-level fails to find a solution within the current cost limit, the high-level keeps increasing the cost limit by one and re-running the low-level, until the solution is found.

The solutions found by the IDA* algorithm are guaranteed to be optimal if the heuristic function h is admissible [16], which is more forgiving than the graph version of A* (which guarantees optimality only if h is consistent). Another improvement over the graph version of A* is that the space complexity of IDA* is only $O(bd)$, where b is the branching factor and d is the optimal solution depth. As with A*, the time complexity of IDA* depends on the properties of the heuristic function used. However, as the low-level of IDA*

Algorithm 6 The iterative deepening A* search algorithm

```
function IDA-Star(initial_node)
    cost_limit ← h(initial_node)
    loop
        solution_sequence ← Low-Level(initial_node, cost_limit)
        if solution_sequence ≠ failure then return solution_sequence
        cost_limit ← cost_limit + 1
    end loop
end function

function Low-Level(initial_node, cost_limit)
    open ← empty LIFO queue
    push initial_node to open
    loop
        if open is empty then return failure
        node ← pop a node from open
        if node contains a goal state then return Solution-Sequence(node)
        for each child_node in Expand(node) do
            child_estimated_cost ← Cost(node) + h(child_node)
            if child_estimated_cost ≤ cost_limit
                then push child_node to open
        end for
    end loop
end function
```

is based on the tree version of depth-first search, IDA* tends to recalculate subtrees. Despite this, and despite the fact that the low-level must typically be run several times (depending on how far the initial cost limit is from the true optimal solution cost), IDA* coupled with a reasonably powerful heuristic function is sufficient for solving the (N^2-1)-puzzle optimally.

1.4.5 Manhattan distance heuristic

One of the simplest ways to obtain a heuristic for the (N^2-1)-puzzle is to just sum the individual Manhattan distances of the individual pebbles from their goal positions. This is called the Manhattan distance heuristic. Manhattan distance is the distance in a world where agents can only move horizontally or vertically (not diagonally), which is the case of the (N^2-1)-puzzle. The number that the Manhattan distance heuristic returns can also be interpreted as the optimal solution cost of a relaxed version of the problem, where the pebbles assume that no other pebbles exist (so the pebbles can slide through others). Despite its simplicity, the Manhattan distance heuristic is relatively powerful and was used, along with the IDA* algorithm, to find the first optimal solutions of the 15-puzzle [14]. The Manhattan distance heuristic is both admissible and consistent [13].

The drawback of the Manhattan distance heuristic is that in almost all instances of the (N^2-1)-puzzle, the heuristic heavily underestimates the real optimal solution cost, which results in the search algorithms searching through unnecessary parts of the state-space and expanding nodes which are located far from the optimal path. The information bias of the Manhattan distance heuristic comes from the fact that the heuristic considers each pebble as independent, ignoring the pebble interactions and conflicts [14]. Fortunately, the heuristic can be improved by introducing various enhancements, resulting in tighter lower-bound estimates.

The most famous improvement of the Manhattan distance heuristic is the *linear conflict enhancement*. It is very simple: when two pebbles are in their goal row or column, but are reversed relatively to their goal positions, then two moves must be added to the predicted cost: one move for one of the pebbles to step out of its row or column, and one for it to step back in. This does not overestimate the optimal solution length, resulting in an admissible heuristic. [2, 5] Other similar enhancements can be used, such as the *last moves heuristic*, which addresses movements occurring in the last steps of the solution, or the *corner-tiles heuristic*, which applies to some pebble situations around the corners of the board [2]. The downside of these improvements is that they are merely domain-specific hacks, and have little value outside of the field of the (N^2-1)-puzzle. However, some of these improvements can be simulated automatically using methods such as *pattern databases*, without domain-specific reasoning [2].

1.4.6 Pattern database heuristic

The most basic version of the Manhattan distance heuristic assumes that pebbles can move freely without colliding with each other. This results in the heuristic strongly underestimating the optimal solution costs. The heuristic would be more informative if it also took the conflicts which occur between pebbles into account. This is the improvement brought by the pattern database (PDB) heuristic.

The pattern database heuristic divides the pebbles into disjoint groups, each forming a *subproblem*. In each subproblem, the goal is to get the corresponding pebbles and the blank to their goal positions, without caring for the goal positions of the remaining pebbles. The cost of each subproblem is the number of moves of the pebbles corresponding to the subproblem — the moves of pebbles belonging to other subproblems therefore do not contribute to the subproblem's cost. The resulting heuristic estimate for a given board is then the sum of the costs of all subproblems.

The cost of a given subproblem is dependent only on the positions of the subproblem's pebbles and on the position of the blank. It is independent of

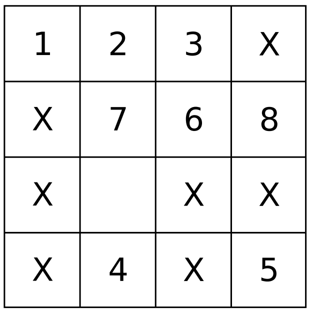


Figure 1.3: A subproblem of the 15-puzzle pattern database heuristic, where the blank and the pebbles 1, 2, 3, 4, 5, 6, 7 and 8 must be placed into their goal positions

the positions of the other pebbles. The costs of all configurations of the subproblem's pebbles can be stored in a database, where the index is formed by the positions of the subproblem's pebbles and the position of the blank. This means that the larger the subproblem is, the more space the database consumes, as it must store the values of all combinations of the pebble positions.

A disadvantage of the pattern database heuristic is that the databases must be computed before the heuristic can be used for making predictions. However, a significant benefit of the pattern database heuristic is that each subproblem's database can be computed in a single backward breadth-first search, which is much faster than performing a search for each of the configurations in the database separately. After the databases of all subproblems have been computed, the heuristic becomes very fast to use, as in order to make a prediction for a given board, it only must query the databases of all subproblems once.

Because each subproblem solves a relaxed version of the original problem and the sets of pebbles corresponding to the subproblems are disjoint, the pattern database heuristic always produces a lower bound for the optimal solution cost and is hence admissible [17]. It is currently unknown whether the PDB heuristic is also consistent.

It is preferable to divide the pebbles into several large subproblems than into many small subproblems, as a subproblem with more pebbles accounts to more potential interactions between pebbles and the resulting heuristic is thus more informative. For example, the PDB heuristic for the 15-puzzle can be composed of only two subproblems of sizes 8 and 7, with the first subproblem formed by pebbles 1, 2, 3, 4, 5, 6, 7 and 8, and the second subproblem formed by pebbles 9, 10, 11, 12, 13, 14 and 15. Such heuristic is then denoted *PDB 7-8*.

1.5 Artificial neural networks

Artificial neural networks are a class of machine learning models. ANNs can be described as computing systems which are able to perform tasks by learning from data, without the need to be explicitly programmed. ANNs were originally inspired by the architecture of neurons in the human brain, although they have gradually become quite different from their biological counterparts.

There are three important properties of ANNs which lead to their current success. ANNs can form an infinitely flexible function, offer all-purpose parameter fitting, and are fast and scalable [18]. These properties deserve further discussion. The ability to form an infinitely flexible function is formally known as the *universal approximation theorem*, which states that a feed-forward ANN with at least a single hidden layer composed of a finite number of neurons can approximate any continuous function with an arbitrary precision. This in essence means that ANNs can be used to fit arbitrarily complex datasets with satisfactory results, given that the network is large enough. The second property states that there exists a method used to train the parameters of the model on an arbitrary dataset. This method is called *backpropagation*, and will be discussed later in this thesis. The last property states that ANNs can be used to effectively tackle both easy and complicated problems: a small network composed of a few neurons can quickly fit a simple dataset, while a large network with hundreds of thousands of neurons can be trained on large amounts of data coming from a complicated problem domain.

The most commonly used architecture of an artificial neural network is the feedforward *multilayer perceptron* (MLP). A MLP is composed of units called *artificial neurons*, which are arranged into layers. The first layer is called the input layer, the following layers are called hidden layers, and the final layer is called the output layer. In a fully-connected MLP, all neurons in a layer are connected with all neurons in their neighboring layers. A diagram showing this arrangement can be seen in figure 1.4. The neurons in the input layer simply supply the input data into the network, and the neurons in the output layer produce the output data. This thesis focuses solely on fully-connected feedforward ANNs.

An artificial neuron is simple. Each connection coming from a neuron in the previous layer is associated with a *weight*. These weights are used as coefficients for the output values of the neurons in the previous layer. All that an artificial neuron does is it first produces a weighted sum of its inputs, then applies an *activation function* to this value, and outputs the result. Each layer also has a special neuron called the bias neuron, whose output value is always 1. These bias neurons are by convention not counted into the total number of neurons in a layer.

If the outputs from the neurons in the previous layer are stored in the vector $\mathbf{x}$, the corresponding weights are stored in the vector $\mathbf{w}$, and the activation function is labeled f, then the output from the neuron is defined as $f(\mathbf{w}^T\mathbf{x})$.

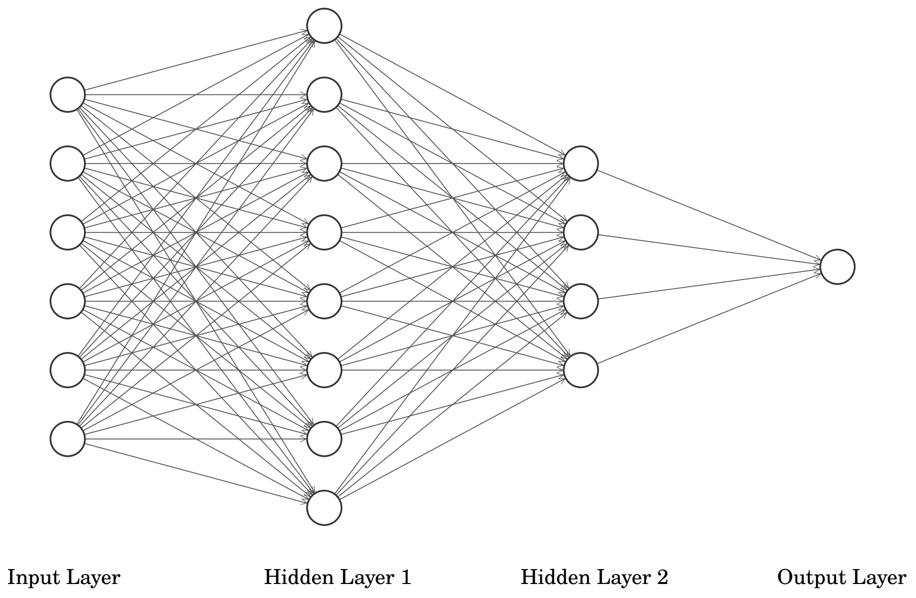


Figure 1.4: Diagram of a fully-connected multilayer perceptron with six neurons in the input layer, eight neurons in the first hidden layer, four neurons in the second hidden layer, and a single neuron in the output layer

The weight w_0, belonging to the output of the bias neuron x_0, is called the *bias term*. The activation function must be non-linear, otherwise the ANN would degrade into a linear model.

A trained artificial neural network produces its output by the following procedure. First, it is fed its input data in the form of outputs of the input neurons. Then the neurons in the first hidden layer calculate their outputs, and these outputs are fed to the next layer. This process is repeated for each following layer, until the signal reaches the output layer, which produces the output value (or output values). The data is often processed in *batches*, or sets of instances. Multiple outputs are then calculated for multiple input instances at the same time, which often results in faster computation.

Given the target output data, a cost function can be used to measure the quality of the outputs produced by the ANN. Arguably, the most commonly used cost function is the mean squared error (MSE), which is defined as

$$\mathrm{MSE}(\hat{Y}, Y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2, \tag{1.2}$$

where n is the number of instances, $\hat{Y}$ is a vector of the predicted values, and Y is a vector of the target values. The higher the cost, the less accurate are the predictions output by the ANN.

In the following sections, the techniques used while training ANNs for experiments in this thesis will be described. The exact definitions are out of the scope of this thesis, so only a brief description and a discussion will be provided for each of the methods.

1.5.1 The backpropagation algorithm

The algorithm commonly used to train artificial neural networks is called *backpropagation*. It was published by Rumelhart et al. in their famous 1986 article [19]. Given its training instances and corresponding target values, the backpropagation algorithm attempts to minimize the cost of the predictions on the training data. The algorithm works as follows: first, for each training instance (or for each batch of training instances), the algorithm feeds the data to the network and computes the outputs of all neurons in each consecutive layer. This is called the forward pass. Then, using the outputs of the neurons in the output layer, the cost of the predictions is calculated, and the algorithm calculates how much each neuron in the last hidden layer contributed to each output neuron's error. The algorithm then proceeds to calculate how much of these error contributions were produced by each neuron in the previous hidden layer, and so on for each previous hidden layer, until the algorithm reaches the input layer. This is called the reverse pass. During the reverse pass, the algorithm calculates the error gradient (in respect to the cost function) across all connection weights in the network. Finally, backpropagation updates all the weights in the network by a small value corresponding to the weights' error gradients. [20, p. 261] This last step is essential, as it readjusts the network's weights so that the next time the network would predict the outputs for the same input data, the predictions would be a little more accurate and the prediction cost would be a little smaller. The error gradients are scaled by a coefficient known as the *learning rate*, which controls the speed at which the ANN's parameters are changed.

By repeatedly performing the process descried above for more and more training instances, the artificial neural network will usually learn to properly approximate the target outputs. However, the goal is usually not to properly predict the targets of only the training instances, but of all instances from the problem domain. It can be expected that as the network is fed more and more training instances (that represent the problem domain reasonably well), the network will learn to correctly respond even to instances it has never been trained on. This is usually the goal of training and when it is successful, it is said that the ANN *generalizes* well.

By today's terms, the backpropagation algorithm would be more accurately described as *gradient descent using automatic differentiation* [20, p. 261]. This is because backpropagation performs gradient descent in respect to the cost function, and calculates the gradients using a method called automatic differentiation. Automatic differentiation lies at the heart of all today's major artificial neural network libraries.

1.5.2 Adam optimization

Simply performing gradient descent, as described in the backpropagation algorithm, is often not sufficient in practice. The cost function with respect to the neural network's parameters is usually not shaped as a simple multidimensional "bowl", but often contains numerous local minima and long flat areas known as plateaus, which must be passed in order to get to the global optimum. The plain gradient descent optimizer suffers from the problems of getting stuck in the local optima, or taking too long to cross the plateaus. Also, when a part of the cost function is shaped as a bowl elongated in one dimension (with respect to one of the ANN's parameters), gradient descent would first get to the bottom part of the valley in the steepest dimension, and only then start to slowly proceed to the bottom of the valley in the other dimension. It would be faster for the optimizer to detect this early on and start going in the right direction which points straight to the bottom in the first place. [20, p. 296]

The Adam optimizer, whose name is derived from the term *adaptive moment estimation*, addresses these problems by borrowing methods introduced by two other optimizers, the momentum optimizer and the AdaGrad optimizer (or its advanced version, the RMSProp optimizer). The momentum optimizer works by remembering the past few gradients of the cost function, which enables it to gain momentum while going downhill. This allows the momentum optimizer to speed up when descending, and quickly cross long plateaus as it keeps its speed after going downhill. The momentum optimizer can even escape small local optima, if they are preceded by a large enough hill. On the other hand, the AdaGrad optimizer mitigates the elongated bowl problem by gradually slowing down, but doing so faster for the steep dimensions than for the dimensions with gentler slopes. This would often cause the optimizer to completely stop before reaching the optimum, however, so the RMSProp optimizer improves upon AdaGrad by only taking into account the gradients from the most recent iterations. [20, p. 297] The Adam optimizer simply combines the ideas of momentum and RMSProp optimization, which turns it into a very powerful all-purpose optimizer suitable for a large majority of tasks.

1.5.3 The problem of unstable gradients

As was discussed in the section on the backpropagation algorithm, backpropagation works by computing the gradient of the cost function with regard to the weights in the ANN, and then using these gradients to update the weights' values. [20, p. 275] The problem with gradients often is that they get smaller as backpropagation progresses back to the first hidden layers (known as lower layers), which results in the weights in the lower layers being changed very little or virtually not at all, slowing training of the lower layers down or even making it impossible. This is known as the vanishing gradients problem. On the other hand, sometimes the gradients can get bigger and bigger as backpropagation progresses to the lower layers, which is known as the exploding gradients problem. This also makes learning problematic as the neural network's parameters can diverge. However, exploding gradients usually do not occur in feedforward ANNs [20, p. 276].

In general, unstable gradients make learning of artificial neural networks problematic, especially in networks that are composed of multiple hidden layers. However, several techniques exist that mitigate the problem of unstable gradients. These techniques include using an appropriate activation function, initializing the ANN's weights using a suitable random distribution, and using a method known as batch normalization.

1.5.3.1 The ELU activation function

The traditionally used activation function in artificial neural networks, the sigmoid, which is defined as $f(x) = 1/(1+e^{-x})$, suffers from a problem known as *saturation* when its inputs are very large or very small. Saturation means that with a large or small input, the output gets very close to either 0 or 1, and the derivative becomes very small, so backpropagation has no gradient to propagate back through the network and training stalls [20, p. 276]. The *rectified linear unit* activation function, known as ReLU and defined as $f(x) = \max(x, 0)$, attempts to solve this problem by not saturating for positive values, as the function is not bounded from above. However, the ReLU activation function suffers from another problem known as *dying ReLUs*: when the input to a neuron which uses the ReLU activation function is negative, the neuron starts outputting zero, and is unlikely to start outputting anything else as the gradient of the ReLU function also became zero [20, p. 279].

The *exponential linear unit* activation function, known as ELU, improves upon ReLU by having a non-zero derivative everywhere. It is defined as

$$\mathrm{ELU}(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0, \end{cases} \tag{1.3}$$

where α is a hyperparameter controlling the slope of the negative section and is usually set to 1. Even though ELU saturates for negative inputs, it

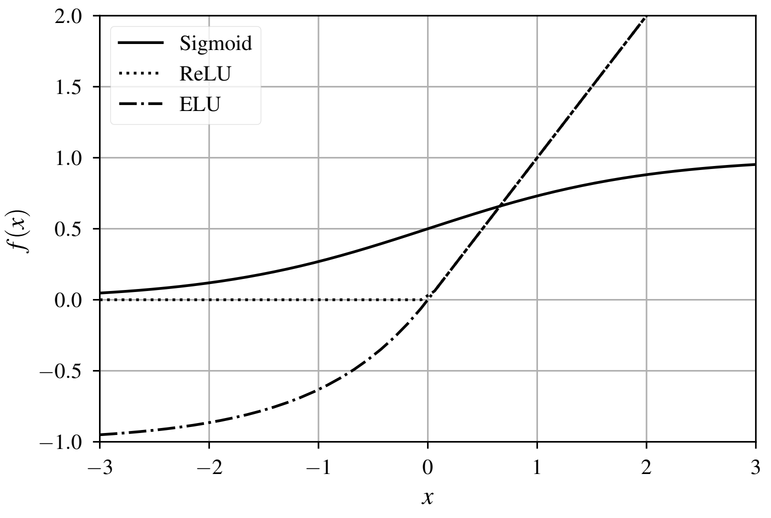


Figure 1.5: Comparison of the sigmoid, ReLU, and ELU activation functions

won't usually get absolutely stuck, as there always is at least a small non-zero gradient to work with. Another benefit of the ELU activation function is that it can output negative values, which allows the artificial neuron to have an average output closer to zero, mitigating the vanishing gradients problem [20, p. 281]. However, ELU is slower for computation than the ReLU activation function.

1.5.3.2 He initialization

For the signal to flow properly through an ANN, the variance of the inputs of each layer must be roughly equal to the variance of the outputs [20, p. 277]. Otherwise, problems such as vanishing gradients may occur. For this to work, connection weights in the network should be initialized using a random distribution which works well together with the activation function used. For the ELU activation function, the use of *He normal initialization* is recommended, which initializes the weights using the normal distribution. The distribution is calculated separately for each layer of neurons — the standard deviation of the normal distribution is set using the number of input and output connections leading from the layer. The formula for the standard deviation is

$$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}, \tag{1.4}$$

where n_{inputs} and n_{outputs} is the number of input and output connections of the layer.

1.5.3.3 Batch normalization

Although using He initialization with the ELU activation function ensures that the layers' inputs and outputs are properly distributed at the beginning of training, plain He initialization does not ensure that the problems won't come back during training. Because of this, many today's ANNs with multiple hidden layers use a technique called *batch normalization.* This method adds an operation in the model after the activation function of each layer, which converts the outputs of the current batch to have zero mean and normalizes them. After that, batch normalization scales and shifts the results using two new parameters per layer, which are learned during training. This ensures that the distributions of each layer's outputs are optimal. Batch normalization strongly mitigates the unstable gradients problem, often to the point that even saturating activation functions can be used. The downside of the method is that it slows down network's training and inference. [20, p. 283]

1.5.4 Learning rate

Learning rate is a very important hyperparameter, which must be set before training any artificial neural network. Learning rate controls how much the gradients get applied to the network's weights at each gradient descent step. The learning rate hyperparameter must be set carefully — if it is too low, training can take a very long time, but if it is too high, training can diverge as the optimizer can start to jump around the parameter space, without ever settling for an optimum. Luckily there exists a powerful technique for setting the optimal learning rate. In this technique, an experiment is run before training is started, in which the ANN is trained on a number of batches. The experiment starts with a very low learning rate, but this learning rate is increased for every batch. Also at each batch, the overall loss on the training instances is recorded, and after the last batch, these loss data are analyzed. The learning rate corresponding to the batch during which the loss was decreasing most rapidly is then chosen as the target learning rate to be used for real training, as this is the learning rate with which the neural network seems to learn fastest.

There also exist methods which gradually decrease the learning rate during training, which are known as *learning rate decay.* The rationale behind there methods is that learning should first be rapid, in order to get near the optimum quickly, and then slow down, in order for the optimizer to stop precisely at the optimum. However, the Adam optimizer actually uses adaptive learning rates for each parameter individually, out of the box. It seems that learning rate decay is not used as frequently with the Adam optimizer, and this thesis relies on constant setting of the learning rate hyperparameter.

1.5.5 Dropout

Artificial neural networks can have millions of parameters, which can result in overfitting of the training data. Various methods, generally known as *regularization*, have been developed to prevent this. Arguably, the most popular regularization technique used with ANNs is *dropout*. However, dropout does not only perform regularization, but also makes the model more robust and helps it generalize better. Actually, many state-of-the-art artificial neural networks got a significant accuracy boost after applying dropout. [20, p. 304]

Dropout works by assigning a probability p (called the *dropout rate*) to each neuron to "drop out" for a single training step, meaning that the neuron with all its connections will be entirely ignored at this training step. This forces the network to become more robust, as it can not rely on information coming from any single neuron. This also results in the network being less sensitive to slight changes in the input data, which often makes it generalize better [20, p. 305].

After training of the network is finished, the neurons do not get dropped out anymore. In order to compensate for the fact that the neurons suddenly sum up data coming from more connections than before, the connection weights of each neuron get lowered after training by an amount corresponding to p.

1.5.6 Deep learning

An artificial neural network that has multiple hidden layers of neurons is known as a *deep artificial neural network*. The field of training deep ANNs is called *deep learning*, and has become the focus of an increasing number of researchers in recent years. The reason for the increasing popularity of deep learning is that with today's algorithms, deep ANNs frequently perform better than shallow networks — they often achieve better accuracy, require a smaller total number of neurons, and converge faster during training.

The explanation of why deep neural networks are so powerful is still a topic of debate. One scientific paper published in 2016 showed that for some mathematical functions, the number of neurons needed by a shallow ANN to approximate them is exponentially larger than the total number of neurons needed by a deep ANN [21]. However, another paper showed that even though deep ANNs are easier to train using today's algorithms than shallow networks, shallow ANNs can be made to "mimic" trained deep neural networks, and then their performance becomes similar [22]. This would indicate that the problem is mostly with the way shallow neural networks are trained. In any case, a popular explanation of why deep neural networks often perform better than shallow ANNs says that deeper networks can work with several layers of abstraction. For example, when researchers analyzed the weights of some deep convolutional ANNs trained to recognize objects in pictures, it turned out that the lower layers detected low-level features such as edges, upper layers combined these simple features to produce higher-level features such as textures and shapes, and the top layers used these features to obtain high-level features such as faces. It is possible that a similar hierarchy of features is often present not only in deep convolutional networks, but in other architectures of deep ANNs as well.

In any case, deep learning only gained its popularity in the recent years, mostly because deep neural networks were difficult to train before the introduction of methods that mitigate the problem of unstable gradients.

Chapter 2

Related Work

Some research has already been done in the field of application of ANNs in solving the (N^2-1)-puzzle. Four studies have been carried out, all of which focused on the state-space search approach and used ANNs as heuristic functions.

In 2002, Hou, Zhang and Zhou [23] used statistical learning techniques, namely k-nearest neighbors (k-NN) and ANNs, as heuristics for solving random instances of the 8-puzzle. They used the locations of tiles as inputs to the heuristics, and the estimated optimal solution length as output. The search algorithm used was A*. In their experiments, they first used plain k-NN and later an ANN with multiple hidden layers as heuristic functions, with only limited results. Then they estimated the confidence of the predictions of the ANN using an auto-encoded ANN, and combined the ANN and k-NN using these confidence values into a single estimator. This yielded better results than a single ANN or k-NN model, but was still a less powerful heuristic than Mannhattan distance.

In 2004, Ernandes and Gori [14] used an ANN as a heuristic for the 15-puzzle, along with a learning scheme which biased the optimal solution cost predictions towards admissibility. They used only the locations of the pebbles as input features. The experiments were successful — IDA* with the neural heuristic achieved an optimal solution in about 50% of cases, and the time cost was reduced compared to the Manhattan distance heuristic by a factor of about 500. Unfortunately, the study did not include comparison against a state-of-the-art heuristic, nor against another powerful inadmissible heuristic. Also, Ernandes and Gori only used a small neural network (with a single hidden layer composed of just 15 hidden units), along with a small training set composed of only 25,000 random instances — factors which could have lead to a relatively high bias of the model.

In 2008, Samadi, Felner and Schaeffer [3] used a different approach. Instead of using pebble positions as inputs to the ANN, they used the outputs of various heuristics as features (such as the Manhattan distance and pattern database heuristics) and fed them to their model, which they trained using the corresponding optimal solution costs. The ANN was supposed to discover the relative influences of the heuristics and their mutual relationships, in order to produce estimates of the optimal solution costs. Samadi et al. applied this idea to the 15-puzzle along with an ANN composed of a single hidden layer and a training set of 10,000 instances. They used RBFS as their search algorithm, and utilized a modified cost function to bias the ANN heuristic towards admissibility, in a similar fashion as Ernandes and Gori did in their study [14]. The results were impressive, the unbiased ANN heuristic resulted in search trees smaller by a factor of 10 than those created by the 7-8 PDB heuristic. The generated solutions were inadmissible by about 4%. Using a neural network which was biased towards admissibility resulted in search trees of half the size of those created with the 7-8 PDB heuristic, but with solutions which were by average within 0.1% of optimal. Samadi et al. also compared their heuristics against the weighted RBFS algorithm with the underlying weighted 7-8 PDB heuristic. Weighted RBFS with the weighted 7-8 PDB heuristic lead to similar results as the biased neural heuristic.

In 2011, Arfaee, Zilles and Holte [24] proposed a procedure called bootstrapping, which they used to suboptimally solve random instances of the 24-puzzle. Bootstrapping is an iterative process that uses machine learning to create a series of heuristic functions. An initial heuristic h_0 is used to attempt to solve a set of given 24-puzzle instances. The heuristic h_0 can be too weak to solve all of the instances, but the instances which were solved within the time limit are used as a training set for the next heuristic function, h_1. The trained heuristic h_1 is then used to attempt to solve (possibly suboptimally) the remaining instances, and this procedure is repeated with gradually stronger and stronger heuristics $h_2, h_3, \dots$, until almost all instances from the input set are solved. Arfaee et al. also came up with an effective procedure for solving an input which is not a set of instances, but merely a single instance. The training algorithm used in their experiments was a small artificial neural network with a single hidden layer of three neurons. The input features were not the raw positions of the tiles, but instead outputs of Manhattan distance and pattern database heuristics, position of the blank, and number of out-of-place pebbles. The results of their experiments were positive, the bootstrapping procedure was able to solve the vast majority of the set of random 24-puzzle instances. The suboptimality of the solutions, which was about 6 to 8 percent, was also relatively satisfactory.

It is interesting that only one study so far has applied ANNs to the solving of the 24-puzzle. Also, deep learning has never been successfully applied to the field of the (N^2-1)-puzzle.

Chapter 3

Designing a New Heuristic

3.1 Introduction

As was discussed in Chapter 1, the most promising method for near-optimal solving of the (N^2-1)-puzzle is the state-space search approach. By far the most widespread way of providing additional information to the search algorithms is via heuristic functions. As this thesis focuses on the application of artificial neural networks, a natural question is whether artificial neural networks can be used in designing a custom heuristic function. The answer is positive.

The objective of a heuristic function is simple. Given a description of a state s, the heuristic function h estimates the cost of the optimal solution from state s to the goal state. The state s can be uniquely described by the locations of the pebbles. Hence the assignment is as follows: the desired function h should take pebble locations as input and return the corresponding estimated optimal solution length as output. The estimated optimal solution lengths should be calculated as accurately as possible — the hypothesis here is that in general, the more accurate the predictions are, the less nodes will have to be expanded by the underlying search algorithm. Since the pattern database heuristic tends to underestimate the optimal solution costs, there is potential for the new heuristic to be more accurate than a moderately powerful PDB heuristic (for example, the PDB 7-8 heuristic), and it is hence possible that search algorithms running with the new heuristic will expand fewer nodes than if running with the PDB heuristic. Another important requirement on the new heuristic is that it should try not to overestimate the predictions — it can be assumed that the less overestimations occur during search, the more likely the solutions found by the search algorithms will be optimal. Ideally, the heuristic would be completely admissible, that is, it would never overestimate the optimal solution cost.

Most supervised machine learning methods, not just artificial neural networks, can be used in creating a heuristic function for the (N^2-1)-puzzle. The heart of the heuristic will then be a statistical model which will have learned the relationship between the pebble locations and the corresponding optimal solution costs. In a sense, this heuristic is similar to a pattern database heuristic as it must be built in a pre-calculation phase, before it can be used to make predictions. The process is structured as follows: first, a training set must be built. This set must contain random (N^2-1)-puzzle instances along with the calculated optimal solution costs. Second, the statistical model is trained on the training set. Third, the trained model is used to make predictions $h(s)$ for a search algorithm.

This thesis imposes an important constraint on the new heuristic: it can only learn from the low-level features it is fed, that is, the positions of the pebbles. The machine learning model can not use, for example, the solution cost estimates produced by other heuristic functions. The reason for the introduction of this constraint is simple: it is more interesting and challenging to design a heuristic function which is powerful on its own, without the information gain brought by other heuristic functions. In practice, this constraint is a severe limitation and can be problematic: consider a 15-puzzle instance proposed by Samadi, Felner and Schaeffer [3]. This instance has all pebbles placed on their goal positions, except of pebbles 1, 2 and 3, which are placed at positions 2, 3 and 1. Here the values of the input features are very similar to those of the goal configuration, which would have an optimal solution cost of zero, yet the optimal solution length is 18 in this case. This is deceptive and the statistical model used must be fine-grained enough to recognize the complexity of the solution. At this point, another hypothesis is introduced: a well-trained, moderate-sized artificial neural network will be powerful enough to yield good predictions with only primitive input features, even on (N^2-1)-puzzle instances it has never seen during training. The new heuristic, based on the artificial neural network, will be called **ANN-distance**. The architecture of the artificial neural network used in this heuristic will be a multilayer perceptron with multiple hidden layers, a model which is often referred to simply as a deep artificial neural network. Due to reasons discussed in section 1.5.6, it can be assumed that a deep ANN will be more effective than a shallow ANN. The ANN-distance heuristic will be trained and run on instances of the 15-puzzle. The 8-puzzle is too simple for the heuristic to yield interesting results, while the 25-puzzle is too difficult for current computers, as obtaining a sufficiently sized training dataset would take massive amounts of resources from today's perspective. In order to tackle the 24-puzzle, more innovative methods such as bootstrapping would have to be used, however, this thesis will not focus on them, partly because of limited computing resources — the bootstrapping experiments performed in [24] took days to finish.

3.2 Cost function

An important limitation of a heuristic based on a statistical model is that it can not guarantee admissibility or consistency. However, an advantage of using an artificial neural network is that admissibility can be controlled using a custom cost function, which would penalize overestimations more than underestimations. In effect, a neural network with such cost function can be expected to prefer underestimating its predictions, which pushes it towards admissibility.

3.2.1 Asymmetric mean squared error

The asymmetric mean squared error (AMSE) is a novel cost function which penalizes overestimating the target value more than underestimating it. Asymmetry between the cost of overestimations and underestimations can be controlled using a parameter. The function is defined as

$$\mathrm{AMSE}(\hat{Y}, Y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2(\mathrm{sgn}(\hat{Y}_i - Y_i) + \alpha)^2, \tag{3.1}$$

where n is the number of instances, $\hat{Y}$ is a vector of the predicted values, Y is a vector of the target values, and α is a parameter between 0 and 1, which controls the skew of the function. The closer α is to 1, the more the overestimations are penalized. AMSE with $\alpha = 0$ is equal to the mean squared error cost function.

For clarity, the cost of an individual instance is defined as

$$c(d_i) = d_i^2(\mathrm{sgn}(d_i) + \alpha)^2, \tag{3.2}$$

where d_i is defined as $\hat{Y}_i - Y_i$, or the difference between the predicted value and the real value. The behavior of c for different prediction errors and various values of α is shown in figure 3.1.

3.3 Input features encoding

One of the decisions in designing a heuristic function based on machine learning is how to encode the input features. If the underlying artificial neural network in ANN-distance was presented plain numbers representing pebble positions, the model would be deceived into believing that, for example, pebble 8 has more in common with pebble 9 than with pebble 12, while these pebbles have nothing in common: actually, the former are 4 positions apart from each other in the solved 15-puzzle, while the latter are only one position apart. To avoid this deception, the pebble positions will be *one hot encoded.* This means that, for the (N^2-1)-puzzle with N^2 positions, the input is defined by N^4 bits, where the bit at position $N^2 \times k + t$ is high if the square at position

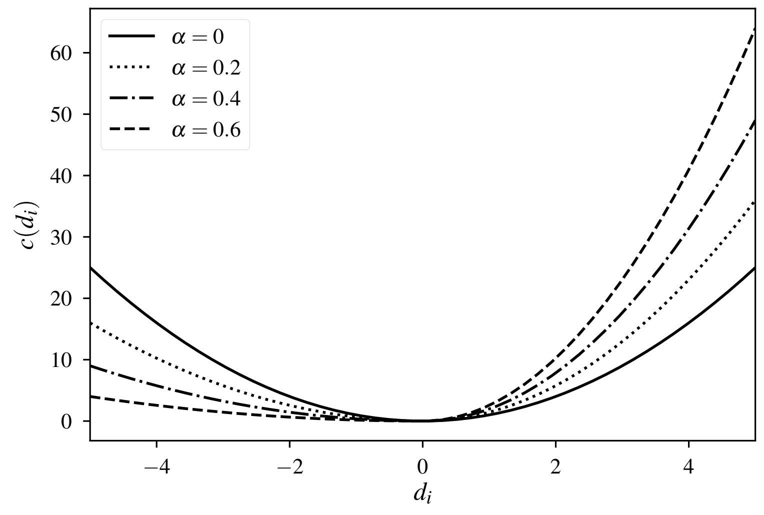


Figure 3.1: The AMSE cost of a single instance for various values of α

k is occupied by the pebble t, and low in every other case [14]. This encoding is very effective in that the pebbles of similar values are no longer perceived as related, but the polynomial growth with respect to N means that in case of the 15-puzzle, 256 input features are needed.

3.4 Training set

Another decision that must be made when designing a heuristic based on a machine learning model is the character of the set of the training instances on which the heuristic will be trained. In order for the heuristic to produce accurate predictions both when the search algorithm is far and close to the solution, the training set should contain instances of both high and low optimal solution costs. However, a test which ran the A* algorithm with the pattern database heuristic revealed that the search algorithm spends most of the time relatively far away from the solution, only converging close to the solution towards the end of the search. Distribution of the true optimal solution costs of the boards which the search algorithm queried for the heuristic estimates is shown in figure 3.2. The search with the ANN-distance heuristic can be expected to behave in a similar way to the PDB heuristic, in the sense that it is likely to spend most of the time far away from the solution. This suggests that it is preferable for the ANN-distance heuristic to be as accurate as possible on predictions for the hard instances. The training dataset should therefore include mostly instances of high optimal solution costs.

In the final training dataset, most instances have a high optimal solution cost of around 45, while some instances have a lower optimal solution cost.

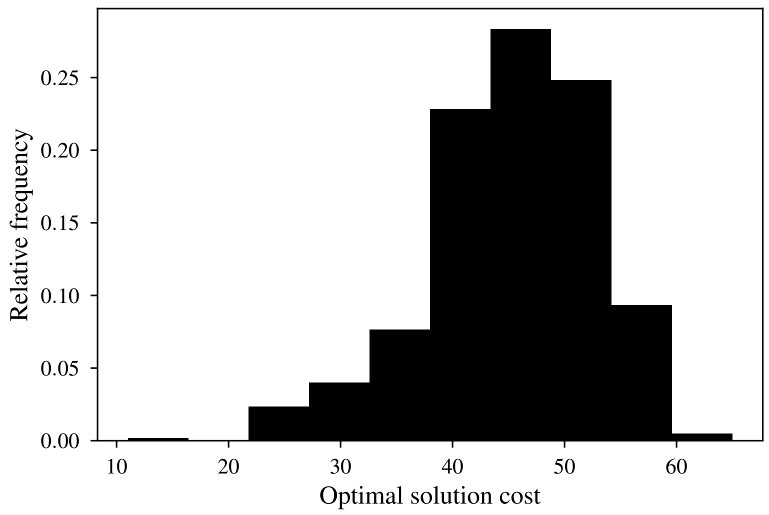


Figure 3.2: Histogram of optimal solution costs of nodes expanded during A* search with the PDB heuristic

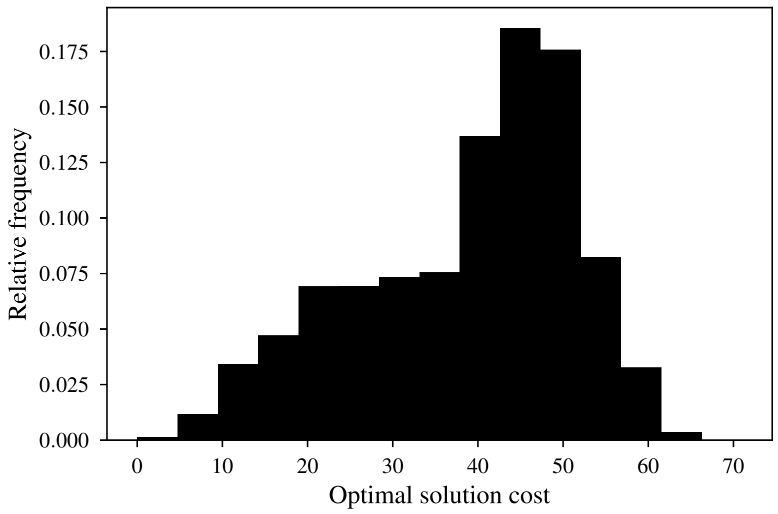


Figure 3.3: Histogram of optimal solution lengths of boards in the training dataset

The distribution can be seen in figure 3.3. The dataset contains about 6 million instances, and was randomly split into two parts — a training set with 4.8 million instances, and a validation set with 1.2 million instances. The instances were generated by random shuffling. Different shuffle counts were used; the hardest instances were generated by shuffling about 10,000 times, while the easier instances were generated by shuffling about 100 times. The algorithm used to obtain optimal solution costs was IDA* with the underlying PDB 7-8 heuristic. The reason for this choice is as follows: obtaining millions of training instances in a reasonable time is only possible using a powerful heuristic, which the PDB 7-8 is. However, because it is unknown whether the PDB heuristic is consistent, the IDA* algorithm had to be used, as it guarantees optimality of solutions even with an inconsistent heuristic. No time limit was imposed on the IDA* searches. Generation time of all the six million instances and their corresponding optimal solution costs was about 120 minutes.

Besides the relatively large size of the training dataset, the dataset covers only a tiny portion of the total state-space of the 15-puzzle. The instances in the dataset were not checked for duplicities, hence it is possible that some instances are present multiple times.

3.5 Training of the ANN-distance heuristics

3.5.1 Hyperparameters and techniques

For some of the hyperparameters of the ANNs, it was impossible to know in advance what values they should be set to. These unpredictable hyperparameters were the number of layers, number of neurons in each layer, and the dropout ratio (set to the same value for all layers). In order to determine what values would work best, several grid searches were run sequentially. Each grid search tried to train the ANN with several different combinations of layer sizes and dropout ratios, and for the values that lead to the lowest validation error, a more fine-grained grid search was run which tried hyperparameter combinations close to the previous best values. The most powerful architecture turned out to be a deep funnel-shaped network with five hidden layers. The layer sizes were 256 neurons for the input layer, then 1024, 1024, 512, 128 and 64 neurons for the hidden layers, and a single neuron for the output layer. Dropout regularization was active for all layers, and the dropout ratio was set to 0.2.

The learning rate was set to 0.01, a value which was experimentally determined by the method described in section 1.5.4. The outputs of all layers were activated using the ELU activation function, except of the output layer, which used no activation. Outputs of all activation functions were batch normalized. Weights of all connections were initialized using He normal initialization. Several experiments with l2 regularization were performed, but even a small l2 regularization value seemed to severely lower the performance of the networks.

3.5.2 Training

The Adam optimizer was used for training of the ANNs. The cost function used was either the mean squared error, or the asymmetric mean squared error with an arbitrary value of α.

The ANNs were trained on the training set for around 40 epochs. This process usually took about 3 hours. At the end of each epoch, the cost on the validation set was calculated, and if this cost was lower than the previous best validation cost, the weights of the network were saved. After the last epoch, the weights corresponding to the best validation performance were loaded and saved as the final trained network. This technique was necessary as the validation cost often fluctuated during training, unlike the cost on the training data, which usually kept decreasing steadily.

3.5.3 Resulting ANN-distance heuristics

Several artificial neural networks were chosen for final evaluation as the underlying ANNs in ANN-distance heuristics. One was a deep artificial neural network with the hidden layers composed of 1024, 1024, 512, 128 and 64 neurons, trained with the mean squared error cost function. Another artificial neural network chosen for further experiments was a shallow ANN with a single hidden layer composed of 2752 neurons. This shallow ANN therefore had the same total number of neurons as the deep neural network. Finally, two deep ANNs (again with the layer sizes described above) were trained using the asymmetric mean squared error, with the α parameter set to 0.4 and 0.8.

The resulting losses on the validation sets were as follows: the deep artificial neural network achieved a validation loss of 1.539 with the MSE cost function. The shallow ANN, which was also trained with the MSE cost function, achieved a validation cost of 2.339, which is significantly higher than the cost achieved with the deep ANN, and the shallow neural network can therefore be expected to produce less accurate predictions. The deep ANN trained with the asymmetric mean squared error cost function with α of 0.4 achieved a validation loss of 1.539, and the deep ANN trained with the AMSE cost function with $\alpha = 0.8$ achieved a validation error of 0.588.

The deep artificial neural networks take up about 23 megabytes of memory, while the shallow ANN takes up about eight megabytes.

3.6 Discussion

In this chapter, five major hypotheses have been stated:

1. The underlying artificial neural network in the ANN-distance heuristic, trained with only pebble positions as input features, will generalize well, that is, it will correctly respond even to instances outside of the training set.

2. The ANN-distance heuristic, trained with only pebble positions as input features, will lead to comparable, or even more accurate estimates of the optimal solution cost than the PDB 7-8 heuristic.

3. Using a more accurate inadmissible heuristic function will generally lead to a lower number of nodes expanded by the search algorithm.
4. Using a heuristic function that tends to overestimate the optimal solution cost less will lead to obtaining an optimal solution more often.
5. Using a heuristic function based on a deep artificial neural network will result in a better performance than using a shallow artificial neural network with the same number of hidden neurons.

The first hypothesis has already been supported by the study performed by Ernandes and Gori in 2004 [14], who used a heuristic based on an ANN, trained it on a small training set with just the pebble positions as input features, and got much better results than with a Manhattan distance heuristic. The second hypothesis has never been tested yet — even though Samadi et al. in 2008 [3] surpassed the performance of the PDB 7-8 heuristic, they trained their ANN using high-level input features and it is unknown whether such performance could be reached using only pebble positions. Evidence for the third hypothesis has been gathered numerous times by running experiments with admissible heuristics, and it seems that the hypothesis is believed to be true by a large number of researchers, but it is unclear whether it holds with inadmissible heuristic functions. As for the fourth hypothesis, Samadi et al. have found supporting evidence, as using a cost function that penalized overestimations while training resulted in obtaing optimal solutions four times more often. The fifth hypothesis has never been tested, as most of previous research of artificial neural networks in the field of the (N^2-1)-puzzle only used shallow networks with a single hidden layer. The only study which used a deep ANN never compared it to a shallow network. In summary, the second, third and fifth hypotheses have never been tested. This thesis will try to find evidence that supports or contradicts them by performing various experiments in the next chapter.

Chapter **4**

Experiments

4.1 Environment description

All experiments (including the training of the ANNs), besides the computation of the pattern database, were run on a cloud machine with an eight-core Intel Xeon E5-2623 v4 CPU, 30 GBs of RAM, and an NVIDIA Quadro P4000 graphics card. The use of the high-end graphics card improved the speed of calculations involved in artificial neural networks, namely the speed of training and inference, by several times.

The pattern databases for the PDB heuristic were calculated on a cloud machine with 24 vCPUs and 350 GBs of RAM.

4.2 Experimental evaluation of the ANN-distance heuristics

4.2.1 Referential heuristics

The ANN-distance heuristics were evaluated against other admissible and non-admissible heuristics, all based on pattern databases, namely a pattern database with patterns of sizes of 7 and 8 (also referred to as *PDB 7-8*). The pattern databases were calculated in two single backward breadth-first searches (one for each pattern), a process which took about 10 hours and consumed around 250 GBs of memory. The exact patterns used were defined as follows: one subproblem database stored the optimal solution costs for the relaxed subproblem of correctly placing pebbles 1, 2, 3, 4, 5, 6, 7 and 8, while the other subproblem database stored the optimal solution costs for the relaxed subproblem of placing the pebbles 9, 10, 11, 12, 13, 14 and 15.

One heuristic used in the tests simply returns the cost estimated by the PDB 7-8, yielding an admissible heuristic function. Other heuristics return the same estimated optimal solution cost, but multiply it by a *weight* $W \geq 1$. This is called a *weighted heuristic* and is used to obtain more accurate esti-

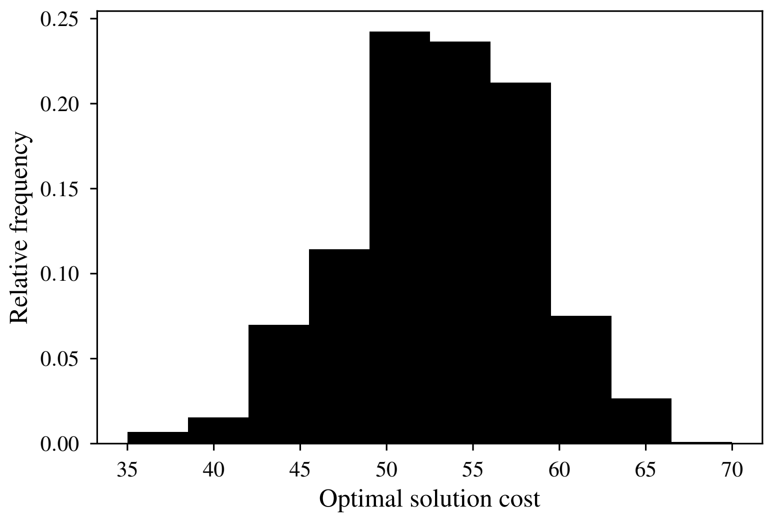


Figure 4.1: Histogram of optimal solution costs of boards in the evaluation dataset

mates of the optimal solution cost, as the optimal PDB 7-8 heuristic tends to underestimate it. According to experiments such as those performed in [3], using a higher value of W tends to result in a smaller number of nodes expanded by the search algorithm, however, weighted heuristics with a $W > 1$ are inadmissible and often lead to suboptimal solutions. This makes the weighted PDB 7-8 heuristic a good reference for the ANN-distance heuristics, which are also inadmissible.

The larger the subproblems of the pattern database heuristic are, the more memory it requires. The PDB 7-8 heuristic takes up about 4.5 GBs of memory.

4.2.2 Evaluation dataset

The evaluation dataset consists of 1172 boards, which were obtained with random permutations. The distribution of the optimal solution costs of the boards can be seen in figure 4.1. The distribution is slightly skewed to the left, with the mean of 52.8 moves.

4.2.3 Evaluation on single predictions

In the first experiment, all of the heuristics were used to produce estimates of the optimal solution costs of boards in the evaluation dataset. The distributions of the optimal solution cost predictions of selected heuristics are shown in figures 4.2 and 4.3. It can be seen that the shapes of the distributions are similar to the shape of the distribution of optimal solution costs of boards in the evaluation dataset. The biggest difference over the evaluation dataset

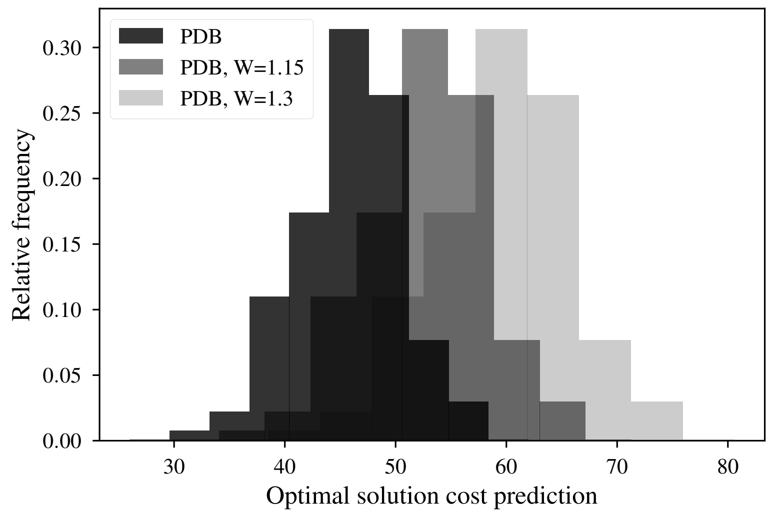


Figure 4.2: PDB heuristics: histogram of single predictions

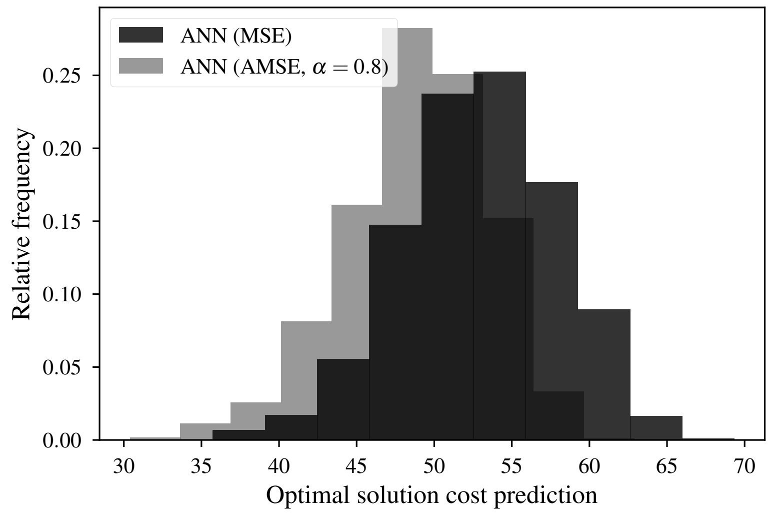


Figure 4.3: ANN-distance heuristics: histogram of single predictions

is that the distributions are shifted to more positive or negative values. The PDB 7-8 heuristic produces smaller estimates than its weighted counterparts, and the distributions of the predictions of the weighted PDB 7-8 heuristics are also spread more widely than the distribution of the optimal solution cost estimates produced by the original PDB 7-8 heuristic. Although not shown, the PDB 7-8 heuristic with the weight of 1.45 produces even higher estimates than the PDB 7-8 heuristic with the weight of 1.3. The ANN-distance heuristic produces optimal solution cost estimates whose distribution is quite similar

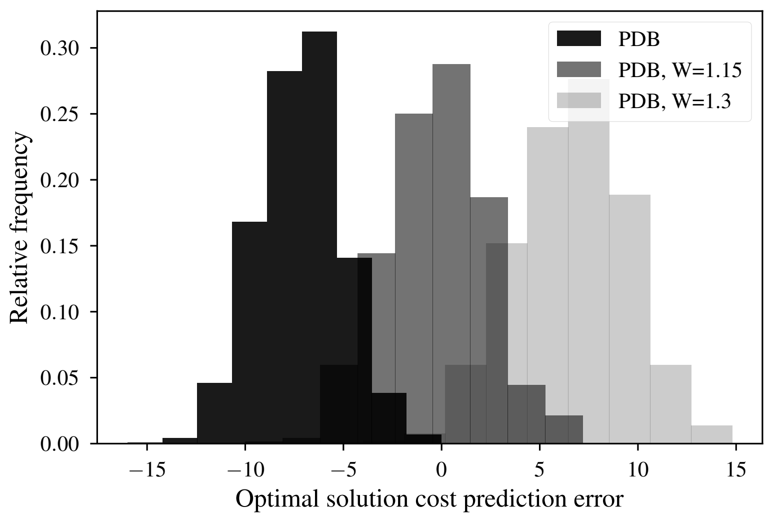


Figure 4.4: PDB heuristics: histogram of errors on single predictions

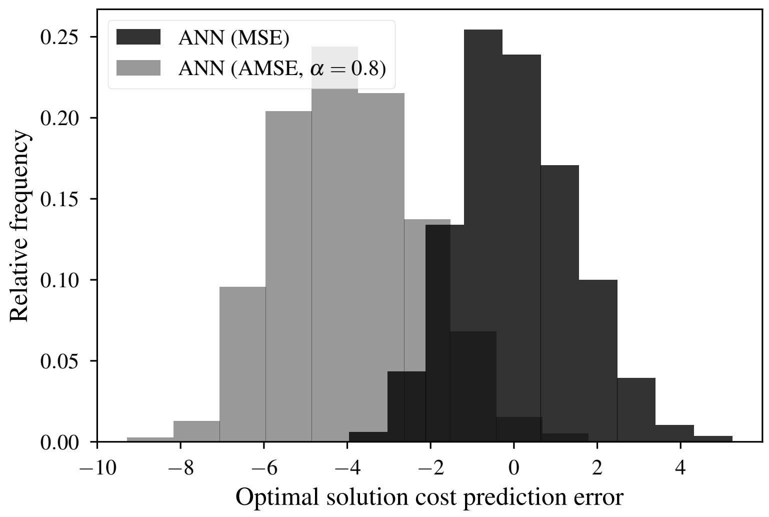


Figure 4.5: ANN-distance heuristics: histogram of errors on single predictions

to the true distribution of the optimal solution costs, with the mean of 52.9. The ANN-distance heuristic that was trained to underestimate its predictions using an asymmetric cost function with $\alpha = 0.8$ produces estimates which are somewhat lower than those produced by the original ANN-distance heuristic trained with the mean squared error cost function, with the mean of 48.9.

When looking directly at the difference of the estimates from the optimal solution costs, which is shown in figures 4.4 and 4.5, it can be seen that the PDB 7-8 heuristic with the weight of 1.15 tends to produce estimates that are roughly accurate, while the PDB 7-8 with higher weights tend to strongly overestimate the optimal solution costs. On the other hand, the unweighted PDB 7-8 heuristic tends to underestimate the optimal solution costs. The ANN-distance heuristic trained with the MSE cost function produces estimates that are accurate on average, with the mean optimal solution cost error of 0.06 and standard deviation of 1.42. However, the ANN-distance heuristic trained with the AMSE cost function with $\alpha = 0.8$ tends to strongly underestimate the optimal solution costs, with the mean error of -3.9. The ANN-distance trained with the AMSE cost function with $\alpha = 0.4$ tends to underestimate the optimal solution cost somewhat less, with the mean error of -1.4. The ANN-distance heuristic whose underlying ANN has only 1 hidden layer produced roughly accurate estimates, similarly to the deep ANN trained with the MSE cost function, with the mean error of -0.14. However, the standard deviation of the error was slightly higher in this case, with the value of 1.49.

It can be concluded that the ANN-distance heuristics trained with the MSE cost functions tend to give very accurate estimates of optimal solution costs, with the mean error of roughly 0 and the standard deviation of the error of only about 1.4. The ANN-distance heuristics trained with the AMSE cost function tend to underestimate the optimal solution costs, while those trained with the AMSE cost with a higher value of α tend to underestimate the costs more than those trained with a smaller value of α.

4.2.4 Evaluation on A* searches

In the second experiment, the heuristics were evaluated as underlying heuristics in A* searches, whose initial states were represented by boards in the evaluation dataset. One of the reasons for choosing A* over IDA* for experimentation is that A* usually does not expand as many states as the IDA* algorithm, as IDA* tends to run the low-level many times and the low-level tends to repeatedly revisit states. The most important reason for choosing A* over IDA*, however, is that in every run of the IDA* algorithm, the lower bound of the cost must first be established. However, some of the ANN-distance heuristics were not trained to underestimate the optimal solution costs, and establishing the lower bound would thus be problematic.

As in the previous experiment, the performance of the ANN-distance heuristics was evaluated against the PDB 7-8 heuristic and its weighted derivatives. No time limit was imposed on the searches, hence solutions to all boards with all heuristics were found. Even though the A* algorithm does not guarantee optimality with an inconsistent heuristic, parallel runs of the IDA* algorithm verified that A* running with the unweighted PDB 7-8 heuristic always found the optimal solutions, and the results for A* with the PDB 7-8 heuristic presented in this thesis can therefore be perceived as optimal. However, it is important to note that the performance of A* with the PDB 7-8 heuris-

Heuristic	Cost	Exp	Time	%Opt	%Avg sub	Exp/s
PDB 7-8	52.8	62,222	6.946	100	0	8958
PDB 7-8, W=1.15	53.4	3,296	0.346	71	1.19	9526
PDB 7-8, W=1.3	55.2	1,017	0.107	38	4.46	9505
PDB 7-8, W=1.45	57.5	540	0.056	25	8.86	9643
ANN	53.7	355	2.405	61	1.62	148
ANN, 1 h.l.	53.7	1,577	5.988	60	1.69	263
ANN, AMSE (α=0.4)	53.5	532	3.543	67	1.34	150
ANN, AMSE (α=0.8)	53.5	2,576	16.812	69	1.23	153

Table 4.1: Performance analysis of A* running with various heuristics

tic should not be perceived as the performance of an algorithm guaranteeing optimality.

The results can be seen in table 4.1. The meaning of the columns is as follows: the first column represents the heuristic function used with the A* searches. The second column states the average cost of solutions found by the searches. The third column shows the average number of states expanded by the search algorithm. The fourth column represents the average running time of the searches. The fifth column states what percentage of the solutions found were optimal. The sixth column represents the precentages of the average suboptimality of the solutions. Average suboptimality is defined as the average cost of the solutions, divided by the average optimal cost of the solutions (and at the end of this computation, 1 is subtracted and the result is multiplied by 100 to obtain percentages). The last column shows the number of expanded nodes per second, a value which corresponds to the speed of the inference of the heuristics. In the rows corresponding to the results, the first four rows represent the PDB 7-8 heuristic and its derivatives. The fifth row represents the deep ANN-distance heuristic trained with the MSE cost function. The sixth row represents the shallow ANN-distance heuristic with a single hidden layer, also trained with the MSE cost function. The last two rows represent the two deep ANNs trained with the AMSE cost function, with different values of α.

In the results, it can be seen that A* with the unweighted pattern database heuristic is already quite powerful, as it is able to find a solution to a random 15-puzzle instance within seconds. This is a very strong result when compared to, for example, weaker versions of the pattern database heuristic (PDB heuristics with more subproblems composed of smaller numbers of pebbles), or the Manhattan distance heuristic, which would typically take hours to find a solution to a random 15-puzzle instance when coupled with A* search. A* with the PDB 7-8 heuristic always found the optimal solution of all boards in the evaluation dataset, although it is unknown whether this is merely a coin-

Heuristic	Opt.	Opt.+2	Opt.+4	Opt.+6
PDB 7-8	1172	0	0	0
PDB 7-8, W=1.15	834	309	29	0
ANN	718	409	44	1
ANN, 1 h.l.	701	421	49	1
ANN, AMSE ($\alpha = 0.4$)	786	357	29	0
ANN, AMSE ($\alpha = 0.8$)	814	336	21	1

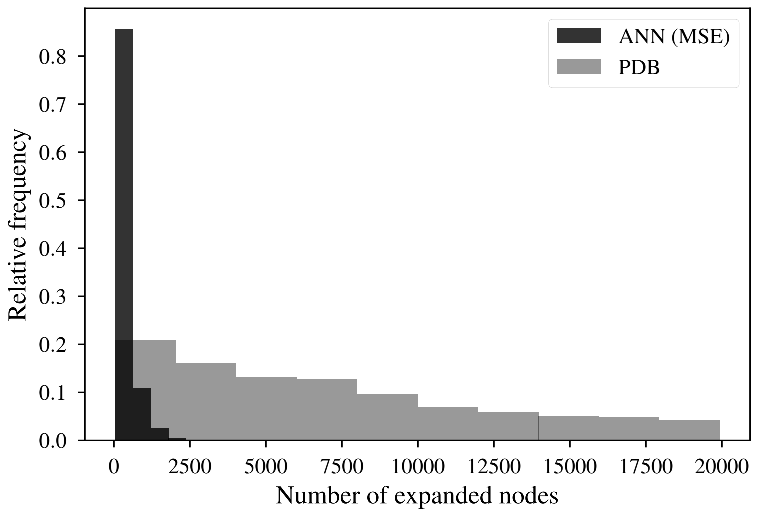
Table 4.2: Suboptimality of the solutions found by A*

cidence, or whether the PDB 7-8 heuristic is truly consistent. The algorithm expanded about 9 thousand nodes per second on average with this heuristic, which is relatively slow, and is likely the consequence of choosing the Python language for implementation.

The weighted variants of the PDB 7-8 heuristic lead to obtaining a solution much faster than with the unweighted variant. A* with the weighted heuristics expands much less states on average, with the PDB 7-8 heuristic of weight 1.45 lowering the number of expanded nodes by two orders of magnitude. However, the higher the weight, the more the suboptimality of the solutions increases, and with the weight of 1.3, solutions found are already optimal in only about 38 percent of cases.

The ANN-distance heuristic, based on an underlying deep neural network trained with the MSE cost function, decreases the number of expanded states even more, with the result of only 355 expanded nodes on average. This is a relatively low number, as the minimum number of nodes that must definitely be expanded by any search algorithm is equal to the optimal solution cost, the average of which is 52 in case of the 15-puzzle. 61 percent of the solutions found were optimal. The exact suboptimalities of all instances can be seen in table 4.2, in which each column represents a solution cost which was at a specific distance from the optimal solution cost. In the table, it can be seen that by far, most suboptimal solutions found with the ANN-distance heuristic are only suboptimal by two extra moves. This leads to average suboptimality of only 1.62 %. The major disadvantage of the ANN-distance heuristic is its inference speed, which is very slow, even though the experiment is run on a GPU. The number of states expanded per second is only about 150, which is 60 times lower than that of the PDB 7-8 heuristic. As a result, the average time required to find a solution is around 2.4 seconds, which is lower than that of the unweighted PDB 7-8 heuristic, but still much higher than the runtimes of the A* algorithm running with the weighted PDB 7-8 heuristics.

The ANN-distance heuristic with the underlying shallow artificial neural network resulted in a similar optimality of solutions as the deep ANN-distance heuristic. However, the number of expanded nodes was almost 5 times higher.



as

Figure 4.6: Histogram of number of nodes expanded during A* search (trimmed)

Another significant difference over the deep neural heuristic is that the shallow ANN-distance heuristic produced its predictions faster, which resulted in the number of expanded nodes of about 260 per second, a significant improvement over the heuristic based on a deep artificial neural network.

The use of ANN-distance heuristics trained with the asymmetric mean squared error, a cost function which was supposed to bias the heuristics towards higher admissibility, indeed lead to a slightly decreased suboptimality of the solutions found. Average suboptimality was decreased from about 1.62 %, corresponding to the unbiased deep neural heuristic, to about 1.34 % and 1.23 %, corresponding to the ANN-distance heuristics with α of 0.4 and 0.8, respectively. However, these heuristics lead to a larger number of nodes being expanded, which resulted in a higher runtime of the A* algorithm.

The roughly exponential distributions of the number of nodes expanded by the A* searches with the deep unbiased ANN-distance heuristic and the unweighted PDB 7-8 heuristic can be seen in figure 4.6.

4.2.5 Competitive comparison against heuristics presented in other studies

The ANN-distance heuristic functions presented in this thesis can be directly compared to neural heuritics presented in two earlier studies, both of which were described in chapter 2. The first paper by Ernandes and Gori from 2004 [14] created a heuristic function based on an artificial neural network, which was then evaluated using the IDA* algorithm. The second paper by Samadi et

Study	Algorithm	Heuristic	Cost	Nodes	Time
This	A*	ANN	53.7	355	2.405
This	A*	ANN, AMSE ($\alpha = 0.4$)	53.5	532	3.543
This	A*	ANN, AMSE ($\alpha = 0.8$)	53.5	2,576	16.812
[14]	IDA*	ANN	54.5	24,711	7.38
[3]	RBFS	ANN (MD+PDB)	54.3	2,241	0.001
[3]	RBFS	ANN (MD+PDB, asym. cost)	52.6	16,654	0.021

Table 4.3: Comparison of search algorithms against heuristics presented in other papers

al. from 2008 also designed a heuristic function based on an ANN, but trained it using outputs of other heuristic functions as features, namely the outputs of the PDB 7-8 heuristic and the Manhattan distance heuristic. Samadi et al. evaluated their heuristic using the RBFS search algorithm. Neither of these studies presented the performance of their network on single predictions, and the algorithms used differ, so the comparison of these heuristics is difficult. Nevertheless, the results are presented in table 4.3.

It is evident that the number of expanded nodes is significantly smaller with the unbiased ANN-distance heuristic presented in this paper than with the heuristics presented in the other papers, even by as much as two orders of magnitude. It is however likely that at least the heuristic presented by Ernandes and Gori would perform significantly better in number of expanded nodes if ran as part of the A* algorithm, as IDA*, which they used, tends to run the low-level multiple times and the low-level revisits large subtrees, and thus the IDA* algorithm expands more nodes than necessary.

As for optimality, Ernandes and Gori stated that 28.7 % of their solutions were optimal, which is worse than the roughly 60 % solutions obtained by the unbiased ANN-distance heuristic presented in this thesis. However, Samadi et al. stated that suboptimality of their biased heuristic is less than 0.1 %, which is significantly lower than the suboptimality of 1.23 % of the biased ANN-distance heuristic trained with the AMSE cost function with $\alpha = 0.8$. Unfortunately, it is not clear whether Samadi et al. used exactly the same metric of suboptimality, however the values of the average solution costs suggest that their definition of suboptimality is the same as in this thesis.

The runtime comparison is included for completeness, but is not very informative when compared over different studies, as for example Samadi et al., whose algorithm only takes a millisecond to find a solution, had likely well optimized their code.

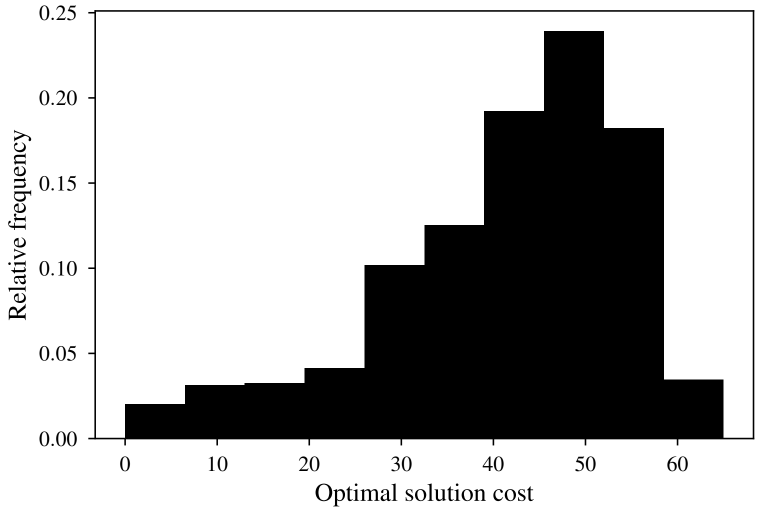


Figure 4.7: Histogram of optimal solution costs of nodes expanded during A* search with the ANN-distance heuristic trained with the MSE cost function

4.2.6 Analysis of the behavior of A* search with the underlying ANN-distance heuristic

When evaluating a heuristic function, it is important to know what parts of the state-space the algorithm spends the most time in. This analysis can be performed by calculating the optimal solution cost belonging to each expanded state. The distribution of the optimal solution costs belonging to nodes expanded by the A* algorithm can be seen in figure 4.7. The data were obtained by running an IDA* search with the PDB 7-8 heuristic to get the optimal solution cost, for each state the A* search with the deep ANN-distance heuristic (trained with the MSE cost function) expanded, for boards obtained with random permutations.

It can be seen that the distribution is much more skewed to the left than the distribution of optimal solution costs of states expanded by the A* algorithm with the underlying PDB 7-8 heuristic, which was presented in figure 3.2. This could mean that the heuristic predictions are very accurate for instances of high optimal solution costs, so the search algorithm with the ANN-distance gets faster from the initial state than with the PDB 7-8 heuristic. However, this is merely a hypothesis and it is possible that a different explanation is true.

Discussion

Experimental evaluation of the ANN-distance heuristics performed in the previous chapter shows that the ANN-distance heuristics are highly powerful and competitive, even when compared to other state-of-the-art admissible and inadmissible heuristic functions.

Supporting evidence for all of the untested hypotheses presented in chapter 3 has been found:

- The hypothesis that the ANN-distance heuristic will lead to comparable or even more accurate estimates of the optimal solution cost than the PDB 7-8 heuristic: ANN-distance has a mean prediction error of about zero, and a lower variance of the error than the PDB 7-8 heuristic, which tends to strongly underestimate the optimal solution cost.

- The hypothesis that using a more accurate inadmissible heuristic function will generally lead to a lower number of nodes expanded by the search algorithm: using a more accurate inadmissible heuristic function truly seems to lower the number of nodes expanded by a search algorithm. The most accurate heuristic, the deep ANN-distance heuristic trained with the mean squared error cost function, expanded a smaller number of nodes on average than all other heuristic functions based on ANN-distance.

- The hypothesis that using a heuristic function based on a deep artificial neural network will result in a better performance than using a shallow artificial neural network with the same number of hidden neurons: experiments show that a deep artificial neural network is able to achieve a lower validation error during training and performs significantly better when coupled with the A* algorithm than a shallow artificial neural network with the same number of hidden neurons.

An interesting result is that using the ANN-distance heuristic biased towards admissibility still often resulted in obtaining suboptimal solutions, even though the biased ANN produced predictions which almost always underestimated the optimal solution cost. It is possible that most of this suboptimality was caused by the properties of the graph version of the A* algorithm, which guarantees optimality of the solutions only when the heuristic function used is consistent. As the ANN-distance is inconsistent, it would be interesting to try the heuristic with a different search algorithm, specifically one that guarantees optimality of solutions even when the underlying heuristic is inconsistent. If the obtained solutions were still often suboptimal, it would lead to the conclusion that even only occasional overestimations of a heuristic often lead to suboptimal solutions.

When compared to the results of other papers, the ANN-distance heuristic turns out to be more powerful than all similar heuristics presented in previous research. The ANN-distance heuristic trained with the mean squared error cost function expands by two orders of magnitude less nodes than the most powerful heuristic in [14], and by an order of magnitude less nodes than the most powerful heuristic in [3]. However, the search algorithms utilized in the experiments performed in these studies differ, and the accuracy of the comparison is hence limited. The previous papers unfortunately did not analyze the single predictions of their ANNs. The optimality of the solutions obtained with the ANN-distance heuristics is comparable to the optimality of solutions presented in other papers.

If the ANN-distance heuristic truly is significantly more powerful than those presented in other papers, it is unclear what exactly caused the ANN-distance heuristic presented in this paper to be so precise. It is possible that a good choice of a training set helped. However, it is more likely that the biggest enhancement was the significant increase of the size of the ANN and of the training set. Also, utilizing a deep artificial neural network helped, as experiments confirmed.

A big advantage of the ANN-distance heuristic is its small memory requirements. An underlying ANN in the ANN-distance heuristic consumes only about 20 MBs of RAM, which is much smaller than around 4.5 GBs of RAM necessary for the PDB 7-8 heuristic. This would make it possible to use the ANN-distance heuristic even in cheap embedded systems.

Another possible advantage of the ANN-distance heuristic is that it could be used in other problem domains similar to the (N^2-1)-puzzle. A hypothesis here is that it could be possible to use the ANN-distance heuristic even in problem domains which are “opaque”, that is, problem domains whose inner structure is not understood and therefore other custom heuristic functions can not be created for these problem domains. This is merely a hypothesis, however, as it is unknown whether an ANN-distance heuristic could be trained on these kinds of problem domains.

The method of using artificial neural networks for solving the (N^2-1)-puzzle used in this thesis also has its limitations. It likely could not currently be applied to the 24-puzzle in the same manner, as it would be computationally difficult to obtain a sufficient number of training instances with corresponding optimal solution costs. It seems that more advanced methods, such as the one presented in [24], would have to be used for instances of the (N^2-1)-puzzle with a higher N.

Another disadvantage of the ANN-distance heuristic is that it takes time to gather the training dataset and train the underlying artificial neural network. In experiments performed in this paper, the time to generate the training dataset and train the ANN-distance heuristic was comparable to the time taken to calculate the databases of the PDB 7-8 heuristic.

Conclusion

This thesis focused on the problem of applying artificial neural networks in solving the (N^2-1)-puzzle. Both goals of the thesis have been fulfilled: in the first part of the thesis, the reader was introduced to the current state-of-the-art methods for solving the (N^2-1)-puzzle, and was presented the history of the existing applications of artificial neural networks in near-optimal solving of the (N^2-1)-puzzle. In the second part of the thesis, a heuristic function based on an artificial neural network was introduced, and was experimentally evaluated against other powerful heuristic functions.

The new heuristic function turned out to be very effective when used together with the A* algorithm — it helped to quickly find solutions which were mostly optimal, while expanding a low number of nodes. The number of expanded nodes was improved over the best existing heuristic functions based on artificial neural networks by an order of magnitude. The new heuristic also takes up much less memory than a comparable pattern database heuristic.

The thesis confirmed the hypothesis that the philosophy of deep learning can be successfully applied to the problem domain of the (N^2-1)-puzzle, as the first such experiments in history showed that a deep neural network can perform better than a shallow neural network. It was also shown that a heuristic function based on an artificial neural network can be trained to prefer underestimating its predictions, which results in reaching an optimal solution more often by the search algorithm.

There is still a lot of room for more research to be performed. Training larger and deeper artificial neural networks, possibly on much larger datasets, would likely create more accurate heuristics that would lead to fewer nodes being expanded by the search algorithms (at the expense of higher time per prediction). Also, the distribution of the training dataset could be experimented with, in order to obtain the most effective distribution of optimal solution costs. Other informed search algorithms employing the new heuristic function could also be tested, particularly algorithms that do not require a consistent heuristic function to guarantee optimality. And it would be possible to move from the domain of the 15-puzzle to the more challenging domain of the 24-puzzle. Especially, transfer learning could be used, where the ANN trained on the 15-puzzle could be used as a basis for an ANN designed for solving random 24-puzzle instances.

Bibliography

[1] Ratner, D.; Warrnuth, M. Finding a Shortest Solution for the NxN Extension of the 15-Puzzle is Intractable. In *Proceedings of AAAI-86*, 1986, pp. 168–172.

[2] Korf, R. E.; Taylor, L. A. Finding Optimal Solutions to the Twenty-Four Puzzle. In *Proceedings of AAAI-96*, MIT Press, 1996, pp. 1202–1207.

[3] Samadi, M.; Felner, A.; et al. Learning from multiple heuristics. In *Proceedings of AAAI-08*, 2008, pp. 357–362.

[4] Slocum, J.; Sonneveld, D. *The 15 Puzzle Book: How it Drove the World Crazy*. Slocum Puzzle Foundation, 2006, ISBN 1890980153.

[5] Korf, R. E. Sliding-tile puzzles and Rubik's Cube in AI research. In *IEEE Intelligent Systems*, 1999, pp. 8–12.

[6] Parberry, I. A Memory-Efficient Method for Fast Computation of Short 15-Puzzle Solutions. *IEEE Transactions on Computational Intelligence and AI in Games*, volume 7, no. 2, June 2014: pp. 200–203.

[7] Cahlík, V.; Surynek, P. On the Design of a Heuristic based on Artificial Neural Networks for the Near Optimal Solving of the (N2–1)-puzzle. In *Proceedings of the 11th International Joint Conference on Computational Intelligence - Volume 1: NCTA, (IJCCI 2019)*, 2019, pp. 473–478.

[8] Ryan, M. Tiles Game. 2004, online. Accessed: 2019-12-30. Available from: `https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html`

[9] Parberry, I. A real-time algorithm for the $(n^2 - 1)$-puzzle. *Information Processing Letters*, volume 56, 1995: pp. 23–28.

[10] Surynek, P.; Michalík, P. The joint movement of pebbles in solving the $(N^2 - 1)$-puzzle suboptimally and its applications in rule-based cooperative path-finding. *Autonomous Agents and Multi-Agent Systems*, 09 2016, doi:10.1007/s10458-016-9343-7.

[11] Sharon, G.; Stern, R.; et al. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, volume 219, 02 2015: pp. 40–66, doi:10.1016/j.artint.2014.11.006.

[12] Sharon, G.; Stern, R.; et al. The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, volume 195, 01 2011, pp. 662–667, doi:10.1016/j.artint.2012.11.006.

[13] Russell, S.; Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, third edition, 2009, ISBN 0136042597, 9780136042594.

[14] Ernandes, M.; Gori, M. Likely-Admissible and Sub-Symbolic Heuristics. 01 2004, pp. 613–617.

[15] Holte, R. Common Misconceptions Concerning Heuristic Search. 09 2010.

[16] Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, volume 27, no. 1, 1985: pp. 97 – 109, ISSN 0004-3702, doi:https://doi.org/10.1016/0004-3702(85)90084-0.

[17] Korf, R. E.; Felner, A. Disjoint pattern database heuristics. *Artif. Intell.*, volume 134, 2002: pp. 9–22.

[18] Howard, J. Fast.ai Massive Open Online Course, Lecture 1. 2018, online. Accessed: 2019-12-14. Available from: `https://www.youtube.com/watch?v=IPBSB1HLNLo`

[19] Rumelhart, D. E.; Hinton, G. E.; et al. Learning representations by back-propagating errors. *Nature*, volume 323, no. 6088, Oct 1986: pp. 533–536, doi:10.1038/323533a0.

[20] Geron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., first edition, 2017, ISBN 1491962291, 9781491962299.

[21] Liang, S.; Srikant, R. Why Deep Neural Networks? *CoRR*, volume abs/1610.04161, 2016.

[22] Ba, L. J.; Caruana, R. Do Deep Nets Really Need to be Deep? *CoRR*, volume abs/1312.6184, 2013.

[23] Hou, G.; Zhang, J.; et al. Mixture of experts of ANN and KNN on the problem of puzzle 8. Technical report, Technical report, Computing Science Department University of Alberta, 2002.

[24] Arfaee, S. J.; Zilles, S.; et al. Learning heuristic functions for large state spaces. *Artificial Intelligence*, volume 175, no. 16, 2011: pp. 2075 – 2098, ISSN 0004-3702, doi:https://doi.org/10.1016/j.artint.2011.08.001.

Appendix A

Acronyms

AMSE	Asymmetric Mean Squared Error
ANN	Artificial Neural Network
CBS	Conflict-Based Search
CPF	Cooperative Path Finding
BFS	Breadth-First Search
DFS	Depth-First Search
FIFO	First in, First out
ICT	Increasing Cost Tree
ICTS	Increasing Cost Tree Search
IDA*	Iterative Deepening A*
k-NN	k-Nearest Neighbors
LIFO	Last in, First out
MAPF	Multi-Agent Path Finding
MLP	Multilayer Perceptron
MSE	Mean Squared Error
PDB	Pattern Database

Appendix **B**

Contents of Attached CD

`README.txt`	file with CD contents description
`fifteen-puzzle`	directory with the source files and data
`thesis.pdf`	the thesis text in PDF format