MSc Thesis
Utrecht University
Artificial Intelligence

A data-driven approach for making a quick evaluation function for Amazons

Author:
Michel Fugers
Supervisor and first examiner:
Dr. Gerard Vreeswijk
Second examiner:
Dr. Frank Dignum

Utrecht University

May 2017

Contents

1 Introduction **2**

2 The game **5**

2.1 Origin and rules 5

2.2 Game mechanics 7

2.3 Generalizations 10

3 Computer Amazons **11**

3.1 Search strategies 11

3.2 Evaluation functions 16

3.3 Skipping search 19

3.4 Comparing computer players 20

4 Preliminary study on Tic-tac-toe **23**

4.1 Method 24

4.2 Results 28

5 Study on Amazons **31**

5.1 Method 31

5.2 Results 37

6 Discussion **44**

6.1 Complexity 44

6.2 Representation 45

6.3 Topology 45

7 Conclusion **47**

References **48**

1 Introduction

This thesis is about computers playing board games. More specifically it is about computers playing the game *Amazons*. Amazons is a two player game, much like Draughts, Chess or Go. Computers playing games is not new. In 1994, a computer called Deep Blue by IBM beat the at the time reigning world champion Garry Kasparov at Chess. More recently, in 2016, for the first time a computer was able to beat a professional human Go player in an even game. This illustrates that the game of Go is a lot harder than Chess, at least from an artificial intelligence point of view. Amazons is a game that is more complex than Chess, but not as complex as Go. That makes it a natural test bed for computer game research.

Traditionally in computer Amazons research an algorithm called alpha-beta pruning, or a variation thereof, is used to search for the best move to play. Kloetzer et al. (2007) introduced Monte Carlo tree search as an alternative search algorithm. Monte Carlo tree search works by simulating many random progressions of the game. Lorentz (2008) refined this technique and proved that it can be applied in a computer player that is able to play at tournament level.

Monte Carlo tree search performs better when more random games are considered. The more simulations, the better. Usually the time to act is limited, so that means: the faster a simulation is calculated, the more simulations can be done, the better. One method of speeding up is not to simulate the game until the end is reached, but rather to stop at a certain depth. The board is then evaluated by an evaluation function. Naively one might think that the best evaluation function yields the best results. In combination with Monte Carlo tree search, this is not necessarily true. More advanced evaluation functions typically use more time than simpler evaluation functions, which leads to fewer simulations and thus poorer results. There is an interesting balance to strike: the evaluation function should be reliable, yet not take much time to compute.

One measure Lorentz (2008) takes to speed up the simulations is using a simpler evaluation function rather than a complex one. With the traditional alpha-beta pruning based algorithm, the advanced evaluation function performs better. With Monte Carlo tree search the simpler function performs better.

In this thesis artificial neural networks are trained in order to act as an evaluation function. Neural networks may take long to train, but once trained they are able to quickly map inputs to outputs. Even if the quality of the

evaluation is less then traditional evaluation functions, they may increase the performance of Monte Carlo tree search because they allow for many more simulations.

Experiments show that it is possible to train neural networks to evaluate combinatorial games using a database of played games. The constructed neural network is able to evaluate a simple game like Tic-tac-toe reliably. The proposed technique does not scale well to the more complex game of Amazons.

Section 2 sets out the rules of the game of Amazons. It also discusses some relevant terms and concepts. Readers who are already familiar with the game can safely skip this section.

Section 3 provides an overview of Amazons as a research field. In it, also a way to effectively use Monte Carlo tree search with neural network based evaluation functions is proposed. Basic knowledge of the game is assumed, so this section is best read after the previous one. This section describes and substantiates the central question of the thesis: can a database of example games in combination with machine learning allow for the construction of a quick and reliable evaluation function for Amazons?

Section 4 describes a research paper on neural network based evaluation functions that are applied to other games. It also presents the methods and results of two experiments on Tic-tac-toe. Reading this section does not require any knowledge or understanding of Amazons, and can be read independently of the previous sections.

Section 5 discusses a method of answering the central question that was introduced in Section 3, using of the techniques that are proven effective in Section 4. As this section discusses research on neural network based evaluation functions on Amazons, it builds upon the information provided on all previous sections.

Section 6 discusses the extent of the success of the proposed method and provides suggestions for possible further research. This section refers back to the research described in Section 5.

Section 7 concludes with a brief summary of the preceding sections.

A dependency graph of sections in this thesis is shown in Figure 1.

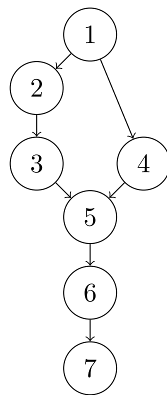


Figure 1: Section dependency graph

2 The game

Amazons is a board game for two players. This section will cover the origin and rules of the game. Anyone who is already familiar with the game and its rules, can safely skip this section, and continue reading the next section.

2.1 Origin and rules

In 1988, Argentine game designer Walter Zamkauskas invented the game. Four years later he published its rules in a puzzle magazine under the name *El Juego de las Amazonas* (translated: *The Game of the Amazons*). This name is still a trademark of the Argentine puzzle magazine publisher *Ediciones de Mente* (Keller, 2009). Michael Keller translated the original Spanish article to English. In practice the game is simply referred to as 'Amazons', rather than 'The Game of the Amazons'. It gained particular interest in the academic world because of its interesting properties for artificial intelligence research, because it is more complex than Chess, but not as complex as Go.

Story

The game tells the story of two tribes of amazons that battle for the largest territory. By shooting burning arrows they can burn down certain parts of the land to form impassable barricades. Both tribes try to imprison the members of the opposing team by burning down all land around them, while at the same time securing the largest territory for themselves without getting imprisoned by their opponents.

Materials and setup

It is not common even for specialized game stores to sell complete Amazons sets, but the game can be bought online (Kadon Enterprises, Inc., 2014). It is easy to compile your own Amazons set using parts of other games. The game requires these materials:

- One 10×10 square-tiled board, e.g. a Draughts board
- Four black and four white pawns, e.g. eight Chess queens or pawns, called *amazons*

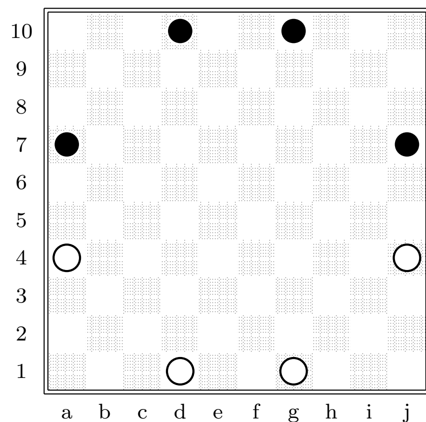


Figure 2: Initial state in Amazons

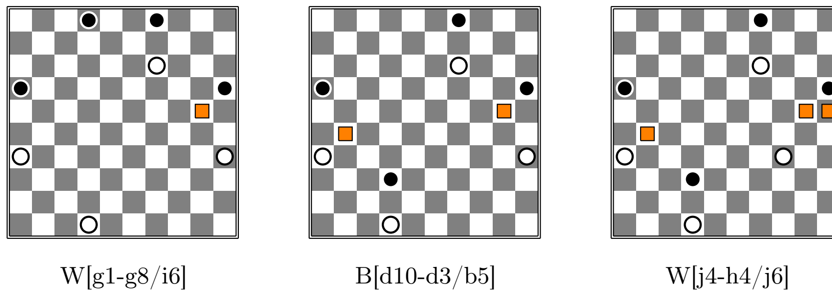


Figure 3: A common sequence of plies at the start of the game

- 92 tokens[1], e.g. Go stones

The initial state of the amazons is shown in Figure 2. White is first to play.

Turn sequence

Each *ply* (a turn of a single player) consists basically of two actions: move an amazon, and shoot an arrow. Firstly, the player selects one amazon and moves it one or more fields horizontally, vertically or diagonally, without crossing another amazon or a barricade (much like a queen in Chess). Secondly, the selected amazon must shoot to a field one or more fields horizontally, vertically or diagonally from its new position, again without crossing an amazon or a

[1]Hardly ever all tokens are used. Usually when about 40 tokens are on the board, it is clear who will win the game. When no player resigns and the game is played until a terminal game state, up to 92 tokens may be necessary. Together with the 8 amazons they then occupy all 100 fields of the board.

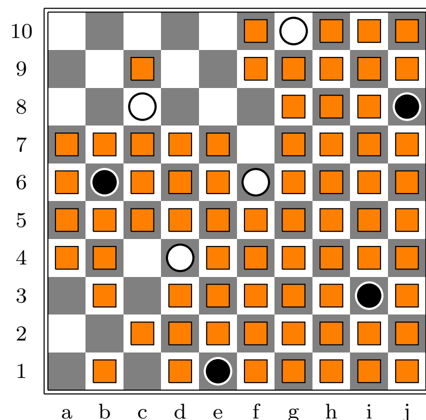


Figure 4: One possible terminal state, White wins

barricade. For the rest of the game this field will be a barricade, which is marked by a token. Some authors refer to barricades as *burned square* (Berlekamp, 2000), *burn-off squares* (Muller & Tegos, 2002) or simply *arrows* (Tegos, 2002). Figure 3 depicts the first few plies of a game.

Termination

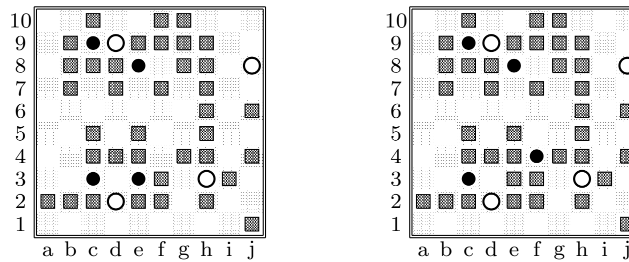
When a player is unable to play, i.e. when all their amazons are unable to move, the other player wins the game. For example, Figure 4 shows a situation in which Black is unable to move, therefore White wins. It is not possible to have a draw in Amazons.

Combinatorial game

Similar to Chess and Go, Amazons is a *discrete*, *deterministic* game with *perfect information*; the board is divided in 100 discrete fields, no dice are involved and both players can observe all aspects of the game. It also is ensured to end (games cannot last longer than 92 plies), and has no draws. Because of these properties, Amazons is considered a *combinatorial game* (Nowakowski, 1998).

2.2 Game mechanics

The previous section describes the sec rules, and although this is enough to be able to play a valid game, the introduced terminology might run short to describe all tactical aspects of the game. Therefore some important terms and



(a) All black amazons are blockers

(b) B[e3-f4/e3]

Figure 5: By shooting back at its original field the black amazon can move without allowing White to access the central territory

concepts are introduced in this section.

Territory

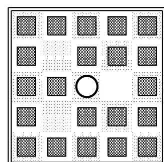
Since barricades are permanent and amazons cannot cross them, they can divide the board into separate areas. If such a separate part contains only black or white amazons, the area is called *territory of Black* or *White* respectively. If amazons of both colors are inside such a part, it is called an *active area* (Müller, 2001). At first the whole board is basically one big active area. If no amazons are present inside such a part, the area is called *dead*.

Blocker

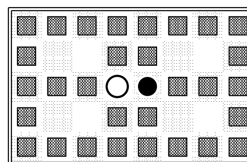
Amazons are, like barricades, not able to be passed by both other amazons and arrows. This means that amazons too can function to form distinct separate areas, and thus territories. Such amazons are called *blockers* (Muller & Tegos, 2002) or *guards* (Lieberum, 2005). Territories made by blockers are not final, because the amazons can move away from their position. Each blocker can make their territories final by moving into the area, and shooting back at their original positions. Figure 5 shows an blocker that finalizes its territory.

Endgame

Once all active areas are divided into Black's and White's territories, players cannot influence each other anymore. This phase is called the *endgame*. In it players can only move in their own territories, and shoot an arrow to make it smaller.



(a) Defective territory



(b) Zugzwang

Figure 6: Examples of defective territory and zugzwang

One endgame tactic is moving the amazon one field at a time, while shooting at its original position. This method of filling the territory is called *plodding.*

In practice, the player with the smallest territory often resigns at the start of the endgame, and the player with the largest territory wins.

Defective territory

Plodding does not guarantee optimal use of the available area. The player needs to try not to cut off the amazon from a part of the territory. Sometimes this is not possible, and the player has to sacrifice a part of their territory, to be able to fill another part. Such an area is called *defective territory*, because the size of the territory does not equal the amount of plies it provides before the amazon is unable to move. Figure 6a shows an example of a defective territory. The depicted amazon can only use one of its three adjacent free cells.

Solving an Amazons endgame, that is, finding a ply sequence of maximum length for a territory is a problem that is *NP-complete* (Buro, 2001).

Zugzwang

In some situations it would be best if the other player plays first. For example, in Figure 6b, if Black plays first, White would end up having a larger territory, and if White plays first, Black would. Since players are not allowed to pass their turn, the player that is first to play has to make a move that is unfavorable to not making that move. This is called *zugzwang*[2]. Sometimes, zugzwang can be avoided by playing another amazon elsewhere on the board.

[2]The German word *Zugzwang* means *compulsion to move.*

Mobility

The *mobility of an amazon* is the number of fields the amazon can move to. In the initial state (Figure 2) the mobility per amazon is 20. The *mobility of a player* is the sum of the mobilities of the player's four amazons. The total mobility of the losing player at the end of the game is 0 by definition.

2.3 Generalizations

There are a couple of ways the game of Amazons can be generalized. That is, some of the rules can be a bit more loosely interpreted, resulting in essentially different games, while maintaining the mechanics of the original game. Researching a generalized version of Amazons can be useful to reduce the complexity and make it easier to test or prove propositions about the game in a simpler setting.

Amazons can be generalized by varying the size of board and the number of amazons per player. Berlekamp (2000) analyzed all boards of size $2 \times n$ with 1 amazon per player, and Song & Muller (2015) solved Amazons for different board sizes, with 4 amazons per player. They proved that with *perfect play*, games on board sizes 4×5, 5×4, 4×6, 5×6 and 4×7 are sure wins for White, and games on a board of 6×4 are sure wins for Black.

When the board is split up into multiple areas, these areas can be seen as multiple, independent Amazons games, with smaller board sizes and possibly fewer amazons. Combinatorial game theory would regard the whole game as a sum of multiple smaller games (Berlekamp, 2000).

Creating a computer player that is able to generalize its knowledge of Amazons in order to be able to perform well with any board size would be a interesting and ambitious feat, this is, however, outside the scope of the thesis. In this thesis Amazons is assumed to be played with the default rules on a 10×10 board and four amazons per player.

3 Computer Amazons

Michael Keller (see Section 2.1) made the first, albeit weak, computer Amazons program in 1994. Many others, amongst whom many artificial intelligence researchers, have made other and stronger computer Amazons programs since. The *International Computer Games Association* holds annual *Computer Olympiads* for various games including Amazons. The winner of Computer Olympiad 2016 (ICGA, 2016) is a program called *Invader*. Other strong programs are for example *8QP*, and *Amazong*.

Because this thesis is part of the scientific discourse concerning computer Amazons, this section is devoted to give a broad overview of this field of research. The main points of interest are *search strategies* and *evaluation functions*. These are discussed in Sections 3.1 and 3.2 respectively. Section 3.3 is about other ways to optimize play, apart from search strategies and evaluation functions. Assessing the quality of play amongst different computer players is elaborated upon in Section 3.4.

3.1 Search strategies

Computer game algorithms are basically search algorithms. They search the best ply amongst all possible plies. In the case of Amazons, during the opening there can be thousands of possible plies in which case it is very hard to find the best one. Conversely in endgames there may be just one possible ply in which case it is trivial to find the best one.

The way an algorithm finds the best ply is typically by building a *game tree*. A game tree is a graph representing all possible progressions from a certain initial game position. Each node in the graph is a configuration of the board. The initial game position is the root node of the tree. A node has child nodes for all possible plies possible from that game position. In a full game tree, the *leaf nodes* (nodes with no children) are the terminal game positions.

A *partial game tree* is a graph with a subset of the nodes of a full game tree. Partial game trees may only include nodes up to a certain depth, or not all possible plies may be included as child nodes. In a partial game tree the leaf nodes are not necessarily terminal game positions.

In each ply a barricade is added to the board and barricades are never removed, so it is impossible to have cycles in the game tree. It may be possible to end up in a certain position via multiple different paths. Thus formally

game trees can be seen as directed acyclic graphs, but since these transpositions happen very rarely they are treated as proper trees throughout this thesis (Karapetyan & Lorentz, 2004).

Ideally one would calculate the full game tree: generate all plies, and reactions and reactions-to-reactions, until each branch of the tree ends in a terminal game state wherein one of the players wins. As discussed earlier these game trees are astronomically huge. Computing the whole tree is obviously not feasible. Only in endgames, when only few plies are left to play, the full game tree might be computable. For this reason all algorithms build only partial game trees, including only subsets of all possible plies and counter plies.

The exact method of building the tree is different for each algorithm, each with its unique advantages and drawbacks. All depend on 'a rough sense' of what game states are preferable and what game states are not. This 'sense' is acquired by using an evaluation function. Evaluation functions are discussed in detail in Section 3.2.

Some well known search algorithms that use these evaluation functions are *minimax*, *alpha-beta pruning*, and *Monte Carlo tree search*.

Minimax

Minimax assigns values to game states within the game tree. It assumes one player to always pick the ply with the highest value, and the other player to pick the ply with the lowest value. The maximizing player is called 'White' and the minimizing player is called 'Black'. The values are assigned by using a depth first tree traversal. The value of a terminal node is dependent of who won; if White won the value is positive infinity, if Black won, the value is negative infinity. How the value of the parent nodes are calculated is dependent of whose turn it is. If it is White's turn, it is the maximum of the child nodes, if it is Black's turn, it is the minimum of the child nodes.

For many games it is often not feasible to calculate the full game tree all the way to the end of the game. Minimax can still be used though, by using partial game trees. The leaf nodes will not represent terminal game states, so the corresponding game values have to be estimated. These estimates are based on heuristics and are calculated by so called evaluation functions.

It is a non-trivial task to know the ideal maximum depth of the partial game tree. Running the minimax on a tree that is too big may take more time than

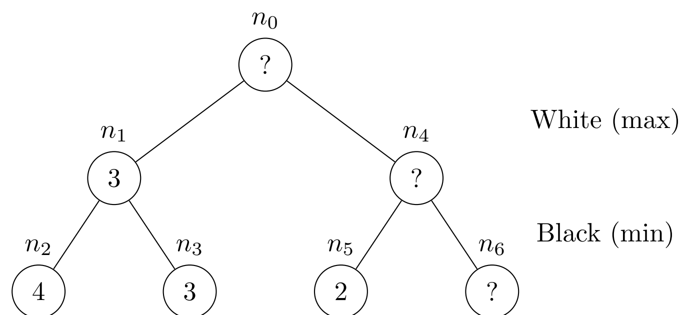


Figure 7: An example of a partially explored game tree with 7 nodes. White always selects the ply with the highest game value, Black always selects the ply with the lowest game value. The values for the leaf nodes are provided by a hypothetical evaluation function.

is available, and running the algorithm on a tree that is too small yields less precise results. A method of finding the best depth is called *iterative deepening*. With iterative deepening the algorithm is run multiple times, each time on a tree with one extra level of depth. This method has an overhead because the values for nodes at lower depths are calculated multiple times. However, this overhead is relatively small, since in each iteration the number of new nodes (leaf nodes) is much higher than the number of already encountered nodes (internal nodes).

Alpha-beta pruning

Alpha-beta pruning is an optimization of the simple minimax algorithm. It prevents the exploration of branches that will not influence the end result. Consider for example the partially explored game tree in Figure 7.

In order for minimax to calculate n_0, it first needs to evaluate n_6: the value of n_0 is the maximum of n_1 and n_4 and n_4 is in turn the minimum of n_5 and n_6. All variables except n_6 are available, so it needs to evaluate that node in order to calculate n_0. However, calculating n_6 is in fact not necessary.

Suppose n_6 has a very high value, say 99. The value of n_4 is decided by Black, and Black chooses the minimum of n_5 and n_6, so n_4 is 2. The value of n_0 is decided by White, and White chooses the maximum of 3 and 2, so n_0 is 3.

Now suppose n_6 is a very low value, say -99. Black chooses this low value, so n_4 is -99. White chooses the maximum of 3 and -99, so n_0 is 3.

Whatever the value of n_6, it will not influence the value of n_0. This means that exploring this node can be skipped. This is the essence of the alpha-beta

pruning algorithm. In large game trees it may prune significant sections from the tree, yielding a faster processing time. In the worst case, no sections can be cut, and alpha-beta pruning is equally fast compared to minimax.

To be able to tell which branches to prune and which to explore, the algorithm defines two variables α and β. The first variable is initially assigned negative infinity and represents the maximum value that White is assured to achieve. The second variable is initially assigned positive infinity and represents the minimum value that Black is assured to achieve. Whenever β is equal or less then α it means that the corresponding node will never be selected by either player, and exploring of that branch can be skipped.

Like minimax, alpha-beta pruning can use an evaluation function and iterative deepening to find the optimal depth of search in a partial game tree, and is thus very well suited for computer programs that play games such as Amazons.

Monte Carlo tree search

The *Monte Carlo method* is a way of getting information by aggregating over many random samples. Although the method was originally applied in physics, it has been successfully applied in many fields, one of which being artificial intelligence.

A simple Monte Carlo search algorithm for Amazons would generate the list of possible plies, pick those as a starting point and start taking random samples. A random sample would be playing randomly selected plies until one of the players wins. The ratio of winning samples versus losing samples then determines which ply is most likely to win.

Note that 'most likely' assumes that both players play totally randomly which is off course not true. So if one possible ply results in a game state where most of the responses will lead to a win, while also offering a single response in which the opponent wins immediately, its ratio will indicate that it is a favorable situation. However, once that ply is played, the opponent will of course only ply their winning ply. The primary assumption of the Monte Carlo method therefore is too simplistic for playing games.

Monte Carlo tree search is an extension to the simple Monte Carlo method. Instead of simply keeping track of a list of plies, the algorithm constructs a game tree. In each iteration a leaf node is selected, and either a simulation is executed or the tree is expanded. A simulation contributes to the ratio of all

ancestor nodes of the selected leaf. When the tree expands, all the children of the leaf node are generated and added to the game tree. A node gets expanded upon after it has been visited a set number of times. This way, promising plies can prove that there will be no catastrophic responses available to the opponent, because if there are, their win/loss ratio will drop, making the ply less favorable.

A simulation needs to be performed rather quickly. That is why there is no evaluating or decision making involved, it is just a series of random plies until one player wins. Generating all possible plies is still not free though. Especially in the the early phase of the game, the cost adds up. This results in fewer simulations that can be done, and in poorer performance. One method to counter that is simulating up to a certain depth and invoking an evaluation function on the resulting endgame.

This evaluation function is invoked very often. For each non-expanding visit where the simulation does not end in a terminal game state the function is invoked. This means that the speed of the evaluation function is extra important for Monte Carlo tree search. It needs to be quicker than a full random game.

There are a couple of ways proposed for selecting the node to simulate or expand. One of these is called *Upper Confidence Bound applied to trees* (UCT), which combines two measures. The first is the win/loss ratio. The second is a measure of how often that node is considered. The first is used to promote exploitation, proving that the assumed 'good' plies are actually good. The second is used to promote exploration by considering yet neglected nodes. The balance between these two measures determines how the algorithm grows the game tree, either by deepening of widening.

In 1993, Monte Carlo tree search algorithms were first applied in computer Go programs (Brügmann, 1993). This allowed for much better performance compared to the more traditional alpha-beta pruning. In 2016, the Monte Carlo tree search based program AlphaGo won from 9-dan professional Go player Lee Sedol in a full 19×19 Go match without handicaps (Silver et al., 2016).

In 2007, a French/Japanese team made the first Monte Carlo tree search based Amazons program, named Campya, albeit with poor results (Kloetzer et al., 2007). One year later, award winning alpha-beta pruning based program Invader was adapted to use Monte Carlo tree search, with surprisingly good results (Lorentz, 2008). This new *InvaderMC* was able to beat Invader in 80 % of the matches.

No lookahead

A computer player can also play without an intricate search algorithm. It may apply an evaluation function to all available plies, and pick the one that scores best. Although this may be the quickest search algorithm, computer players with no lookahead are not expected to play well.

3.2 Evaluation functions

Most search algorithms described in the previous section require some kind of evaluation function, but what are evaluation functions exactly? Evaluation functions are functions that map game states to real numbers. In case of Amazons, game states are equivalent to board configurations and the resulting number is referred to as the *evaluation value*. Evaluation values highly resemble *game theoretical values*: negative numbers denote an advantage for Black, positive numbers denote an advantage for White[3]. The purpose of evaluation functions is to compare and rank game states based on their usefulness. The lower the value, the better for Black, the higher, the better for White.

Game theoretical values may be the basis for the evaluation value (Snatzke, 2002). However, it is not feasible to compute them for states other than late endgames. That is why in practice either approximations of the game theoretical values or entirely different measures are used.

The results of different evaluation functions generally cannot be compared one-to-one. For example, one evaluation function f can map to the interval $[-1, 1]$, while another evaluation function g can map to the interval $[-92, 92]$. The value -1 will mean a certain win for Black in f, but only a very slight advantage for Black in g. This is not a problem because the functions are merely meant to compare and rank game states. As long as one compares outcomes of a single function, this is not a problem.

Evaluation functions are not perfect. If evaluation functions were perfect, a player could apply no lookahead and still be able to select the best ply: simply calculate the values for the game states resulting from all possible plies, and pick the ply that yields the best value. In practice evaluation functions are far from perfect, but can still help to decide which plies are promising, and which plies are not. In other words they guide the search, and are essential for practically

[3]Some use positive numbers to indicate advantageous for the current player, be it either White or Black, for example Kloetzer et al. (2007).

all search strategies.

In most search algorithms it is common to call the evaluation function very often. Because of this, the evaluation function needs to be able to run quickly. This is especially important for the Monte Carlo tree search algorithm.

Typical evaluation functions do not build their own game trees in order to evaluate a game state. In most cases this would be redundant since search algorithms already tend to build (partial) game trees. Moreover, building a game tree is usually a lot of work, and making a new tree for each call would take far too much time for the evaluation function to run in a timely fashion.

What *do* evaluation functions use to calculate the value? That depends on the function because each evaluation function has it own heuristics. Some evaluation functions for Amazons are discussed next.

Minimal distance

One straightforward, off-the-shelf method for making an estimation of the size of the territory for both players is called *minimal distance*[4]. Here the assumption is that the player with the largest territory will win.

It defines a function $D^j(a)$. Here j is a player (either Black or White), and a is a field on the board. The value of $D^j(a)$ is the lowest number of plies player j has to make in order to place one of their amazons on a. For all empty fields a the values $D^{\text{Black}}(a)$ and $D^{\text{White}}(a)$ are determined. Whenever one is lower than the other, it is assumed that that field belongs to the player who is nearest. The difference between the size of Black's territory and the size of White's territory is returned.

During the opening phase this tends to be a very unreliable estimation since there are no clearly defined territories yet.

Lieberum's evaluation function

Evaluation functions can be complex and consist of multiple metrics that are combined into a single numeric value. The weights with which they contribute may vary over the different phases of the game.

Examples of such functions are provided by Tegos (2002) and Lieberum (2005) for their computer Amazons players *Antiope* and *Amazong* respectively.

[4]Some researchers refer to this as *minimal distance* (Lieberum, 2005) others as *minimum distance* (Muller & Tegos, 2002). In this thesis the former expression is used.

Not all parameters of these functions are provided by the researchers, so it is not possible to exactly replicate these evaluation functions based on the original papers. However, since Lieberum explains the different metrics in close detail and provides examples of inputs and corresponding outputs, the values of the implicit parameters can be inferred. In Section 5 an approximation of Lieberum's function is used, that simulates the original as closely as possible. In this thesis, this function is referred to as *Lieberum's evaluation function* or simply *Lieberum*.

Some of the measures used in Lieberum's evaluation function are:

- Another territory estimate, much like minimal distance. But rather than treating the amazons as chess queens, they are treated as chess kings. This value is more stable in the opening phase of the game.
- The mobility of the amazons, i.e. the number of fields they can move to. It is disadvantageous to have amazons that cannot move anymore.
- The distribution of the amazons on the board. Having an amazon in each quadrant of the board is preferable to having all amazons in a single cluster.

In order to know in what phase the game is in, the number of plies remaining until the endgame is estimated.

The measures are combined into a single value and the weight of each is dependent on the phase in the game. For example, in the opening phase, the distribution of the amazons is more important, while in the middlegame and endgame, the territory is more important. This yields in a high quality evaluation function.

One drawback of such a sophisticated evaluation function is its execution time. More measures mean a longer execution time per board. This reduces the total number of boards that can be evaluated within the limited time available.

Random

Another quick evaluation function is *random*. It simply returns a uniformly distributed random value between -0.5 and 0.5. Although this is a bogus evaluation function, combined with a search algorithm it will not always pick random plies (Hensgens, 2001).

A game state that has many possible plies, has a higher probability that at least one of the resulting game states scores highly, while a game state that has only few possible plies will have a lower probability that one of them gets a high score. In an algorithm such as alpha-beta pruning, this score is passed on, leading to a preference for game states where the player has a high mobility.

Neural Networks

An *artificial neural network* (ANN) can also act as an evaluation function, as is shown by Patist & Wiering (2004). This is elaborated upon in Section 4 where some of their research concerning Tic-tac-toe is replicated, and in Section 5 where these techniques are applied to Amazons.

3.3 Skipping search

Using a search algorithm to find the best ply to play may be a time consuming endeavor, especially when the search space is large. As discussed earlier, for Amazons this space is very large. In official tournament matches players usually have time constraints, so some researchers have examined methods of reducing the search time. One of these methods is building databases with standard responses to common situations. This would allow a computer Amazons player to simply pick a ply out of the database, without the necessity for building a game tree. The state space of Amazons is far too big for building a database containing all game states, that is why research is mainly focused on the opening and endgame.

Opening books

In other games such as Chess it is pretty common for players, both computer and human, to use some form of opening book. Such a book is a database that contains a part of the game tree with all plies that are considered good, along with their optimal response, up to a certain depth. These books can be either compiled manually by experts or constructed automatically and optionally curated by human experts afterwards.

Existing techniques for automatic construction of opening books that have proven to be successful in other games yield game trees that are deep and narrow. Amazons has a high branching factor at the start of the game compared the the

middle and the end of the game. Because of this, most games will not stay inside the opening book for more than a one or two plies.

During the opening phase, many of the available plies are of reasonably good quality, so one may consider an opening book based on a broader game tree. In order to keep the size of the database within reasonable limits, this opening book has to be shallow. The shallowness of the game tree is exactly where this approach falls short. Because of this, all games will not stay inside the opening book for more than one or two plies.

Opening books are by their nature impractically big or very limited. However, Karapetyan & Lorentz (2004) found that even such a limited opening book may prevent a computer Amazons player from making catastrophic mistakes at the very beginning and this allows for a slight advantage in play.

Endgame databases

As the game progresses, the board gets partitioned in separate and independent subboards. If only a single player has an amazon in such a separate area and plays carelessly, defective territory can be introduced, reducing the chances of winning. If both players have amazons in such small area, they still need to play in order to gain the largest territory in the subboard. Since each subboard can be regarded as a separate game, its game tree is much smaller. This allows for the construction of endgame databases (Song & Muller, 2015).

A program for playing endgames is called an *endgame engine*. It can be implemented by means of an endgame database or a specialized search algorithm and is regarded a necessary part for a computer Amazons player to play on tournament level (Kloetzer et al., 2009). However, this technique falls outside the scope of the thesis. For this reason such databases are not applied in the players introduced in Section 5.

3.4 Comparing computer players

It can be challenging to compare the performance of different computer Amazons players. There are three main reasons why it is hard for researchers to determine the quality of play of different algorithms (Avetisyan & Lorentz, 2002).

The first reason is that a method for quality assessment, that is commonly used in other games, is not applicable to Amazons. This technique involves replaying games played by experts. There are two problems that prevent this

method to be applied to Amazons. Firstly, there are few examples of high quality games available. Secondly, in case of Amazons, it is hard to quantify the quality of play this way. This has to do with the large branching factor and the chaotic nature of the game. When an algorithm is able to mimic other experts, this is obviously a sign of good play. However, when the plies are different, it is hard to quantify the error or gain. Moving the same amazon to the same spot, while shooting an arrow the other way may look like a very similar ply, but the slight difference can yield very different results. Similarly, moving another amazon instead, looks like a very different ply, but can in theory be equally good. Therefore, comparing plies this way takes a lot of knowledge of the game, and this knowledge is not readily available. Because of these two problems, replaying games played by experts is not a reliable method for assessing the quality of play of a computer Amazons player.

The second reason why it is hard to determine the quality of play is that programs do not necessarily have a consistent level of play. This means that program A can on average be better than program B, but if it occasionally makes fatal mistakes, it will still lose from B. This too makes comparing two programs a nontrivial task.

The third reason why it is hard to determine the quality of play is because it is a measure that is not easily quantifiable due to its nonlinear nature. That is, the 'plays better than' relation is not transitive. Consider two players A and B that are compared against a reference player. If player A has a higher equity than player B, player A is supposed to play better, and be able to win from player B. This is not necessarily true. Such a situation may occur in the game *Rock-paper-scissors*, for example. Suppose the reference player always plays *rock*, player A always plays *paper*, and player B always plays *scissors*. A will have equity 1, B will have equity -1. Although B has a lower equity than A, it will win in a match between the two. This fabricated example is quite extreme, but it shows that this way of comparing players is subject to the strategy of the reference player. Mapping the quality of play to a number suggests a linear ordering that in fact does not exist.

Avetisyan & Lorentz (2002) try to resolve these issues in a couple of ways. They look at quantifiable measures in their search algorithm, such as the number of nodes visited within a set amount of time. Within a single experiment this is a viable way of comparing the performance of different algorithms. However, this approach has its limitations too. Programs with different search algorithms

are hard to compare since they do not need to share the same metrics. For example alpha-beta pruning and Monte Carlo tree search have a very distinct way of exploring nodes, which makes this metric unfit for comparison. Moreover, running the same program on different hardware can have a large effect on the results. Lastly, they try to find particularly good or bad plies, in order to analyze the strengths and weaknesses of the programs. This technique requires a certain level of game insight and expertise from the researchers, and yields only scant data.

Patist & Wiering (2004) compare different algorithms by letting them play matches against each other for many times. One might miss subtleties in the programs because relatively good players may lose because of occasional blunders, and thus be considered bad. This is regarded as the most fair way to compare two players nonetheless. In the end, good players tend to win more games than bad players.

The measure they use is called *equity*. The equity of one player against another player is defined as

$$equity = \frac{w - l}{w + l + d}$$

where w is number of wins, l is number of losses, and d is the number of draws. In case of Amazons d always equals 0. When a player is unbeaten their equity is 1, and when a player loses invariably their equity is -1. Whenever a player is just as good as their opponent the equity of both players is 0.

There are two main drawbacks to using equity as a means to compare players.

Firstly, using equity as measure for comparing players requires a good reference player. This player cannot be too good nor too bad. If it is too good it will win most of the time, yielding an equity of nearly -1 for all examined players, regardless of their playing level. If the reference player plays too poorly it will lose most of the time, yielding an equity of nearly 1 for all examined players. So equity is only a valid measure, when a reasonable good reference player is available.

Secondly, as equity is a number, it assumes a linear order on the set of all players. Because this is a false assumption, it may lead to false conclusions.

4 Preliminary study on Tic-tac-toe

Using neural networks as evaluation functions is not a new idea. Patist & Wiering (2004) showed that it is feasible to use this technique for the two games Tic-tac-toe and Draughts. In this section two of their experiments regarding the former are replicated. In Section 5 this technique is extended to Amazons as well.

Tic-tac-toe is a two player game, typically played with pen and paper, in which the players take turns placing their mark (either × or ○) in a 3×3 grid. A player wins if three of their marks are in a row, either horizontally, vertically or diagonally. Figure 8 shows an example of a Tic-tac-toe match.

Traditionally Tic-tac-toe is played with two players × and ○. For the sake of similarity with Amazons they are called White and Black respectively in this thesis.

Two experiments with Tic-tac-toe are conducted. In the first experiment the training examples are generated on the fly without using a database, whereas in the second experiment a database of readily available games is used.

The resulting ANN is a function that maps game states to numeric values. Because it is an evaluation function, a value of zero means the game state offers no advantage to either Black or White, and the more positive the value is the more advantageous its game state is for White and reversely, the more negative the value is the better its game state is for Black.

The resulting network can be used with any search algorithm that requires an evaluation function to construct a Tic-tac-toe player. Tic-tac-toe is a very simple game, with a game tree complexity of 10^5. For comparison, for Chess

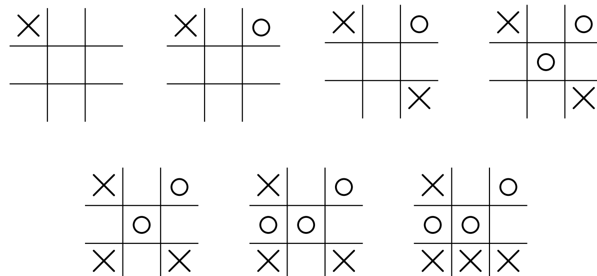


Figure 8: A typical progression of Tic-tac-toe. In this example × wins after 7 plies.

the game tree complexity is 10^{123}, for Amazons it is 10^{212} (Hensgens, 2001) and for Go it is 10^{360} (Allis et al., 1994). Because of this relatively low complexity, a search algorithm with lookahead would quickly lead to a *brute force* approach. That is, all possible futures until game termination can be considered, and there is no need for an evaluation function. This does not give any insights in the performance of the acquired networks. For this reason testing the performance of the networks is done by using them in combination with naive algorithms that do not use any lookahead. It simply selects the ply with the most advantageous value according to the evaluation function. Because this player only uses the network and nothing more, this player is called NETWORK.

There are two other Tic-tac-toe players that are used as opponents to NETWORK in these experiments. These opponents are called EXPERT and RANDOM.

EXPERT is fairly good Tic-tac-toe player, albeit not perfect. Its strategy is strictly defined, although it does not play deterministically. Its strategy is this: play a random winning ply (if there is any), otherwise block a winning ply for the opponent (if there is any). If that is both not the case, it will simply play a random ply. It does not deliberately construct any *fork*[5], nor does it aim for the center field over corners or sides, as one would in perfect play (Crowley & Siegler, 1993). Thanks to this imperfect play it is a valuable opponent to be used when comparing the performance of NETWORK. This comparing is elaborated upon in Section 4.2.

RANDOM simply selects a ply at random every time it plays. This leads to very poor performance, yet such an player is easy to construct, and its algorithm is very fast to execute. NETWORK will play against this player when constructing training examples on the fly.

4.1 Method

When training an artificial neural network, there are many parameters. This section describes these parameters and the selected values.

Representation

Neural networks take numerical column vectors as their input. That means that in order for a neural network to rate a Tic-tac-toe game state, the board needs

[5] A fork is a situation in which a player has two opportunities to win. In Figure 8, player × creates a fork in the fifth ply.

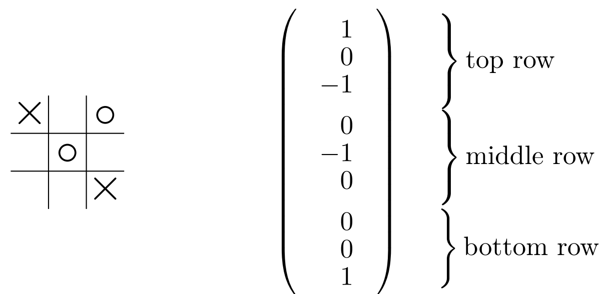


Figure 9: A game state of Tic-tac-toe and its representation as a column vector. The fields of the board are mapped to elements in the vector in natural reading order.

to be represented as a vector. This is done by mapping each field of the board to an element in the vector, yielding a column vector with 9 elements. An empty field is represented by 0, a field for Black is represented by negative 1, and a field for White is represented by positive 1. An example of the board to vector mapping can be seen in Figure 9.

Network topology

The constructed neural network is a fully connected feedforward network with a single hidden layer, and no skip-layer connections. The number of input nodes is equal to the number of elements in the input vector and thus also equal to the number of fields a board, i.e. 9. The output vector has just one single numerical element (the evaluation value), so there needs to be only a single output node. The number of hidden nodes is varied within each experiment and is either 40, 60 or 80.

All nodes that are not input nodes are called *activation nodes* and have an activation function. Each hidden node has an Elliot symmetric activation function, which is defined as

$$y = \frac{\beta \cdot x}{1 + |\beta \cdot x|}$$

where x is the sum of the inputs of the node, y is the output of the node, and β is the *neuron sensitivity* or *steepness* (Sibi et al., 2013). The output node has a linear activation function. All activation nodes have a *bias*, i.e. an extra input that always has the value 1.

Training algorithm

The network is trained using reinforcement learning. This means that the network is iteratively presented with an example game of which the true value is known, that serves as target. After each exposure to an example, the weights of the network are slightly modified such that the error (the difference between the output and the target) for this specific example is reduced.

The used algorithm is a form of *temporal difference learning*. This is a modification of back propagation that is particularly well suited for discrete time series like combinatorial games. With classical back propagation, the network is trained to predict the final outcome of the game. In contrast, with temporal difference learning the network is trained to predict the outcome of the next ply, rather than the final outcome of the game.

The variant of temporal difference learning that is used is called the *TD*(λ) *algorithm*, in which the parameter λ indicates to what extend the future steps are taken into consideration when defining the target value for an example game state. For $\lambda = 1$ only the final outcome is considered (that is, TD(1) is equivalent to the classical back propagation algorithm), for $\lambda = 0$ only the single next game state is considered. A value somewhere in between typically yields the best predictions (Sutton, 1988). This means that the first upcoming game step contributes the most, and the subsequent steps contribute progressively less.

Patist and Wiering changed the sensitivity rate β during the training phase, in order to speed up the training process. This would add another parameter to the experiments, and because modifying the sensitivity rate is not part of the scope of this research, this technique is not applied. To be able to reach similar results as Patist and Wiering, more training examples are used instead.

All parameters used for the experiments are summarized in Table 1.

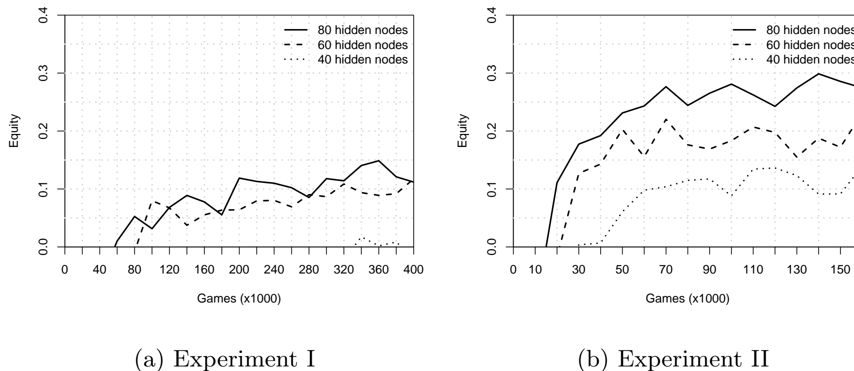
The training examples

Two experiments with Tic-tac-toe were conducted, the only difference being the way the training examples are obtained. In Experiment I the training examples are constructed on the fly, in Experiment II they are provided by a database of readily available games.

The training examples that are created during Experiment I are games played by NETWORK against RANDOM. Since RANDOM plays so poorly these

Experiment	I	II
Learning paradigm	Reinforcement	Supervised
Number of input nodes	9	9
Number of hidden nodes	40, 60 or 80	40, 60 or 80
Activation function hidden nodes	Elliot symmetric	Elliot symmetric
Hidden neuron sensitivity β	5	5
Number of output nodes	1	1
Activation function output nodes	Linear	Linear
Output neuron sensitivity β	1	1
Random initial weights range	$[-0.2, 0.2]$	$[-0.2, 0.2]$
Learning rate weights	0.00105	0.00105
Learning rate neuron sensitivity	0	0
Lambda λ	0.8	0.8
Number of training examples	40,000	160,000
Examples in database	N.A.	40,000
Performance test after each	2,000	10,000
Number of simulations	10	10

Table 1: Configurations of the Tic-tac-toe experiments



(a) Experiment I

(b) Experiment II

Figure 10: Results for Tic-tac-toe

games are hardly representational to games played by EXPERT. However, since NETWORK is able to apply its own strategy and see how it holds up, it can get feedback on its own way of play.

The training examples that are used for Experiment II are games played beforehand by NETWORK against EXPERT. Since EXPERT plays reasonably well and NETWORK is learning during the construction of the database, these games are of better quality compared to Experiment I. However, the network that is trained during the construction of the database is discarded afterwards. The network that is trained during the actual experiment is only allowed to look at the database, and is not able to put its own strategy to the test.

This means that Experiment I is an example of *reinforcement learning* while Experiment II is an example of *supervised learning.*

4.2 Results

To assess the quality of the obtained ANNs, the equity is calculated. In order to get insight in how the quality of the generated network progresses during the training process, the equity of NETWORK against EXPERT is determined after every 2,000 and 10,000 examples for Experiment I and II respectively. In order to smoothen the results, each experiment is run for 10 simulations and the results are averaged. These results are graphically shown in Figure 10.

For both experiments holds: having more hidden nodes yields better results than having less hidden nodes. However, there is no significant difference between having 60 and having 80 hidden nodes in Experiment I. This can be

explained by the fact that the network learns from playing against RANDOM, which plays so poorly that it cannot distill a solid strategy. And the simple strategy it does learn can be captured in 60 nodes as easily as in 80. However, when learning from EXPERT (in Experiment II), it is able to learn a more complex strategy (creating a *fork* for example), and this is more easily picked up on in 80 hidden nodes rather than in 60.

In both experiments, the equity plateaus once they are exposed to about 50 % of the training examples. This suggests that even more training cycles with these examples would not result in improved play. There is, however, a big difference in the value at which the equity levels out. In Experiment I the network player has a maximum equity of 0.15 and in Experiment II it has a maximum equity of 0.30. One would expect that reinforcement learning would yield better results compared to supervised learning. This is not the case, because the quality of the training examples in Experiment II is so much better, it overcompensates for the fact that the network player in Experiment II is not allowed to select its own plies.

These results are similar to the results from Patist & Wiering (2004). They find for Experiment I a maximum equity of about 0.14, which is roughly equivalent to 0.15. However, they find an even larger maximum equity for Experiment II, 0.50.

The difference between the obtained results and the results of Patist and Wiering may be explained by the randomness that is inherent to the experiments. The initial weights are randomly chosen, and also the behavior of the reference players (both RANDOM and EXPERT) is not deterministic. The database is constructed with the aid of both the network and the reference players, so its construction is subject to randomness as well. This randomness causes the experiments to yield different results every time they run. On the other hand, Patist and Wiering also use an average of 10 runs, which reduces the noise, so there is probably something else that contributes to the different outcomes.

Another possible explanation is that adaptive neuron sensitivity is not used in both experiments, while it was used by Patist and Wiering. This technique leads to faster convergence. Although more training cycles are used to account for this slower convergence, this may not have been enough to fully compensate for the lack of adaptive neuron sensitivity.

As it is stated in the original work of Patist and Wiering, the conclusion of these experiments is that using a database of training examples can be a valid

way of training a network. In the next section these same techniques will be applied to Amazons.

5 Study on Amazons

Several experiments with Amazons are conducted. As with the Tic-tac-toe experiments, the constructed neural networks can be applied as evaluation functions.

Because Tic-tac-toe is an easy game, it is possible to assess the quality of the networks quite fast. However, Amazons is much more complicated and time consuming when played by a computer. That is why it is not possible to construct graphs like Figure 10 for the experiments with Amazons. Only at the end the equity is calculated for each network.

The equity is relative to a reference player. In order to give a more objective measure of how well the data are learned also the errors of the networks are calculated.

5.1 Method

The method used for these experiments is mostly the same as for the experiments with Tic-tac-toe. Some game-specific aspects are discussed in this section.

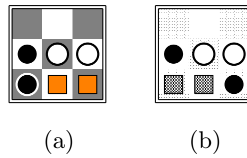
Representation

An evaluation function takes a game state as input and produces a numerical value as output. In the case of Amazons a game state is simply a board configuration. A neural network takes a column vector of integers as input, so before an Amazons board can be processed by a neural network based evaluation function it needs to be be transformed into a vector.

Mapping Amazons boards to vectors of integers can be done in a multitude of ways. Since Tic-tac-toe does not have barricades as Amazons does, simply using the same method as in Section 4.1 is not possible. Each field has one of four different values: containing a black amazon, being empty, containing a white amazon or containing a barricade. It is possible to map each value to a different integer, for example $-1, 0, 1, 2$ respectively. This would impose an ordering on these values that in fact do not have any natural ordering at all.

To avoid such an artificial ordering three different representations are compared. All of these encode a field by multiple integers instead of just one.

The first representation is most similar to the one used in Tic-tac-toe, except that each field is encoded by two integers. The first integer indicates the presence



(a) (b)

Figure 11: Two different simplified Amazons boards may share the same representation as column vector when using the MOBILITY representation, as is shown in Figure 12.

of an amazon. Similar to Tic-tac-toe, negative 1 means an amazon for Black, positive 1 means an amazon for White and 0 means no amazon at all. The second integer indicates the presence of a barricade, either a barricade or an amazon. Here 1 means a barricade and 0 means no barricade. Since this representation only uses unit values it is referred to as 'UNITS'.

The second representation is a modification of the first. Instead of using unit values (1 and -1) to indicate the presence of amazons, the mobility of the amazon in question is used. This, however, yields a set of encodings that does not have a bijective relation with the set of boards, i.e. amazons with no mobility become indistinguishable from simple barricades. See for example Figures 11 and 12. Since these amazons are not useful for the player, this will have a minimal effect on the evaluation value. That is why this property is not considered to be problematic. Since this representation uses the mobility value of amazons it is referred to as 'MOBILITY'.

The third representation is a bit representation, where each bit is actually the integer 0 for *false* or 1 for *true*. It consists of four bits, one for each possible value. The first bit indicates the presence of a black amazon, the second bit indicates the presence of a white amazon, the third bit indicates the presence of a barricade and the fourth bit indicates an empty field. Since this representation uses bits it is referred to as 'BITS'.

Having each field represented by multiple integers has a major drawback: the input vector has more elements, causing the network to get bigger. This in turn results in longer times for propagating (predicting) and back propagating (learning). Moreover, because the number of weights between input nodes and hidden nodes gets multiplied, there are more weights to optimize so the search space for the learning algorithm becomes bigger too, leading to slower convergence.

$$
\begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 5 \end{pmatrix}
\begin{array}{l} \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{color} \\ \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{blockades} \\ \left.\begin{array}{c} \\ \\ \\ \end{array}\right\} \text{features} \end{array}
$$

UNIT

$$
\begin{pmatrix} 0 \\ 0 \\ 0 \\ -2 \\ 3 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 5 \end{pmatrix}
\begin{array}{l} \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{mobility} \\ \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{blockades} \\ \left.\begin{array}{c} \\ \\ \\ \end{array}\right\} \text{features} \end{array}
$$

MOBILITY

$$
\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 2 \\ 5 \end{pmatrix}
\begin{array}{l} \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{white amazons} \\ \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{black amazons} \\ \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{barricades} \\ \left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{empty fields} \\ \left.\begin{array}{c} \\ \\ \\ \end{array}\right\} \text{features} \end{array}
$$

BITS

Figure 12: The three representations of the simplified Amazons board depicted in Figure 11a. For intelligibility, the elements are ordered by category and then by field rather than by field and then by category. Each category except 'features' has nine elements, corresponding to the nine fields on the board in natural reading order. The category 'features' contains three global features and do not correspond with any particular field.

Representation	Units	Mobility	Bits
Field with black amazon	$(-1, 1)$	$(-m, 1)$	$(1, 0, 0, 0)$
Field with white amazon	$(1, 1)$	$(m, 1)$	$(0, 1, 0, 0)$
Field with barricade	$(0, 1)$	$(0, 1)$	$(0, 0, 1, 0)$
Empty field	$(0, 0)$	$(0, 0)$	$(0, 0, 0, 1)$
Dimensions (excl. features)	200	200	400
Dimensions (incl. features)	203	203	403

Table 2: Considered representations of the Amazons board. For fields with an amazon, m represents the mobility of the amazon in question.

Patist & Wiering (2004) claim that learning on raw board data alone is hard, that is why they include higher level game features into their board representation of Draughts. They differentiate between two kinds of features, *global features* and *structural features*.

Global features are features that cannot be localized somewhere on the board. An example of a global feature in Draughts is material balance. Structural features are patterns of occupied and empty fields that can occur somewhere on the board. An example of a structural feature in Draughts is the *fork lock*[6].

An obvious global feature for Amazons is the size of the territory. Calculating the size of the territory requires a *flood fill* which is a time expensive algorithm. Using the size of the territory as metric would defeat the purpose of using otherwise quick neural networks as an evaluation function. That is why only very simple features are included in the representations. These features are:

- Number of plies in the game. This is equal to the number of barricades on the board.
- Total mobility of Black.
- Total mobility of White.

Table 2 shows a comparison of all three representations. Figure 12 compares them by means of an example.

[6] The *fork lock* also known as *fence position* or *hekstelling*.

Experiment	Amazons
Learning paradigm	Supervised
Number of input nodes	203 or 403
Number of hidden nodes	40, 80 or 160
Activation function hidden nodes	Elliot symmetric
Hidden neuron sensitivity β	3
Number of output nodes	1
Activation function output nodes	Linear
Output neuron sensitivity β	1
Random initial weights range	$[-0.2, 0.2]$
Learning rate weights	0.0005
Learning rate neuron sensitivity	0
Lambda λ	0.9
Number of training examples	200,000
Examples in database	8,618
Number of simulations	10

Table 3: Configurations of the Amazons experiment

Network topology

The network topology for this experiment is similar to the one used in the experiments with Tic-tac-toe. One big difference is the number of input nodes. As stated in Table 2 this is either 203 or 403.

Patist & Wiering (2004) have had positive results with the same number of hidden nodes (40, 60 or 80) in combination with 222 input nodes for the game of Draughts. However, Amazons is a more complex game than Draughts. In order to allow for more complexity, the size of the hidden layers is increased to 40, 80 or 160 hidden nodes.

Training algorithm

The training algorithm is similar to the algorithm used with the experiments with Tic-tac-toe. The parameters used are shown in Table 3.

Training examples

In the Tic-tac-toe experiments the construction of the database is part of the procedure. The database is constructed by collecting all games played by a new neural network that learns from playing against EXPERT. In contrast to Tic-tac-toe, Amazons cannot be played quickly, so learning by playing would take extremely long. Moreover, there are some reasonably good computer players but these are not readily available to construct a database of representative games. Therefore, the Amazons database has to come from a different source.

Little Golem is an online game platform where players can play various games against each other, including Amazons. Most players are human, although some players are actually computers or *bots*[7]. Little Golem has a ranking system, and all games played are stored and retrievable through the website. All finished games played by the 60 highest-ranking players form the basis for the dataset.

There is some noise in the dataset, i.e. not all games are suitable to be used in the database. For example, some players start a game, but abandon it without ever playing their first ply. Others abandon it half way through the game. These games are not considered useful to learn from and are excluded from the database. Only games that clearly have a winner are included in the database.

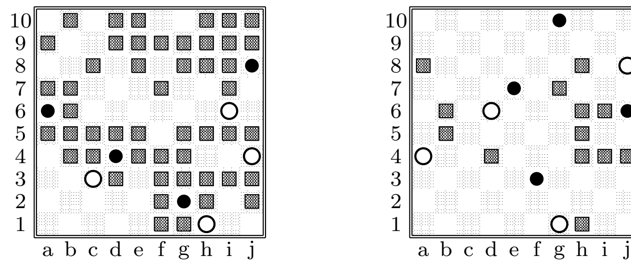
Since the endgame tends to be the most tedious part of the game, most games are not played through until a terminal game state is reached. In most of these games, there is a clear winner, though. That is, unless one player makes a blunder, the other player is not able to win. In order to determine whether or not a game that does not end in a terminal game state has a clear winner, the endgame is simulated.

Before simulating the endgame, the last available game state is evaluated to find whether Black or White seemed to be in the advantageous position. Then the game is finished by a good computer player[8] and a computer player that does not play very well[9]. The simple player plays on the winning side and the good player plays on the losing side. If the simple player is able to win against the good player, this game is considered to be an indisputable win, and it is included in the database. And vice versa, if the good player is able

[7] Examples of Amazons bots on Little Golem are *Invader_bot* and *Lysistrate bot.*

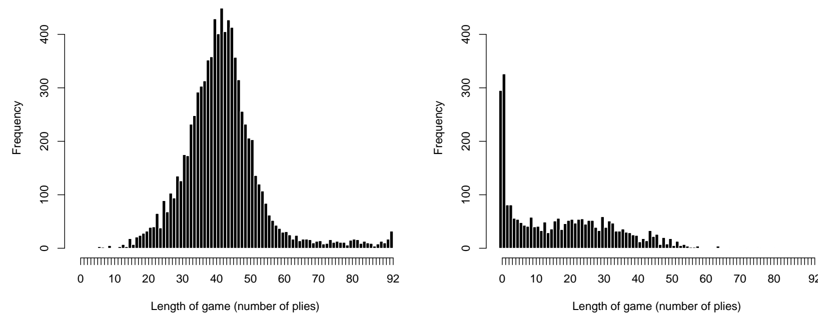
[8] The good player uses an evaluation function based on the work of Lieberum (Lieberum, 2005) and alpha-beta pruning in combination with iterative deepening.

[9] The simple player uses a simple minimal distance evaluation function and no lookahead.



(a) Clear win for White; accepted (b) No clear winner; rejected

Figure 13: The last states of two games from the database. One has an indisputable winner, and one does not because it was abandoned early.



(a) Accepted games (b) Rejected games

Figure 14: Histograms of the length of the games in plies.

to make a comeback and win even though it started with a disadvantage, this game is considered to be still undecided, and it is excluded from the database. Examples of accepted and rejected games are shown in Figure 13. 8,618 games are accepted, and 2,624 games are rejected. Figure 14a shows the lengths of the games included in the dataset. Figure 14b shows the lengths of the games that are rejected.

In order to assess the quality of the network, the database is randomly split up in a training set comprising 90 % of the games and a test set of the remaining 10 % of the games. The networks are trained using only the records in the training set.

5.2 Results

Two measures are used to assess the quality of the acquired networks.

On the one hand the neural networks are treated simply as predictors. When the networks are used to evaluate an instance of a board in the test, its predicted value is compared to the known true value. The difference is called the error.

On the other hand the neural networks are treated as part of a computer Amazons player. Their equities with respect to a reference player are a measure for how good they are at playing Amazons. This is the same measure as is used in the experiments with Tic-tac-toe.

The three representations (UNIT, MOBILITY and BITS) combined with three amounts of hidden neurons (40, 80 or 160) produce a set of nine different configurations. Each of these configurations is used to run 10 simulations, with the same data but with different initial weights. For each configuration only the simulation in which the network has the lowest error is taken into account.

Error of evaluation functions

To determine the error of a network, it is tested against a test set of 37,322 boards (out of 863 games). Each of these boards is represented as a vector and used as an input for the network. The value of the output neuron of the network is indicative of which of the players is expected to win. Because all true outcomes of all games in the test set are known, the percentage of boards for which the network is mistaken can be calculated. This percentage is called the *error*.

This definition of error can be applied not only to networks, but also to other evaluation functions. For comparison, also the errors of an advanced evaluation function, a simple evaluation function and a bogus evaluation function are calculated. See Section 3.2 for more details about these functions.

Figure 15 shows the errors of the evaluation functions and the networks. The most advanced evaluation function Lieberum has a lower error than the simpler evaluation function minimal distance (0.24 and 0.25 respectively). The random evaluation function has an average error of 0.50, being correct about half of the time.

There is little variation in the errors of the networks ranging from 0.37 to 0.38.[10] The best network based evaluation function is the network with the most hidden nodes (160) and the most complex representation (BITS); the worst network based evaluation function is the network with 80 hidden nodes and the

[10]This range is shown in Figure 15 by a dashed line for easy comparison.

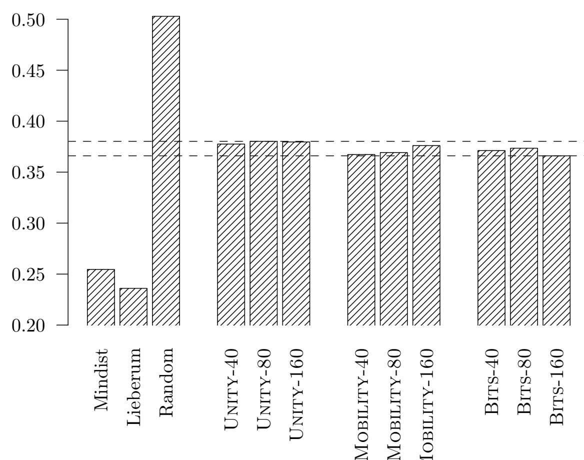


Figure 15: Bar plot of the mean errors on the test set of the different network configurations, as well as for the reference classifiers. The highest and lowest errors of the networks are marked with dashed lines, for easy comparison.

Unity representation.

As is shown in Figure 14 not all games last the full 92 plies. There are far more training examples for early and middlegame boards (up to about ply 50) and far fewer endgame boards (about 50 and higher). This means that the performance of the networks may vary over the course of a game. To give insight in this phenomenon, the average of the error of the networks is plotted as a function of the number of plies played on the board in Figure 16.

Analogous to the training set, the test set gets sparser for higher plies, leading to more unstable graphs. This effect is best exemplified by the random evaluator in Figure 16a. This evaluator is expected to have an constant error of 0.50. The graph is pretty stable around the value 0.50 for plies 0 to 50, but becomes progressively more variable for higher plies.

All other evaluators share these characteristics: at the start of the game (ply 0) the error is 0.50 which is to be expected since neither of the players start with an advantage over the other. As the game progresses the error becomes smaller and smaller, i.e. it becomes more clear which player is going to win.

There is, however, a big difference between Lieberum and minimal distance and the network based evaluators. The traditional evaluators keep having a much lower error pretty much from the start, dropping to below 0.10 for boards on ply 37 and up. The network based evaluators errors stagnate after about 10

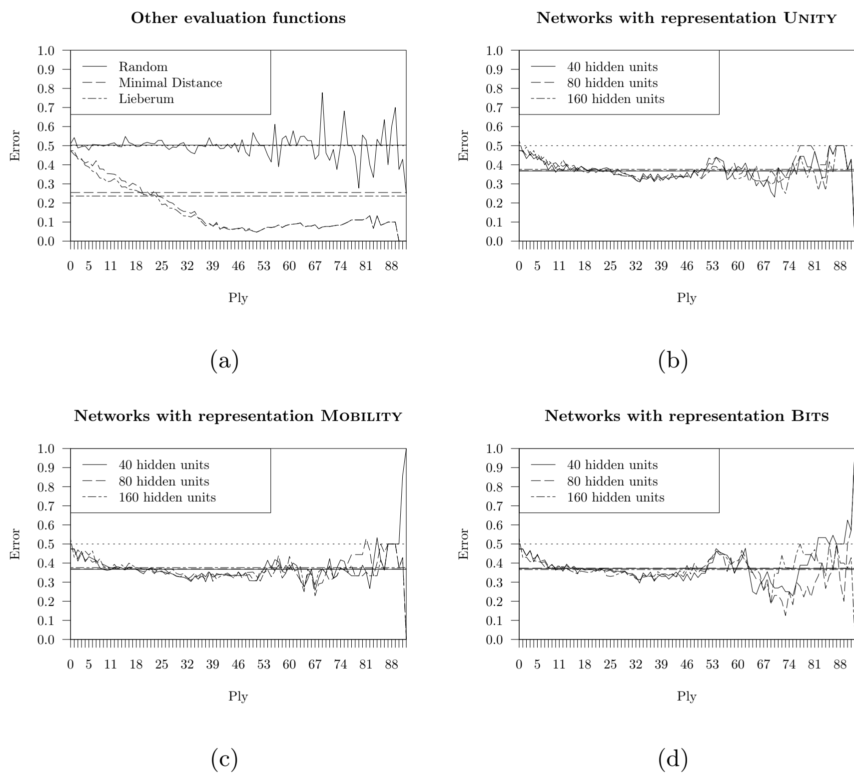


Figure 16: The average error of the evaluation functions broken down per number of plies played. The horizontal line is the average error over the whole test set.

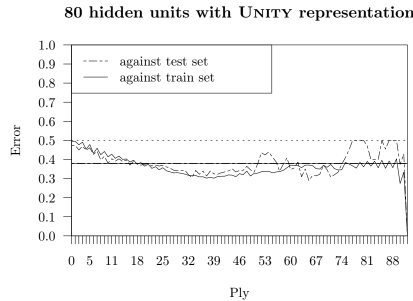


Figure 17: The average error of an evaluation function broken down by number of plies played on both the train and the test set.

plies, and do not get any lower than 0.30, except for some networks in the late endgame.

The error measure can be used to check whether the networks are overfitted. An overfitted network is characterized by an error in the training set that is much lower compared to the error in the test set. Figure 17 shows the error in the test set and in the original training set for the one configuration with the poorest performance. The test set graph is a bit less stable, due to the smaller sample size compared to the training set. Both graphs are almost identical for the first half of the game, until about ply 50. After that the network has better results on the training set than on the test set. This suggests that the network is not overfitted for the first half of the game, and becomes increasingly overfitted in the second half. Since the number of games to learn from drops significantly after ply 50, this result is to be expected.

Equities

Since the networks are meant to make computers play better, an obvious way to compare the quality of the networks is by using them in actual play. In Section 4.2 the equity of the networks is determined after each additional 2,000 training examples, showing the progress during the training phase. As Amazons is more complex compared to Tic-tac-toe and a single game may take up to 20 minutes to finish, the equities of the networks are determined only once, i.e. at the end of the training phase.

Computer Amazons players are constructed with the networks and with three different search algorithms: no lookahead, alpha-beta pruning, Monte

	No lookahead	Alpha-beta	MC with UCT
Lieberum	-0.94	-0.02	-0.76
Minimal distance	-0.98	-0.68	-0.98
Random	-1.00	-1.00	-1.00
Any network[11]	-1.00	-1.00	-1.00

Table 4: Equities of all players against the good player.

Carlo tree search. See Section 3.1.

As with the calculation of the errors, the three reference evaluation functions are included as well: minimal distance, Lieberum and random.

As with with Tic-tac-toe, an appropriate reference player to play against is required. The results against the good player from Section 5.1 are shown in Table 4. The good player is a combination of the Lieberum evaluation function with the alpha-beta pruning search algorithm. That is why the equity of that combination is nearly zero (-0.02). All other combinations perform worse. So much so, that these equities convey hardly any information at all, and it is useless to compare the different networks with this measure.

In order to be able to compare the in-game performance of the networks, a second, more inferior reference player is used: the simple player from Section 5.1. These results are shown in Table 5. The simple player uses the minimal distance evaluation function with no lookahead. That is why the equity of that combination is zero. All network based players have a negative equity, meaning they lose more often than they win. The variance in equities is discussed in Section 6.

[11] Of the nine networks, two actually achieved an equity of -0.98 instead of -1.00 viz. the network with 40 hidden nodes and the MOBILITY representation (using MC with UCT) and the network with 80 hidden nodes and the BITS representation (using alpha-beta pruning). However, since these results are not considered relevant, the table is simplified and all networks are rallied in a single category.

	No lookahead	Alpha-beta	MC with UCT
Lieberum	0.42	0.98	0.78
Minimal distance	0.00	0.82	0.44
Random	−1.00	−0.98	−1.00
Network: Unity-40	−0.92	−0.86	−0.70
Network: Unity-80	−1.00	−0.84	−0.68
Network: Unity-160	−0.96	−0.86	−0.72
Network: Mobility-40	−0.92	−0.80	−0.50
Network: Mobility-80	−0.92	−0.76	−0.70
Network: Mobility-160	−0.92	−0.88	−0.78
Network: Bits-40	−0.94	−0.86	−0.66
Network: Bits-80	−0.96	−0.90	−0.84
Network: Bits-160	−0.96	−0.90	−0.78

Table 5: Equities of all players against the simple player.

6 Discussion

Section 5 describes the construction of neural networks that are able to evaluate Amazons boards, and that as such can assist computer players in searching for the best ply to play. Traditional evaluation functions tend to produce evaluations of higher quality compared to the neural network based functions. Computer Amazons players that use the neural network based evaluation functions are not able to play at the level of traditional computer Amazons players. In this section potential explanations for these results are discussed.

6.1 Complexity

As is shown in Figure 15, neither the complexity of the network (the number of hidden neurons) nor the used representation have a significant effect on the error. There is a hint that the UNITY representation has a lower performance than the other two, although this effect is very small.

The complexity of the network does have a big impact on performance in actual play. As is shown in Table 5, for all representations holds that the *least* complex network outperforms the more complex networks. This effect is seen most strongly in players that use the Monte Carlo tree search algorithm.

The quality of each individual evaluation is less important for Monte Carlo tree search than for example for alpha-beta pruning. For Monte Carlo tree search, the more paths are simulated, the more reliable the result is. This explains why the less complex (and quicker) networks outperform the more complex (and slower) networks.

In contrast to performance of play, the error measure is not influenced by the execution time. So why do the more complex networks not have a lower error? In case of the MOBILITY representation there is an inverse relation between network complexity and error. This may be due to the fact that more complex networks take longer to converge. An ill chosen parameter in the training method, such as the number of training examples can contribute to this phenomenon. It may also be true that a 'complex' network of 160 hidden nodes is still too simple for capturing anything relevant with respect to the game of Amazons.

6.2 Representation

When representing Draughts boards as vectors, Patist & Wiering (2004) not only include the raw board data, they also include high level features. They used 23 global features and 99 structural features. These networks have good performance.

There is no list of structural features readily available for Amazons. Only three global features and no structural features are used in the experiments with Amazons (see Section 5.1). The resulting networks do not have the expected performance.

This suggests that including useful features to represent Amazons boards can be beneficial when learning from a database.

There is an obvious pitfall to this: extracting these features takes time. For example, getting the size of the territory of both players in Amazons requires the expensive flood fill algorithm. Using this measure as global feature may yield a lower error. However, it also means that when new boards are evaluated by the network these features need to be calculated. And if these operations are time expensive, it may nullify all the time benefits from using a quick neural network as evaluation function. When evaluating takes a long time, the search algorithm such as Monte Carlo tree search, will not be able to evaluate many boards before it runs out of time, which leads to poor performance.

The question of which features are beneficial for the game of Amazons may be answered in future research.

6.3 Topology

Some research takes the concept of features, and incorporates it into the network itself. For example Fogel & Chellapilla (2002) construct an ANN for Checkers[12]. They use a feed forward neural network with a single output neuron. The network is not fully connected and has three hidden layers. The first hidden layer is a spatial preprocessing layer, where each node represents a square segment of the board. On a board of 8×8 fields there are 36 subsets of fields of size 3×3, and 25 subsets of size 4×4, etc. Each of these subsets gets its own node in the first hidden layer. This topology allows the network to extract features based on the spatial characteristics of the board.

[12] Checkers is a variant of Draughts that is played on a 8×8 board, instead of 10×10 board.

Another promising approach is *deep learning*. This involves a topology with many hidden layers. Although it is proven that all networks with multiple hidden layers can be transformed into equivalent single hidden layer networks, there is a computational advantage to using multiple hidden layers instead (Mitchell et al., 1997).

There are successful studies of using *Convolutional Neural Networks* (a neural network architecture for deep learning) for feature extraction in images. Because Amazons boards can be seen as 10×10 bitmap images, similar techniques may be used for extracting features or evaluation values from Amazons boards.

Others have had very positive results with playing computer Go using a neural networks with up to 14 hidden layers (Silver et al., 2016).

Whether it is beneficial to use multiple hidden layers when finding structural features for Amazons may be a topic of future research.

7 Conclusion

In Sections 2 and 3 of this thesis, the rules and strategies of Amazons are explained, and an overview of Amazons as a research field is provided. Also a way to effectively use Monte Carlo tree search with neural network based evaluation functions is proposed.

In Sections 4 and 5, several experiments are performed to evaluate the usability of this suggested technique. It is shown that previous research with a data-driven approach to using neural networks as evaluation functions can be successfully replicated. This technique applied to Amazons yields functions that are able to evaluate Amazons boards, albeit not as adequately as traditional evaluation functions such as *minimal distance*.

Future research could explore the possibilities of using structural features as input, rather than raw board data. This may be achieved either by close analysis of the game and extracting a formally defined list of possibly predicting patterns, or by using more complex network structures to extract these patterns by means of machine learning as well.

References

Allis, L. V., et al. (1994). *Searching for solutions in games and artificial intelligence.* Ponsen & Looijen.

Avetisyan, H., & Lorentz, R. J. (2002). Selective search in an amazons program. In *Computers and games* (pp. 123–141). Springer.

Berlekamp, E. R. (2000). Sums of n× 2 amazons. *Lecture Notes-Monograph Series*, 1–34.

Brügmann, B. (1993). *Monte carlo go* (Tech. Rep.). Physics Department, Syracuse University. (Vol. 44)

Buro, M. (2001). Simple amazons endgames and their connection to hamilton circuits in cubic subgrid graphs. In *Computers and games* (pp. 250–261). Springer.

Crowley, K., & Siegler, R. S. (1993). Flexible strategy use in young children's tic-tac-toe. *Cognitive Science*, *17*(4), 531–561.

Fogel, D. B., & Chellapilla, K. (2002). Verifying anaconda's expert rating by competing against chinook: experiments in co-evolving a neural checkers player. *Neurocomputing*, *42*(1), 69–86.

Hensgens, P. P. (2001). *A knowledge-based approach of the game of amazons.* Unpublished doctoral dissertation, Universiteit Maastricht.

ICGA. (2016). *Computer Olympiad 2016.* `https://icga.leidenuniv.nl/?page_id=1764`. (Online; retrieved November 2015)

Kadon Enterprises, Inc. (2014). *Abstract Strategy Games – Page 3 of 6.* `http://www.gamepuzzles.com/abstrct3.htm#AM`. (Online; retrieved October 2014)

Karapetyan, A., & Lorentz, R. J. (2004). Generating an opening book for amazons. In *International conference on computers and games* (pp. 161–174).

Keller, M. (2009). *El Juego de las Amazonas (The Game of the Amazons) by Walter Zamkauskas.* `http://www.solitairelaboratory.com/amazons.html`. (Online; retrieved October 2014)

Kloetzer, J., Iida, H., & Bouzy, B. (2007). The monte-carlo approach in amazons. In *Proceedings of the computer games workshop* (pp. 185–192).

Kloetzer, J., Iida, H., & Bouzy, B. (2009). Playing amazons endgames1. *Icga Journal*, *32*(3), 140–148.

Lieberum, J. (2005). An evaluation function for the game of amazons. *Theoretical computer science*, *349*(2), 230–244.

Lorentz, R. J. (2008). Amazons discover monte-carlo. In *Computers and games* (pp. 13–24). Springer.

Mitchell, T. M., et al. (1997). *Machine learning. wcb.* McGraw-Hill Boston, MA:.

Müller, M. (2001). Solving 5×5 amazons. In *The 6th game programming workshop (gpw 2001)* (pp. 64–71).

Muller, M., & Tegos, T. (2002). Experiments in computer amazons. *More Games of No Chance*, *42*, 243.

Nowakowski, R. J. (1998). *Games of no chance* (Vol. 29). Cambridge University Press.

Patist, J. P., & Wiering, M. (2004). Learning to play draughts using temporal difference learning with neural networks and databases. In *Proceedings of the thirteenth belgian-dutch conference on machine learning, edited by ann nowe.*

Sibi, P., Jones, S. A., & Siddarth, P. (2013). Analysis of different activation functions using back propagation neural networks. *Journal of Theoretical and Applied Information Technology*, *47*(3), 1264–1268.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Snatzke, R. G. (2002). Exhaustive search in the game amazons. *More Games of No Chance*, *42*, 261.

Song, J., & Muller, M. (2015). An enhanced solver for the game of amazons. *Computational Intelligence and AI in Games, IEEE Transactions on*, *7*(1), 16–27.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, *3*(1), 9–44.

Tegos, T. (2002). *Shooting the last arrow.* Unpublished doctoral dissertation, University of Alberta.