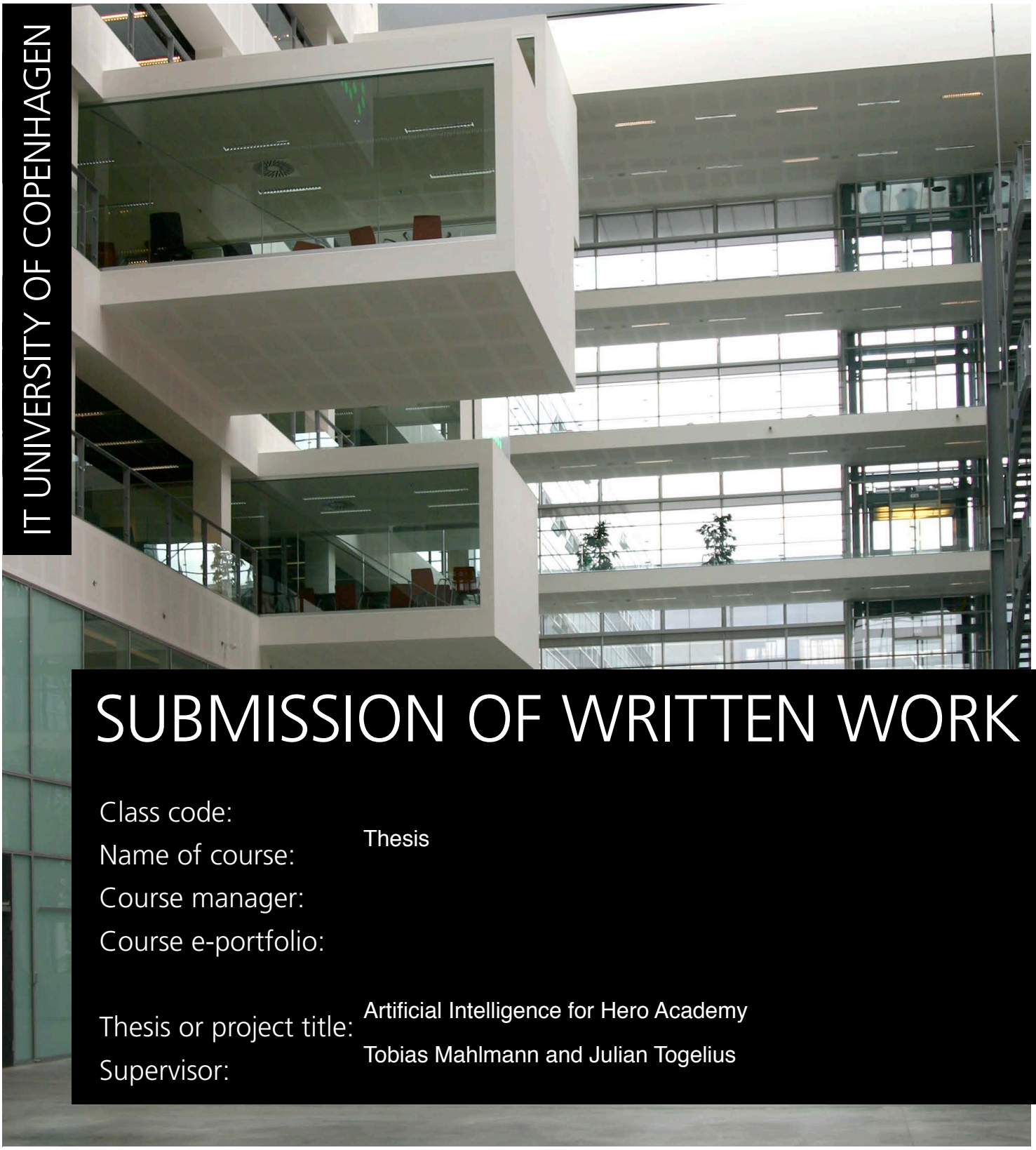
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:

Name of course: Thesis

Course manager:

Course e-portfolio:

Thesis or project title: Artificial Intelligence for Hero Academy

Supervisor: Tobias Mahlmann and Julian Togelius

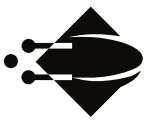
	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Niels Orsleff Justesen	13/01-1989	noju@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Artificial Intelligence for Hero Academy

Niels Justesen

Master Thesis

Advisors: Tobias Mahlmann and Julian Togelius
Submitted: June 2015



IT University
of Copenhagen

Abstract

In many competitive video games it is possible for players to compete against a computer controlled opponent. It is important that such opponents are able to play at a worthy level to keep players engaged. A great deal of research has been done on Artificial Intelligence (AI) in games to create intelligent computer controlled players, more commonly referred to as *AI agents*, for a large collection of game genres. In this thesis we have focused on how to create an intelligent AI agent for the tactical turn-based game Hero Academy, using our own open source game engine *Hero AIcademy*. In this game, players can perform five sequential actions resulting in millions of possible outcomes each turn. We have implemented and compared several AI methods mainly based on Monte Carlo Tree Search (MCTS) and evolutionary algorithms. A novel progressive pruning strategy is introduced that significantly improves MCTS in Hero Academy. Another approach to MCTS is introduced, in which the exploration constant is set to zero and greedy rollouts are used, that also gives significant improvement. An online evolutionary algorithm that evolves plans during each turn achieved the best results. The fitness function of the evolution is based on depth-limited rollouts to determine the value of plans. It did, however, not increase the performance significantly. The online evolution agent was able to play Hero Academy competitively against human beginners but was easily beaten by intermediate and expert players. Aside from searching for possible plans it is critical to evaluate the outcome of these intelligently. We evolved a neural network, using NEAT, that outperforms our own manually designed evaluator for small game boards, while more work is needed to obtain similar results for larger game boards.

Acknowledgements

I would like to thank my supervisors Tobias Mahlmann and Julian Togelius for our many inspiring discussions and their continuous willingness to give me feedback and technical assistance. I also want to thank Sebastian Risi for his lectures in the course *Modern AI in Games* as they made me interested in the academic world of game AI.

Contents

Chapter 1

Introduction

Ever since I was introduced to video games, my mind has been baffled by the question: "How can a computer program play games?" The automation and intelligence of game-playing computer programs have kept me fascinated since and have been the primary reason why I wanted to learn programming. Game-playing programs, or Artificial Intelligence (AI) agents as they are called, can allow a greater form of interaction with the game system e.g. by letting players play against the system or as a form of assistance during a game. During my studies at the IT University of Copenhagen I have had the opportunity to explore the underlying algorithms of such AI agents providing me with answers to my question. Today I am not only interested in the methods but also for which types of games it is possible, using state of the art methods, to produce challenging AI agents. While the current state of research in this area can be seen as a stepping stone towards something even more intelligent, it also serves as a collection of methods and results that can inspire game developers in the industry. Exploring the limits and different approaches in untested classes of games is something I believe is important for both the game industry and the field. My interest in this question has previously led me to explore algorithms for the very complex problem of controlling units during combat in the real-time strategy game *StarCraft*, which turned into a paper for the IEEE Conference on Computational Intelligence and Games in 2014 [1]. This thesis will have a similar focus of exploring AI algorithms for a game with a very high complexity, but this time for the Tactical Turn-Based (TTB) game Hero Academy. Before explaining why TTB games have caught my interest, a very brief overview of Artificial Intelligence (AI) in games is presented.

Hereafter I will argue why TTB games including Hero Academy is an unexplored type of game in the game AI literature.

1.1 AI in Games

The fields of *Artificial Intelligence* and *Computational Intelligence* in games are concerned with research in methods that can produce intelligent behavior in games. As there is a great overlap between the two fields and no real agreement on when to use which term, *game AI* will be used throughout this thesis to describe the two joined fields.

AI methods can be used in many different areas of the game development process, which have formed several different sub-fields within game AI. Ten sub-fields were recently identified and described by Togelius and Yannakakis [2]. One popular sub-field is *Procedural content generation*, in which AI methods are used to generate content in games such as weapons, storylines, levels and even entire game descriptions. Another is *Non-player character (NPC) behavior*, which is concerned with methods used to control characters within a game. In this thesis we will only be concerned with methods that are used to control an agent to play a game. This sub-field is also called *Games as AI benchmarks*. Research in this sub-field is relevant for other sub-fields within game AI as they can be used for automatic play-testing and content generation by simulating the behavior of human players. Intelligent AI agents can also be used as worthy opponents in video games, which is essential for some games. Game AI methods can also be applied to many real-world problems related to planning and scheduling, and one could say that games are merely a sand box in which AI methods can be developed and tested before it is applied to the real world. On the other hand, games are a huge industry, where AI still has a lot of potential and unexplored opportunities, especially when it comes to intelligent game-design assistance tools.

To get a brief overview of the field of game AI, let us take a look at which type of games researchers have focused on. This is far from a complete list of games, but simply an effort to highlight the different classes of games that are used.

Programming a computer to play Chess has been of interest since Shannon introduced the idea in 1950 [3]. In the following decades a

multitude of researchers helped the progress of Chess playing computers, and finally in 1997 the Chess machine Deep Blue beat the world chess champion Garry Kasparov [4]. Chess playing computer programs implement variations of the minimax search algorithm that in a brute force style makes an exhaustive search of the game tree. One of the most important discoveries in computer Chess has been the Alpha-beta pruning algorithm that enables the minimax search to ignore certain branches in the tree.

In 2007 Schaeffer et al. announced that the game *Checkers* was solved [5]. The outcome of each possible opening strategy was calculated for when no mistakes were made by both players. Since the game tree was only partly analyzed, the game is only *weakly solved*. Another game that recently has been weakly solved is *Heads-up limit hold'em Poker* [6], which requires an immense amount of computation due to the stochastic nature of the game. Most interesting games are, however, not solved, at least not yet, and may perhaps never be solved due to their complexity.

After Chess programs reached a super-human level, the classic board game Go has become the most dominant benchmark game for AI research. The number of available actions during a turn in Go is much larger than in Chess and creates problems for the Alpha-beta pruning algorithm. The average number of available actions in a turn is called the *branching factor* and is an important complexity measure for games. Another challenge in Go is that game positions are very difficult to evaluate, which is essential for a depth-limited search. Monte Carlo Tree Search (MCTS) has greatly influenced the progress of Go-playing programs. It is a family of search algorithms that uses stochastic simulations as a heuristic and iteratively expands the tree in the most promising direction. These stochastic simulations have turned out to be very effective when it comes to evaluation of game positions in Go. Machine learning techniques in combination with pattern recognition algorithms have enabled recent Go programs to compete with human experts. The computer Go web site keeps track of matches played by expert Go players against the best Go programs[1]. MCTS has shown to work well in a variety of games and has been a popular choice for researchers working with AI for modern board games such as *Settlers of Catan* [7], *Carcassonne* [8] and *Dominion* [9].

A lot of game AI research have also been made on methods applied to real-time video games. While turn-based games are slow and allow

[1]http://www.computer-go.info/h-c/index.html2013

AI agents to think for several seconds before taking an action, real-time games are fast-paced and often require numerous actions within each second of the game. Among popular real-time benchmark games are: the racing game *TORCS*, the famous arcade game *Ms. PacMan* and the platformer *Super Mario Bros*. Quality reverse-engineered game clones exist for these games and several AI competitions have been held using these clones to compare different AI methods.

Another very popular game used in this field is the real-time strategy (RTS) game *StarCraft*. Since this game requires both high-level strategic planning as well as low-level unit control, it offers a suite of problems for AI researchers and has also been subject for several AI competitions that are still ongoing. Few open source RTS games such as *WarGus* (a clone of *WarCraft II*) and *Stratagus* have also been used as benchmark games.

The general video game playing competition has been popular the last years, where researchers and students develop AI agents that compete in unseen two-dimensional arcade games [10]. A recent approach to general video game playing has been to evolve artificial neural networks to play a series of Atari games, which even surpassed human scores in some of the games [11]. Evolutionary computation has been very influential to game AI as interesting solutions can be found when mimicking the evolutionary process seen in nature.

The mentioned games in this section can be put into two categories. The first is turn-based games and the second is real-time games. The branching factors of the mentioned turn-based games is around 10 to 300 while some real-time games have extremely high branching factors. Prior the work presented in this thesis it became apparent to me that very little work has been done on AI in turn-based games with very large branching factors. Most turn-based games have been board games, where players perform one or two actions each turn. A popular class of turn-based games, where players take multiple actions each turn, is TTB games. These games can have branching factors in the thousands and even millions and will be the focus in this thesis as they, to my knowledge, seem like a unexplored domain in the field of game AI. Figure 1.1 shows how TTB games are unique from some of the other game genres. TTB games are a genre that seems to always lie in the category of turn-based games with high branching factors. Some board games, such as *Risk*, and turn-based strategy video games, such as *Sid Meier's Civilization*, do also belong to this category while they may not be classified as

TTB games. Other dimensions are of course important when categorizing games by their complexity such as hidden information, randomness and amount of rules. Still, this figure should express the kind of games that I believe needs more attention.

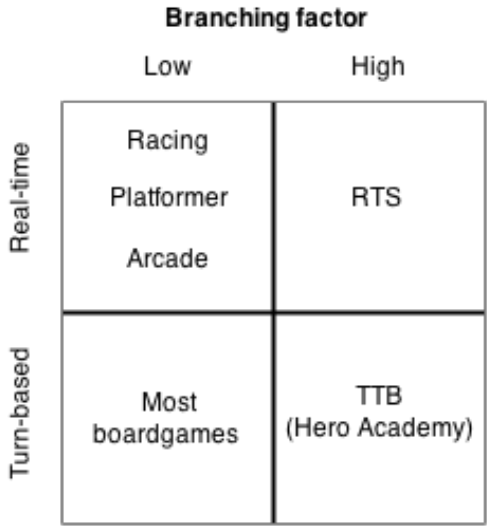


Figure 1.1: A few game genres categorized after their branching factor and whether they are turn-based or real-time. Tactical Turn-based games, including Hero Academy, is among Turn-based games with very high branching factors.

Next section will explain a bit more about TTB games and give concrete examples of some published TTB games.

1.2 Tactical Turn-based Games

In strategy games players must make long term plans to defeat their opponent, and it requires tactic manoeuvres to obtain their objectives. Strategy is extremely important in RTS games as players produce buildings and upgrades that will have a long term effect on the game. One sub-genre of strategy games is Tactical Turn-based (TTB) games. In these games the continuous execution in each turn is extremely critical. Moving a unit to a wrong position can easily result in a lost game. These games are often concerned with small-scale combats instead of series of battles and usually have a very short cycle of rewards. Strategy games are mostly implemented as real-time games, while tactical games are

more suited as turn-based games. Modern games such as *Blood Bowl*, *X-Com* and *Battle for Wesnoth* are all good examples of TTB games. These games have very large branching factors since players have to make multiple actions each turn. I also refer to such games as *multi-action* games in contrast to *single-action* games like Chess and Go. It would be ignorant to say that these games do not require any strategy, but the tactical execution is simply much more important. One very interesting digital TTB game is *Hero Academy* as its branching factor is in the millions, and it has a very short cycle of rewards. This game was chosen as the benchmark TTB game for this thesis and a thorough introduction to the game rules are given in the following chapter.

Among other TTB games that have been studied in game AI research is *Advance Wars*. This game has a much larger game world than Hero Academy and requires more strategic choices such as unit production and complex terrain analysis. Hero Academy is thus a more focused test bed as it is mostly concerned with tactical decisions. Bergsma and Sprock [12] designed a two-layered influence map that is merged using an evolved neural network to control units in Advance Wars. The influence map was used to identify the most promising squares to either move to or attack.

1.3 Research Question

In the previous sections I argued why TTB games are an under-explored type of game and why I think it is important to focus on these. The game *Hero Academy* was chosen as the benchmark TTB game for this thesis and the goal will be to explore AI methods for this game and examine the achieved playing level compared to human players. One focused research question was made based on this goal:

> **Research question**
> How can we design an AI agent that is able to challenge human players in Hero Academy?

In this introduction I have referred to myself as *I*, but throughout this thesis *we* will be used instead, even though the work presented was solely made by me, the author.

Chapter 2

Hero Academy

Hero Academy is a two-player Tactical Turn-Based (TTB) video game developed by Robot Entertainment. It was originally released on iOS in 2012 but is now also available on Steam, Android and Mac. The game is a multi-action game as players have five action points each turn they can spend to perform actions sequentially. It is played asynchronous typically over several days but are played in one sitting during tournaments. It was generally well received by the critics[1] and hit the top 10 list of free games in China within just 48 hours[2].

2.1 Rules

The rules of Hero Academy will be explained as if it was a board game, as it simply is a digital version of game that could just as well be released physically. This will hopefully make it easier to understand the rules for people familiar with board game mechanics.

Hero Academy is played over a number of rounds until one player have lost both crystals or all units. The game is played on a game board of 9x5 squares containing two deploy zones and two crystals for each player. Both players have a deck of 34 cards from which they draw cards onto their secret hand. In the beginning of each round cards are drawn until the maximum hand size of six is reached or the deck is empty. This is the only element in the game with hidden information

[1] http://www.metacritic.com/game/pc/hero-academy

[2] http://www.gamasutra.com/view/news/177767/How_Hero_Academy_went_big_in_China.php

and randomness. Everything else is deterministic and visible to both players. The graphical user interface in the game does actually not visualize these elements as playing cards but they do work mechanically equivalent. Cards can either represent a unit, an item or a spell. The initial game board does not contain any units but will gradually be filled as more rounds are played.

In Hero Academy players control one of six different teams: Council, Dark Elves, Dwarves, The Tribe, Team Fortress and The Shaolin. In this thesis we will only focus on the Council team which is the first team players learn to play in the tutorial. The game offers several game boards with different features. Again, only one will be used for this thesis (see Figure 2.1). Features, mechanics and rules described in this chapter will thus only be those relevant to the selected game board and the Council team.

Figure 2.1: The selected game board with the following square types on (x,y): Player 1 deploy zones (1,1) and (1,5) , player 1 crystals (2,4) and (3,2), assault square (5,5), defense square (5,1), power square (3,3) and (7,3), player 2 crystals (7,2) and (8,4). Image is from the Hero Academy Strategy blog by Charles Tan (http://heroacademystrategy.blogspot.dk/). Some squares are not described in this introduction, but a can be looked up in the blog by Charles Tan.

2.1.1 Actions

Players take turn back and forth until the game is over. During each turn players have 5 Action Points (AP) they can spend on the actions

described below. Each action costs 1 AP and are allowed to be performed in any order and even several times each turn. It is also allowed to perform several actions with the same unit.

Deploy

A unit can be deployed from a players hand onto an unoccupied deploy zone they own. Deploying units simply means that a unit card on the player's hand is discarded and the unit represented on the card is placed on the selected deploy zone.

Equip

One unit on the board can be equipped with an item from the hand. Units cannot carry more than one of each item and it is not possible to equip opponent units. Similar to the deploy action, the item card is discarded and the item represented on the card is given to the selected unit.

Move

One unit can be moved a number of squares equal to or lower than its *Speed* attribute. Diagonally moves are not allowed. Units can, however, jump over any number of units along their path as long as the final square is unoccupied.

Attack

One unit can attack an opponent unit within the number of squares equal to its *Attack Range* attribute. The amount of damage dealt is based on numerous factors such as the *Power* attribute of the attacker, which items the attacker and defender holds, which squares they stand on and the resistance of the defender. There are two types of attacks in the game: *Physical Attack* and *Magical Attack*. Likewise, units can have *Physical Resistance* and *Magical Resistance* that protects them against those attacks. The defender will in the end lose health points (HP) equal the calculated damage. If the HP value of the defender reaches zero the defender will become *Knocked Out*. Knocked Out units cannot perform any actions and will be removed from the game if either a unit moves onto its square (called *Stomping*) or if it is still knocked out by the end of the owners following turn.

Special

Some units have special actions such as healing. These are described later when each unit type is described.

Cast Spell

Each team have one unique spell that can be cast onto a square on the board from the hand, where after the spell card is discarded. Spells are very powerful and usually saved until a very good moment in the game.

Swap Card

Cards on the hand can be shuffled into the deck in hopes of drawing better cards in the following round.

2.1.2 Units

Each team has four different types of basic units and one so called *Super Unit*. In this section a short description of each of the five units on the Council team is presented. Units normally have Power 200, Speed 2, 800 HP and no resistances. Only the specialities of each unit are highlighted below.

Figure 2.2: The five different units on the Council team. From left to right: Archer, cleric, knight, ninja and wizard.

Archer

Deals 300 physical damage within range 3. Archers are very good at knocking out enemy units in one turn with their long range and powerful attack.

Cleric

As a special action the cleric can heal friendly units within range 2. Healed units gain HP equal to three times the power of the cleric. Knocked out units can also be revived using this action but then only gains two times the power in HP. The cleric deals 200 magical damage within range 2 and has 20% Magical Resistance. It is critical to have at least one cleric on the board to be able to heal and revive units.

Knight

With 1000 HP and 20% Physical Resistance the knight is a very tough unit. It deals 200 physical damage within range 1 and knocks back enemy units one square, if possible, when attacking. The Knight is able to hold and conquer critical positions on the board as it is very difficult to knock out in one turn.

Ninja

The super unit of the Council team. As a special action the ninja can swap positions with any other friendly unit that is not knocked out. The ninja deals 200 physical damage within range 2 but deals double damage when attacking at range 1. Additionally, the ninja has speed 5. This unit is very effective as it is both hard hitting and allows for more mobility on the board.

Wizard

The wizard deals 200 magical damage within range 2. Furthermore, its attack makes two additional chain attacks if enemy units or crystals stand next to the target. The game guide says that the the chain attacks are random but the actual deterministic behavior have been described by Hamlet in his own guide[3].

2.1.3 Items and Spells

Each team has different items that can be given to units on the board using the equip action, and one spell that can be cast onto the board with the cast spell action. This section will briefly describe the five different items and the spell of the Council team.

Figure 2.3: The items and spell of the Council team. From left to right: Dragonscale, healing potion, runemetal, scroll, shining helmet and inferno.

[3]http://iam.yellingontheinternet.com/2012/08/10/hero-academy-mechanics-guide/#elves

Dragonscale (item)
Gives +20% Physical Resistance and +10% maximum HP.

Healing Potion (item)
Heals 1000 HP or revives a knocked out unit to 100 HP. This item is used instantly and then removed from the game.

Runemetal (item)
Boosts the damage dealt by the unit with 50%.

Scroll (item)
Boosts the next attack by the unit with 300% where after it is removed from the game.

Shining Helmet (item)
Gives +20% Physical Resistance and +10% HP.

Inferno (spell)
Deals 350 damage to a 3x3 square area. If units in this area are already knocked out, they are instantly removed from the game.

The council team starts with 3 archers, 3 clerics, 3 knights, 1 ninja, 3 wizards, 3 dragonscales, 2 healing potions, 3 runemetals, 2 scrolls, 3 shining helmets and 2 infernos in their deck.

2.2 Game Engine

Robot Entertainment have, as most other game companies, not published the source code for their games and no open source clones existed prior to this thesis to our knowledge. To be able to perform experiments in the game Hero Academy, we have developed our own clone. The development of this clone was initiated prior to this thesis, but the quality was continuously improved while it was used, and several features have been added. This Hero Academy clone have been named *Hero AIcademy* and is written in Java.

Hero AIcademy only implements the Council team and the square types from the game board on Figure 2.1. Besides that, all rules have been implemented. The main focus of Hero AIcademy have been on allowing AI agents to play the game rather than on graphics and animations. In this section a brief overview of how AI agents interact with the engine is presented. The complete source code of the engine as of

1st of June 2015 is provided on this GitHub page[4] and the continued development can be followed on the official Hero AIcademy GitHub page[5].

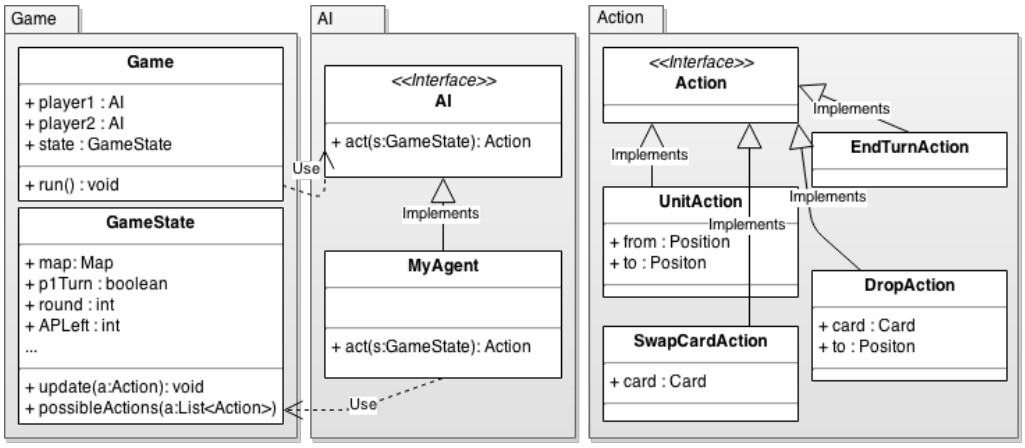


Figure 2.4: Simplified class diagram of Hero AIcademy

The Game class is responsible for the game loop and repeatedly requests actions from the agents in the game. The act(s:GameState) method, that must be implemented by all agents, is given a clone of the game state held by the Game class. Using this game state, agents can call the possibleActions(a:List<Action>) on the GameState object to abtain a list of available actions. The Game class also calls a UI implementation during the game loop that uses some of the graphics from the real game. Human players can interact with the interface using the mouse.

2.3 Game Complexity

This section will aim to analyse the game complexity of Hero Academy. The complexity of a game is important knowledge when we want to apply AI methods to play it, as some methods only works with games of certain complexities. The focus here is on the game-tree complexity, including the average branching factor, and the state-space complexity. The estimated complexity will throughout this section be compared to the complexity of Chess.

[4]https://github.com/njustesen/hero-ai
[5]https://github.com/njustesen/hero-aicademy

Figure 2.5: The user interface of Hero AIcademy.

2.3.1 Possible Initial States

In order to estimate how many possible games of Hero Academy that theoretically can be played, we must know first know the number of possible initial states. Before the game begins each player draws six cards from their deck. The deck is shuffled which makes the starting hands random. This makes 5,730 possible starting hands[6] and $5,730^2 = 32,832,900$ possible initial states by taking both players starting hand into account. In Hero Academy the starting player is decided randomly thus doubling the number to $65,665,800$. This is, however, still only when one game board and one team is used.

2.3.2 Game-tree Complexity

A game tree is a directed graph where nodes represent game states and edges represent actions. Leaf nodes in a game tree represent a terminal state, i.e. when the game is over. The game-tree complexity is

[6]Calculated using http://www.wolframalpha.com/ by searching "subsets of size 6 {a,a,a,c,c,c,k,k,k,n,w,w,w,d,d,d,p,p,r,r,r,s,s,h,h,h,i,i}"

determined by the number of leaf nodes in the entire game-tree which is also the number of possible games. Studying the game-tree complexity of a game is interesting as many AI methods use tree search algorithms to determine the best action in a given state. Tree search algorithms will obviously be able to search through small game trees very fast, while some very large trees will be impractical to search in.

The branching factor of a game tells us the how many children each node has in the game tree. In most games this number is not uniform throughout the tree and we are thus interested in the average. The branching factor thus tells us the average number of available moves during a turn. Since players in Hero Academy are performing actions in several sequential steps, we will first calculate the branching factor of a step.

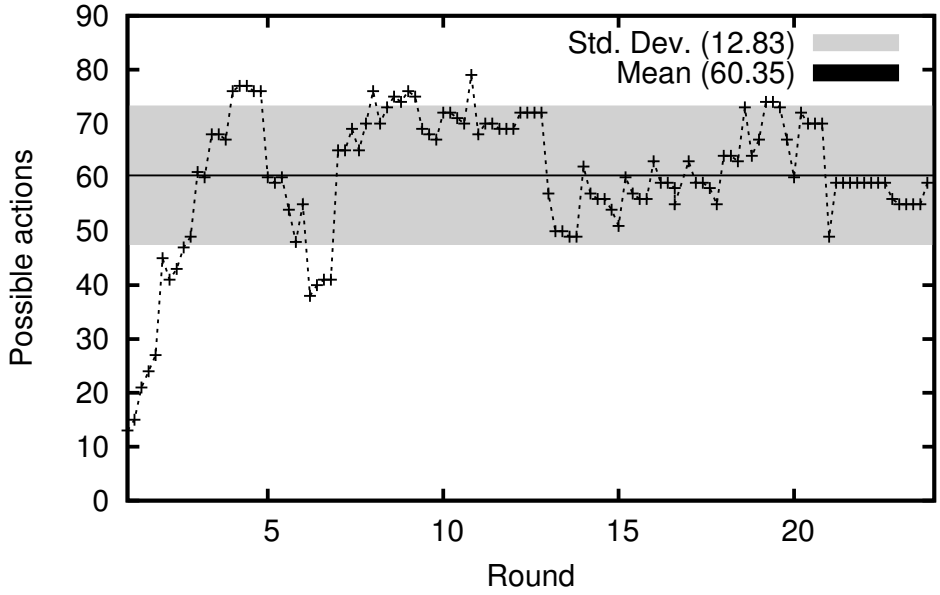


Figure 2.6: Available actions for player 2, playing the Council team, during a recorded game session on YouTube (https://www.youtube.com/watch?v=OMtWTv_Tf4s) uploaded by the user Elagatua on 01/20/2014. We manually counted the number of available actions and the numbers may contain minor errors. Actions using the inferno spell, which would not have dealt any damage, were not counted.

By manually counting the number of possible actions throughout a recorded game on YouTube (see 2.6) we can estimate the average branching factor to be somewhere around 60. The results from 24 games in an ongoing tournament called *Fast Action Swiss Tournament*

(FAST)[7] were collected and these games had an average length of 42 rounds. Nine of the games ended in a resignation before the game actually ended.

With an average branching factor of 60 for one step, a turn with five actions will have $60^5 = 7.78 \times 10^8$ possible variations. One round where both players take turn will have $(60^5)^2 = 6.05 \times 10^{17}$ possible variations. As a comparison Chess has a branching factor of around 35 for a turn and the difference really shows the complexity difference of single-action and multi-action games. A lower bounds of the game-tree complexity can finally be estimated by raising the number of possible variations of one round to the power of the average game length. In this estimation the stochastic events of drawing cards is however left out. Using the numbers from above we end up with a game-tree complexity of $((60^5)^2)^{40} = 1.82 \times 10^{711}$.

Given an initial game state in Hero Academy there are thus a minimum of 1.82×10^{711} possible ways the game can be played. Remember that these calculations are still ignoring the complexities of randomness during the game. The game-tree complexity of Chess was calculated by Shannon in a similar way to be 10^{120} [3].

2.3.3 State-space Complexity

The state-space complexity is the number of legal game states that is actually possible to reach. It is not trivial to calculate this number for Hero Academy and the aim of this section will be to find the number of possible board configurations by only considering units and not items. The game board has 45 squares. Two of these squares are deploy zones owned by the opponent and up to four squares are occupied by crystals. Let us only look at the situation where all crystals are on the board with full health. On this board there are 39 possible squares to place a unit. Hereafter, there are 38, then 37 and so on. Using this idea we can calculate the number of possible board configurations for n units with the following function:

$$conf(n) = \prod_{i=1}^{n}(39 - i + 1)$$

[7] http://forums.robotentertainment.com/showthread.php?5378-Fast-Action-Swiss-Tournament-(FAST)

For $n = 26$, the situation where all units are on the board, $conf(26) = 3.28 \times 10^{36}$. A lot of these configurations will in fact still be identical since placing an archer on (1,1) and then another archer on (2,2) is the same as first placing an archer on (2,2) and then one on (1,1). This however becomes insignificant when we also consider the fact that units have different HP values. I.e. the first archer we place might have 320 HP and the second might have 750. Most units have a maximum HP value of 800, but knights have 1000 and the shining helmet gives +10% HP. For simplicity 800 will be used here for any unit. Adding HP to the function gives us the following:

$$conf\_hp(n) = \prod_{i=1}^{n}((39 - i + 1) \times 800)$$

For $n = 26$, the situation where all units are on the board, $conf\_hp(26) = 9.90 \times 10^{111}$. This is the number of possible board configurations with all 26 units on the board, also considering HP values, but still without items. The number of possible board configurations for any number of units is calculated by taking the product of $conf\_ho(26)$, $conf\_ho(25)$, $conf\_ho(24)$ and so on:

$$conf\_all = \prod_{n=0}^{26} conf\_hp(n)$$

If we calculate $conf\_all$ we will get the number 1.57×10^{199} and it seems pointless to try reaching a more precise number. Since the board configuration is only one part of the game state and items are not considered, the state-space complexity of Hero Academy is thus much larger than this number. As a comparison Chess has a state-space complexity of 10^{43}.

Chapter 3

Related work

Games are a very popular domain in the field of AI, as they offer an isolated and fully understood environment that are easy to reproduce for testing. Because of this, thousands of research papers have been released in this field offering numerous algorithms and approaches to AI in games. In this chapter some of the most popular algorithms are presented that are relevant to the game Hero Academy. Each section will give an introduction to a new algorithm or method which are followed by a few optimization methods that seem relevant when applied to Hero Academy.

3.1 Minimax Search

Minimax is a recursive search algorithm that can be used as a decision rule in two-player zero-sum games. The algorithm considers all possible strategies for both players and selects the strategy that minimizes the maximum loss. In other words, minimax picks the strategy that allows the opponent to gain the least advantage in the game. The minimax theorem that establishes, that there exists such a strategy for both players, was proven by John von Neumann in 1928 [13].

In most interesting games, game trees are so large that the minimax search must be limited to a certain depth in order to reach a result within a reasonable amount of time. An evaluation function, also called a *heuristic*, is then used to evaluate the game state when the depth-limit is reached. E.g. in Chess a simple evaluation function could count the number of pieces owned by each player and return the difference.

In 1997 the Chess Machine called Deep Blue won a six-game match against World Chess Champion Garry Kasparov. Deep Blue was running a parallel version of minimax with a complex evaluation function and a database of games with grandmasters [4].

3.1.1 Alpha-beta pruning

The alpha-beta pruning algorithm is an optimization of minimax, as it stops the evaluation of a node if it is found to be worse than an already searched move. The general idea of ignoring nodes in a tree search is called pruning. An example of alpha-beta pruning is shown on Figure 3.1, where two nodes are pruned because the min-player can select an action that leads to a value of 1, which is worse that the minimax-value of already visited sub-trees. The max-player should never go in that direction and thus the search at that node can stop. Alpha-beta pruning is thus able to increase the search depth while it is guaranteed to find the same minimax value for the root node [14].

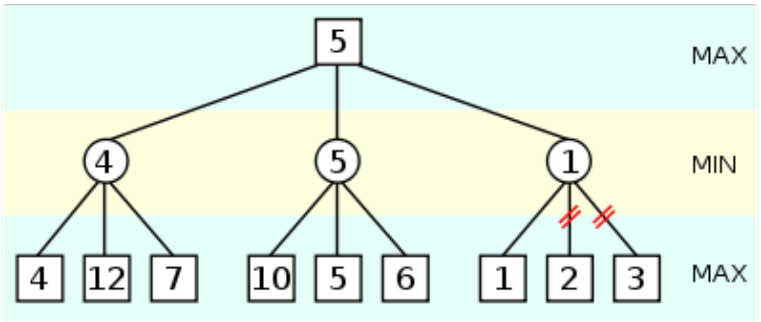


Figure 3.1: The search tree of the alpha-beta pruning algorithm with a depth-limit of two plies. Two branches can be pruned in the right-most side of the tree shown by two red lines. Figure is from Wikipedia[1].

3.1.2 Expectiminimax

A variation of minimax called *expectiminimax* is able to handle random events during a game. The search works in a similar way but some nodes, called *change nodes*, will have edges that correspond to random events instead of actions performed by a player. The minimax value of a

[1]http://fr.wikipedia.org/wiki/%C3%89lagage_alpha-beta

change node is the sum of all its childrens probilistic values, which are calulated by multiplying the probability of the event with its minimax value. Expectiminimax can be applied to games such as Backgammon. The search tree is, however, not able to look many turns forward because of the many branches in change nodes. This makes Expectiminimax less effective for complex games.

3.1.3 Transposition Table

A transposition is a sequence of moves that results in a game state that could also have been reached by another sequence of moves. An example in Hero Academy would be to move a knight to square (2,3) and then a wizard to square (3,3), where the same outcome can be be achieved by first moving the wizard to square (3,3) and then the knight to square (2,3). Transpositions are very common in Chess and result in a game tree containing a lot of identical sub trees. The idea of introducing a transposition table is to ignore entire sub trees during the minimax search. When an already visited game state is reached, it is simply given the value that is stored in the transposition table. Greenblatt et al. was the first to apply this idea to Chess [15], and it has since been an essential optimization in computer Chess. A transposition table is essentially a hash table with one entry for each unique game state that has been encountered. Various methods for creating hash codes from a game state in Chess exists with the most popular being Zobrist hashing which can be used in other board games as well, e.g. Go and Checkers [16]. Since most interesting games, including Chess and Hero Academy, have a state space complexity much larger than we can express using a 64-bit integer, also known as a *long*, some game states share hash codes even though they are in fact different. When such a pair of game states are found, a so-called *collision* occurs but are usually ignored since they are very rare.

3.2 Monte Carlo Tree Search

The Alpha-beta algorithm fails to succeed in many complex games or when it is difficult to design a good evaluation function. This section will describe another popular tree search method that are useful when alpha-beta falls short. Monte Carlo Tree Search (MCTS) is a family of iterative tree search methods that balance randomized exploration of

the search space with focused search in the most promising direction. Additionally, its heuristic is based on game simulations and thus does not need a static evaluation function. MCTS was formalized as a framework by Chaslot et al. in 2008 [17] and has since shown to be effective in many games. Most notably it has revived the interest of computer Go, as the best of these programs today implement MCTS and are able to compete with Go experts [18]. One advantage of MCTS over alpha-beta is that it merely relies on its random sampling where alpha-beta must use a static evaluation function. Creating evaluation functions for games such as Go can be extremely difficult and thus makes alpha-beta very unsuitable. Another key feature is that MCTS is able to search deep in promising directions while ignoring obvious bad moves early. This makes MCTS more suitable for games with large branching factors. Additionally, MCTS is *anytime*, meaning that it at any time during the search can return the best action found so far.

MCTS iteratively expands a search tree in the most urgent direction, where each iteration consists of four phases. These are depicted on Figure 3.2. In the *Selection phase* The most urgent node is recursively selected from the root using a *tree policy* until a terminal or unexpanded node is found. In the *Expansion phase* one or more children are added if the selected node is non-terminal. In the *Simulation phase* a simulated game is played from an expanded node. This is also called a *rollout*. Simulations are carried out in a so called *forward model* that implements the rules of the environment. In the *Backpropagation phase* the result of the rollout is backpropagated in the tree and the value and visit count of each node are updated.

The tree policy is responsible for balancing exploration over exploitation. One solution would be to always expand the search in the direction that gives the best values, but the search would then easily oversee more potent areas of the search space. The Upper Confidence Bounds for Trees (UCT) algorithm solves this problem with the UCB1 formula [19]. When it has to select the most urgent node amongst the children of a node it tries to maximize:

$$\text{UCB1} = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where $\overline{X}_j$ is the average reward gained by visiting the child, n is the visit count of the current node, n_j is the visit count of the child

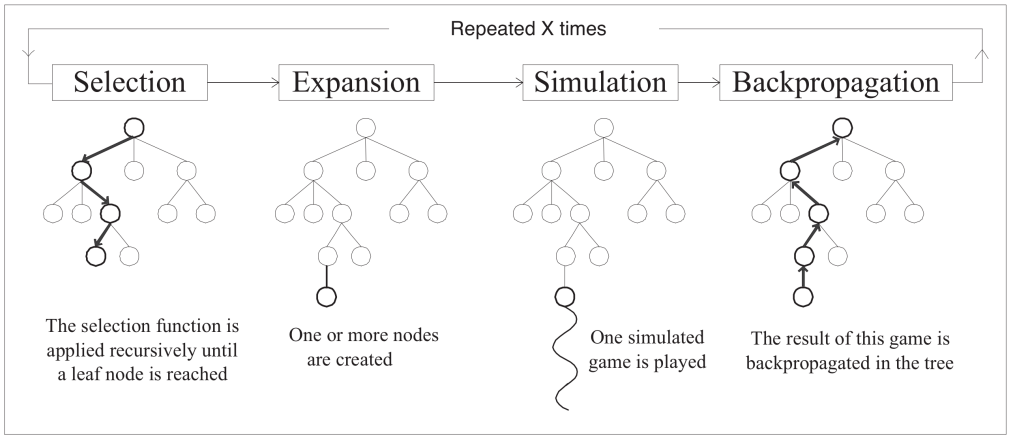


Figure 3.2: The four phases of the MCTS algorithm. Figure is from [17].

j and C_p is the exploration constant used to determine the amount of exploration which varies between domains. We will continue to refer to this algorithm as MCTS, even though the actual name is UCT, when it implements the UCB1 formula.

MCTS became one of the main focus points in this thesis and thus a great deal of time was spend on enhancements for this algorithm. The number of variations and enhancements that exists for MCTS far exceeds what was possible to implement and test in this thesis. We have thus aimed to only focus on enhancements that enables MCTS to overcome large branching factors as these will be relevant to Hero Academy. These enhancements are presented in the following sections.

3.2.1 Pruning

Pruning obvious bad moves can in many cases optimize an MCTS implementation when dealing with large branching factors. However, a great deal of domain knowledge is required to determine whether moves are good or bad. Two types of pruning exists [20]. *Soft pruning* is when moves are initially pruned but may later be added to the search and *Hard pruning* is when some moves are entirely excluded from the search. An example of soft pruning is the *Progressive Unpruning/Widening* technique which was used to improve the Go playing program Mango [21] and MoGo [22]. Next section will describe these progressive strategies, as the main concept was used in our own MCTS implementations.

3.2.2 Progressive Strategies

The concept of *progressive strategies* for MCTS was introduced by Chaslot et al. [21] as they described two of such strategies called *progressive bias* and *progressive unpruning*. With progressive bias the UCT selection function (see the original in Section 3.2) is extended to the following:

$$\text{UCB1}_{\text{pb}} = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}} + f(n_j)$$

where $f(n_j)$ adds a heuristic value when n_j is low. In this way heuristic knowledge is used to guide the search as long as the rollouts produce a reliable result. Chaslot et al. chose $f(n_j) = \frac{H}{n_j+1}$ where H is the heuristic value of the game state j.
The other progressive strategy is progressive unpruning, where a node's branches are first pruned using a heuristic and then later progressively unpruned as its visit count increases. A very similar approach introduced by Coulom called *progressive widening* was shown to improve the Go program *Crazy Stone* [23]. The two progressive strategies both individually improved the play of the Mango program in the game Go, but combining both strategies produced the best result.

3.2.3 Domain Knowledge in Rollouts

The basic MCTS algorithm does not include any domain knowledge other than its use of the forward model. Nijssen showed that using pseudo random move generation in the rollouts could improve MCTS in Othello [24]. During rollouts moves were not selected uniformly but better moves, determined by a heuristic, were preferred and selected more often.

Another popular approach is to use *ϵ-greedy* rollouts that select the best action determined by a heuristic with probability ϵ and otherwise select a random action. This heuristic can either be implemented manually or learned. ϵ-greedy rollouts was shown to improve MCTS in the game Scotland Yard [25].

3.2.4 Transpositions

As for minimax, a transposition table can also improve the performance of MCTS in some domains. If a game has a high number of transposi-

tions, it is more likely that introducing a transposition table will increase the playing strength of MCTS. Méhat et al. tested MCTS in single-player games from the General Game Playing competition and showed that transposition tables improved MCTS in some games while no improvement was observed in others [26].

Usually, the search tree in MCTS contains nodes which have direct references to their children. In order to handle transpositions in MCTS the search tree is often changed to also contain edges, since several nodes can have multiple parents [27]. This changes the tree into a Directed Acyclic Graph (DAG). During the expansion phase a node is only created if it represents an unvisited game state. To save memory and computation, game states are transformed into hash codes. A transposition table is used where each entry holds the hash code of a game state and a reference to its node in the DAG. If the hash code of a newly explored game state already exists in the transposition table an edge is simply created and pointed to the already existing node. In this way identical sub trees are ignored. Backpropagation is simple when dealing with a tree but with a DAG several methods exist [27]. One method is to update the descent path only. This method is very simple to implement and is very similar to how back propagation is done in a tree.

3.2.5 Parallelization

Most devices today have multiple processor cores. Since AI techniques often require a high amount of computation, utilizing multiple processors efficiently with parallelization can increase the playing strength significantly. MCTS can be parallelized in three different ways [28]:

Leaf parallelization
: This parallelization method is by far the simplest to implement. At each leaf multiple concurrent rollouts are performed. MCTS thus runs single-threaded in the selection, expansion and backpropagation phases but multi-threaded in the simulation phase. This method simply improves the precision of evaluations and may be able to identify more promising moves faster.

Root parallelization
: This method builds multiple MCTS trees in parallel, that in the end are merged. Very little communication is needed between threads, which also makes this method simple to implement.

Tree parallelization

Another method is to use one shared tree, where several threads simultaneously traverse and expand it. To enable access to the tree by multiple threads, *mutaxes* can be used as locks. A simpler method called *virtual loss* adds a high number of losses to a node in use, so no other threads will go in that direction.

Chaslot et al. compared these three methods in 13x13 Go and used a *strength-speedup* measure to express how much improvement each methods added [28]. All their experiments were performed using 16 processor cores. A strength-speedup measure of 2.0 means that the parallelization method plays as well as a non-parellelized method with double as much computation time. The leaf parallelization method was the weakest with a strength-speedup of only 2.4, while root parallelization gained a strength-speedup of 14.9. Tree parallelization gained a strength-speedup of 8.5 and remains the most complicated of the three methods to implement.

3.2.6 Determinization

In stochastic games, and games with hidden information, MCTS can use determinization to transform a game state into a deterministic one with open information. Another more naive approach would simply be to create a new node for each possible outcome due to randomness or hidden information, but this will often result in extremely large trees. Cazenave used determinization for MCTS in the game Phantom Go [29], which is a version of Go with hidden stones, by guessing the positions of the opponent stones before each rollout.

3.2.7 Large Branching Factors

MCTS has been successful in many other games than Go. *Amazons* is a game with a branching factor above 1,000 during the 10 first moves in the game. Using depth-limited rollouts with an evaluation function, among other improvements, Kloetzer was able to show that MCTS outperforms traditional minimax-based programs in this game [30].

Kozelek applied MCTS to the game *Arimaa* but only achieved a weak level of play [31]. Arimaa is a very interesting game as it allows players to make four consecutive moves in the same way as in Hero Academy.

The branching factor of Arimaa was calculated to be 17,281 by Haskin[2], which is very high compared to single-turn games but still a lot lower than the 7.78×10^8 in Hero Academy (estimated in Section 2.3.2). One important difference between Arimaa and Hero Academy is that pieces in Arimaa never get eliminated, which makes it extremely difficult to evaluate positions. Hero Academy is in contrast more like Chess as both material and positional evaluations can be made. Kozelek showed that short rollouts with a static evaluation function improved MCTS in Arimaa. Transposition tables were also shown to improve the playing strength significantly, while *Rapid Action Value Estimation* (RAVE), an enhancement often used in Go, did not.

Kozelek distinguished between a step-based and a move-based search. In a step-based search each ply in the search tree corresponds to moving one piece one time, where one ply in a move-based search corresponds to a sequence of steps resulting in one turn. Kozelek preferred the step-based search mainly because of its simplicity and how it with ease can handle transpositions within turns. Interestingly, a move-based search was used by Churchill et al. [32] and later Justesen et al. [1] in the RTS game StarCraft to overcome the large branching factor. The move-based searches in StarCraft is done by sampling a very low number of possible moves and thus ignoring most of the search space. Often heuristic strategies are among the samples of moves together with randomly generated moves. In Arimaa it is difficult to generate such heuristic strategies, and thus the move-based approach is likely to be weak in this game. It seems to be a necessity in StarCraft to use the move-based approach, since the search must find a solution within just 40 milliseconds, and in this game we do know several heuristic strategies. The move-based search would simply ignore very important moves in Arimaa that the step-based search is more likely to find. While no comparison of these approaches in either of the games exist, it seems that the step-based approach is unlikely to succeed in real-time games with very large branching factors. The branching factor of combats in StarCraft is around 8^n, where n is the number of units under control, which easily surpasses the branching factor of Arimaa.

[2]http://arimaa.janzert.com/bf_study/

3.3 Artificial Neural Networks

We have now looked at two methods for searching in the game tree and one more will be introduced when evolutionary computation is introduced later. It seems clear, that good methods for evaluating game states is an important asset to the search algorithms, as dept-limited searches are popular in games with large branching factors. Creating an evaluation function for Hero Academy seems straight forward since material on the board, on the hand and in the deck is easy to count. The positional value of units, and how to balance the pursuit towards the two winning conditions in the game (destroying all units or all crystals), are however two very difficult challenges. Instead of relying on expert knowledge and a programmers ability to implement this, such evaluation functions can be learned, and one method to store such learned knowledge is in *artificial neural networks* or just neural networks. Before explaining how the such knowledge can be learned, let us look a what neural networks are.

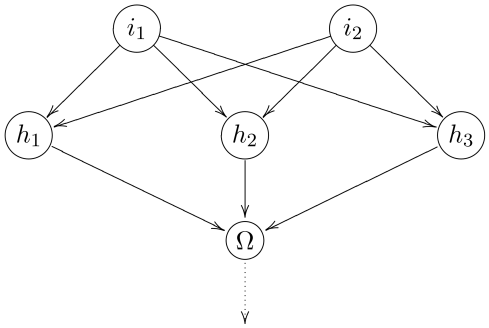


Figure 3.3: An artificial neural network with two input neurons (i_1, i_2), three one-layered hidden neurons (h_1, h_2, h_3) and one output neuron (Ω). Figure is from [33].

Humans are very good at identifying patterns such as those present in game states. An expert game player is able to quickly match these patterns with how desirable they are and from that make solid decisions during the game. One popular choice is to encode such knowledge into neural networks, inspired by how a brain works. A neural network is a set of connected neurons. Each connection links two neurons together and has a weight parameter expressing how strong the connection is, mimicking synapses that connect neurons in a real brain. *Feedforward*

networks are among the most popular classes of neural networks and are typically sorted in three types of layers: an input layer, one or more hidden layers and an output layer (see Figure 3.3).

When a feedforward neural network is activated, values from the input layer are iteratively sent one step forward in the network. The value v_j received by a neuron j is usually the sum of the output o_i from each ingoing neuron multiplied with the weight $w_{i,j}$ of its connection. This is also called the *propagation function*:

$$v_j = \sum_{i \in I} (o_i \times w_{i,j})$$

where I is the set of ingoing neurons. After v_j is computed, it is passed through a so-called *activation function* in order to normalize the value. A popular activation function is the *Sigmoid* function:

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

where e is a constant determining how much the values are squeezed towards 0 and 1.

When all values have been sent through the network, the output of the network can be extracted directly from the output neurons. A neural network can thus be seen as a black box, as it is given a set of inputs and returns a set of outputs, while the logic behind the computations quickly gets complex. This propagation of values simulates how signals in a brain are passed between neurons. The input layer can be seen as our senses, and the output layer is our muscles that reacts to the input we get. Artificial neural networks are of course just a simplification of how an actual brain functions.

The output of a network solely depends on its design, hereunder the formation of nodes and connections, called the topology, and the connection weights. Using supervised learning, and given a set of desired input and output pairs, it is possible to backpropagate the errors to correct connection weights and thus gradually learn to output more correct results [34]. We will not go through the math of the backprogragation method as it has not used in this thesis. One problem with the backpropagation learning method is that it only works for a fixed topology and determining which topology to use can be difficult when dealing with complex problems. Another problem is that, for supervised learning to work it requires a training set of target values. A popular solution

is to evolve both the weights and the topology through evolution which will be described in the section about neuroevolution (see Section 3.4.3), where examples of successful applications also are presented. To understand neuroevolution, let us first get an overview of the mechanics and use of evolution in algorithms, as this is a widely used approach in game AI.

3.4 Evolutionary Computation

Evolutionary computation is concerned with optimization algorithms inspired by the mechanisms of biological evolution such as genetic crossover, mutation and the notion *survival of the fittest*. The most popular of these algorithms are Genetic Algorithms (GA) first described by Holland [35]. In GAs a population of candidate solutions are initially created. Each solution, also called the *phenotype*, has a corresponding encoding called a *genotype*. In order to optimize the population of solutions it goes through a number of generations, where each individual is tested using a fitness function. The least fit individuals are replaced by offspring of the most fit. Offspring are bred using crossover and/or mutation. In this way promising genes from fit individuals stay in the population, while genes from bad individuals are thrown away. Like evolution in nature, the goal is to evolve individuals consisting of genes that make them strong in their environment. Such an environment can be a game and the the solution can be a strategy or a neural network functioning as a heuristic.

3.4.1 Parallelization

Three different methods for parallelizing GAs were identified by Tomassini et al. [36] and are briefly presented in this section. In most GA implementations, calculating the fitness function is the most time consuming part, and thus an obvious solution is to run the fitness function concurrently for multiple individuals. Another simple method is to run several isolated GAs in parallel, either with the same or different configurations. GAs can easily end up in a local optima and by running multiple of these concurrently, there is a higher change of finding either the global optima or at least several local ones. In the *island model* multiple isolated populations are evolved in parallel, but at a certain frequency promising individuals will migrate to other populations to

disrupt stagnating populations. Another approach called the *grid model* places individuals in one large grid where individuals in parallel interact only with their neighbors. In such a grid the weakest individual in a neighborhood will be replaced by an offspring of the other neighbors. Figure 3.4 shows the different models used when parallelizing GAs.

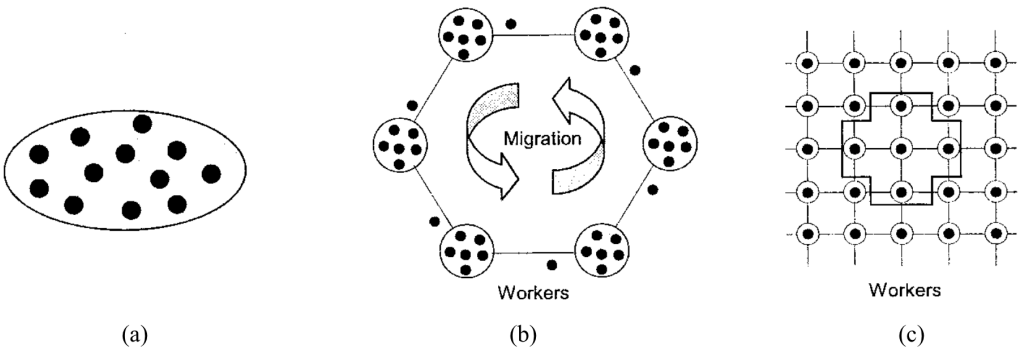


Figure 3.4: The different models used in parallel GAs. (a) shows the standard model where only one population exists and every individual can mate with any other. In this model the fitness function can run on several threads in parallel. (b) shows the island model where several populations exist and individuals can migrate to neighboring populations. (c) shows the grid model where individuals are placed in one large grid and only interacts with their neighbors. Figure taken from [37]

3.4.2 Rolling Horizon Evolution

Evolutionary computation is usually used to train agents in a training phase before they are applied and used in the actual game world. This is called *offline learning*. Online evolution in games, on the other hand, is an evolutionary algorithm that is applied while the agent is playing (i.e. online) to continuously find actions to perform.

Recent work by Perez on agents in real-time environments has shown that GAs can be a competitive alternative to MCTS [38]. He tested the Rolling Horizon Evolutionary (RHE) algorithm for the Physical Traveling Salesman (PTS) problem that is both real time and requires a long sequence of actions to reach the goal. RHE is a GA, where each individual corresponds to a sequence of actions and are evaluated by performing its actions in a forward model and finally evaluating the outcome with an evaluation function. The algorithm continuously evolves plans for a limited time frame, while it acts in the world. RHE and MCTS achieved more or less similar results in the PTS problem.

This approach is of course interesting since Hero Academy is all about planning several actions ahead and may be an interesting alternative to MCTS. There is, however, the difference that Hero Academy is adversarial, i.e. each player takes turn, where the PST problem is a single player environment where every outcome in the future is predictable. In Hero Academy we are not able to evolve actions for the opponent, since we only have full control in our own turn, and the evolution is then limited to plan five actions ahead.

3.4.3 Neuroevolution

Earlier in Section 3.3 we mentioned how neural networks can be used as a heuristic, i.e. a game state evaluation function. One way to train such a network is to use supervised learning by backpropagating errors, but for this approach we need a large training set e.g. created from games logs with human players. Since Hero Academy is played online the developers should have access to such game logs, while we do not. Several unsupervised learning methods exist where learning happens by interacting with the environment. Temporal Difference (TD) learning is a popular choice when it comes to unsupervised learning and was used with success in 1959 by Samuel in his Checkers-playing program [39]. Samuel's program used a database of visited game states used to store the learned knowledge, while this is impractical for games with a high state-space complexity. Tesauro solved this problem in his famous Backgammon-playing program *TD-Gammon* that reached a super-human level [40] by using the TD(λ) algorithm to train a neural network. TD(λ) backpropagates results of played games through the network for each action taken in the game and gradually corrects the weights. The λ parameter is used to make sure that early moves in the game are less responsible for the outcome, while later moves are more. The topology must however still be determined manually when applying TD(λ). In the rest of this section the focus will be on methods that also evolves the topology.

Neuroevolution is a machine learning method combining neural networks and evolutionary algorithms and has shown to be effective in many domains. The conventional neuroevolution method has simply been to maintain a fixed topology and evolve only the weights of the connections. Chellapilla et al. used such an approach to evolve the

weights of a neural network, that was able to compete with expert players in Checkers [41]. Chellapilla evolved a feedforward network with 32 input neurons, 40 neurons in the first hidden layer and 10 neurons in the second hidden layer.

An early attempt to evolve topologies as well as weights are the *marker-based encoding* scheme which, inspired by our DNA, has a series of numeric genes representing nodes and connections. The marker-based encoding seems to have been replaced by superior approaches developed later. Moriarty and Miikkulainen was, however, able to discover complex strategies for the game Othello using the marker-based encoding scheme [42]. Their game playing networks had 64 output neurons each expressing how strongly a move to that corresponding square is considered. Othello is played on a 8x8 game board similar to Chess.

TWEANN (Topology & Weight Evolving Artificial Neural Network) algorithms, are also able to evolve the topology of neural networks. The marker-based encoding scheme described above is an early TWEANN algorithm. Among the most popular TWEANN algorithms today is the NeuroEvolution of Augmenting Topologies (NEAT) algorithm by Stanley and Miikkulainen [43]. NEAT introduced a novel method of crossover between different topologies by tracking genes with innovation numbers. Each gene represents two neurons and a connection. Figure 3.5 shows how *add connection* and *add node* mutations change such gene strings.

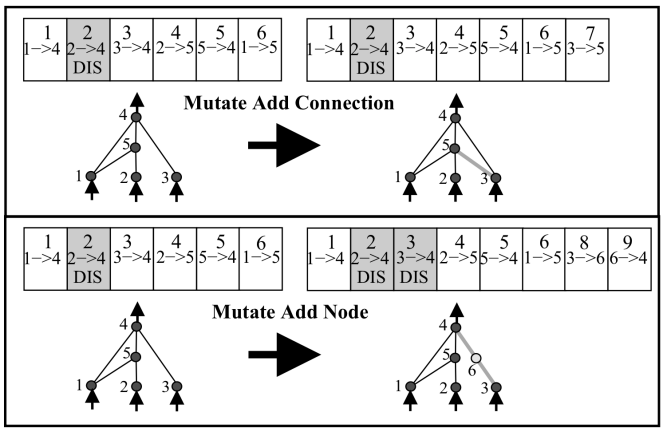


Figure 3.5: Examples of add connection and add node mutations in NEAT and how genes are tracked with innovation numbers shown in the top of each gene. Genes become disabled if a new node is added in between the connection. Figure is from [43] that also shows examples of crossover.

The use of innovation numbers also made crossover very simple as single or multi-point crossover easily can be applied to the genes.
When new topologies appear in a population they very rarely achieve a high fitness value, as they often need several generations to improve their weights and perhaps change their topology further. To protect such innovations from being excluded from the population, NEAT introduces *speciation*. Organisms in NEAT primarily compete for survival with other organisms of the same specie and not with the entire population. Another feature of NEAT, called *complexification*, is that organisms are initialised uniformly without any hidden neurons and then slowly grow and become more complex. The idea behind this is to only grow networks if the addition actually provides an improvement. NEAT has been successful in many real-time games such as Pac-Man [44] and The Open Racing Car Simulator (TORCS) [45]. Jacobs compared several neuroevolution algorithms for Othello and was most successful with NEAT [46], which also was the only TWEANN algorithms in the comparison.

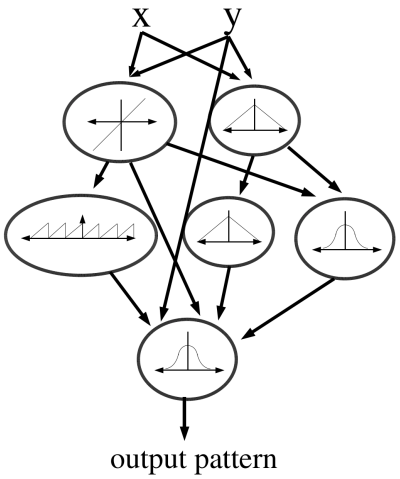


Figure 3.6: Example of a Compositional Pattern Producing Network (CPPN) that takes two input values x and y. Each node represents a functions that alters the value as it is passed trough the network. Figure is from [47].

While NEAT uses a *direct encoding*, a relatively new variation called Hypercube-based NEAT (HyperNEAT) uses an *indirect encoding* called a Compositional Pattern Producing Network (CPPN), which essentially

is a network of connected functions. An example of a CPPN is shown on Figure 3.6. In HyperNEAT it is thus the CPPNs that are evolved which is then used to distribute the weights to a fixed network topology. The input layers forms a two-dimensional plane and thus becomes dimensionally aware. This feature is advantages when it comes to image recognition as neighboring pixels of an image are highly related to each other. Risi and Togelius argues that this dimensional awareness can be used to learn the symmetries and relationships on a game board [48], but to our knowledge only few attempts have been made and with limited success.

Recent work on evolving game playing agents with neuroevolution for Atari games have shown, that NEAT was the superior of several compared methods, and it was even able to beat human scores in three of the games [11]. The networks were fed with three layers of input. One with the raw pixel data, another with object data and the one with random noise. The HyperNEAT method was however shown to be the only useful method when only the raw pixel layer was used.

Chapter 4

Approach

This chapter will explain our approach of implementing several AI agents for Hero Academy using the theory from the previous chapter. This have resulted in agents with very diverse behaviors. This chapter will first describe some main functionalities that were used the agents followed by a description of their implementation and enhancements.

4.1 Action Pruning & Sorting

The Hero AIcademy API offers the method possibleActions (a:List<Action>) that fills the given list with all available actions of the current game state. This method simply iterates each card on the hand and each unit on the game board and collects all possible actions for the given card or unit. Several actions in this set will produce the same outcome and can thus be pruned. A pruning method were implemented that removes some of these actions and thus performs *hard pruning*.

First, identical swap card actions are pruned, which are seen when several identical cards are on the hand. Next, a cast spell action is pruned if another cast spell action exists that can hit the same units. E.g. a cast spell action that targets the wizard on square (1,1) and the archer on (2,1) can be pruned, when another cast spell action exists that can hit both the wizard, the archer and the knight on (2,2). The targets of the second action is thus a super set of the targets of the first. It is not guaranteed that the most optimal action survives this pruning, but it is a good estimate as it is based on both the amount of damage it inflicts and which units it can target. If a cast spell action has no targets on the

game board, it is also pruned.

A functionality to sort actions were also implemented based on a naive heuristic that analyses the current game state and rates each action based on some rules. The following rules were used that give a value v based on the action type:

Cast spell
$v = |T| \times 300 - 500$ where T is the set of targets.

Equip (revive potion)
If $u_{hp} = 0$ **then** $v = u_{maxhp} + |I| \times 200$ **else** $v = u_{maxhp} - u_{hp} - 300$ where u_{hp}, u_{maxhp} and I is the health points, maximum health points and the set of items of the equipped unit respectively, thus preferring to use a revive potion on units with items and low health.

Equip (scroll, dragonscale, runemetal & shining helmet)
$v = \frac{u_{power} \times u_{hp}}{u_{maxhp}}$, thus preferring to equip units if they already are powerful and have full health.

Unit (attack)
If *stomping* **then** $v = d_{maxhp} \times 2$ **else** $v = u_{power}$ where d_{maxhp} is the maximum health of the defender. 200 is added to v if the attacker is a wizard due to its chain attack.

Unit (heal)
If $u_{hp} = 0$ **then** $v = 1400$ **else** $v = u_{maxhp} - u_{hp}$.

Unit (move)
If *stomping* **then** $v = d_{maxhp}$ **else if** the new square is a power, defense or assault square $v = 30$ **else** $v = 0$.

For all other actions $v = 0$. This action sorting functionality is used by several of the following agents.

4.2 State Evaluation

A state evaluation function is used as a heuristic by several of the implemented search algorithms. Experiments with several evaluation func-

tions were performed, but one with a great amount of domain knowledge showed the best results throughout the experiments. This evaluation function named HeuristicEvaluation in the project source code will be described in this section.

The value of a game state is measured by the difference in health points of remaining units both on the hand, in the deck and on the game board by each player. The health point value of a unit is however multiplied by several factors. The final value v of a unit u on the game board is:

$$v = u_{hp} + \underbrace{u_{maxhp} \times up(u)}_{\text{standing bonus}} + \overbrace{eq(u) \times up(u)}^{\text{equipment bonus}} + \underbrace{sq(u) \times (up(u) - 1)}_{\text{square bonuse}}$$

where $up(u) = 1$ if $u_{hp} = 0$ and $up(u) = 2$ if $u_{hp} > 0$. $eq(u)$ adds a bonus to units carrying equipment. E.g. 40 points are added if an archer carries a scroll while a knight is given -40 points for the same item, since scrolls are much more useful on an archer. These bonuses can be seen in Table 4.1. $sq(u)$ adds bonuses to units on special squares which can bee seen in Table 4.2.

	Dragonscale	Runemetal	Shining helmet	Scroll
Archer	30	40	20	50
Cleric	30	20	20	30
Knight	30	-50	20	-40
Ninja	30	20	10	40
Wizard	20	40	20	50

Table 4.1: Bonus added by the HeuristicEvaluation to units with items.

Units on the hand and in the deck are given the value $v = u_{maxhp} \times 1.75$, which is usually lower that values given to units on the game board. This makes a game state with many units on the board more valuable. Infernos are given $v = 750$ and potions $v = 600$, which makes these items expensive to use.

	Assault	Deploy	Defense	Power
Archer	40	-75	80	120
Cleric	10	-75	20	40
Knight	120	-75	30	30
Ninja	50	-75	60	70
Wizard	40	-75	70	100

Table 4.2: Bonus added by the HeuristicEvaluation to units on special squares.

4.3 Game State Hashing

A function that transforms a game state into a 64 bit integer was required by the implemented transposition tables as it reduces the memory requirements significantly. Clever solutions such as Zobrist hashing were not implemented, because game states in Hero Academy are much more complex, and this makes the implementation non-trivial. Instead a more classic Java hashing approach was applied, which can bee seen on below. A relatively high prime number was used to decrease the number of collisions when only small changes are made to the game state. No collision tests were, however, made to prove that this worked.

```
public long hashCode() {
  final int prime = 1193;
  long result = 1;
  result = prime * result + APLeft;
  result = prime * result + turn;
  result = prime * result + (isTerminal ? 0 : 1);
  result = prime * result + p1Hand.hashCode();
  result = prime * result + p2Deck.hashCode();
  result = prime * result + p2Hand.hashCode();
  result = prime * result + p1Deck.hashCode();

  for (int x = 0; x < map.width; x++)
    for (int y = 0; y < map.height; y++)
      if (units[x][y] != null)
        result = prime * result + units[x][y].hash(x, y);

  return result;
}
```

Listing 4.1: GameState.hashCode()

4.4 Evaluation Function Modularity

Several algorithms were implemented in this project using a game state evaluation function. To increase the modularity an interface called IStateEvaluator was made with the method signature double eval(s:GameState, p1:boolean) that must be implemented to return the game state evaluation for player 1 if $p1 = true$ and for player 2 if $p1 = false$. Two important implementations of this interface are the HeuristicEvalutor (described in Section 4.2) and the RolloutEvaluator with the following contructor: RolloutEvaluator(rolls:int, depth:int, policy:AI, evaluator:IStateEvaluator). Again, any IStateEvaluator can be plugged into this implementation, which is use when a terminal node, or its depth-limit, is reached. Any AI implementation can in a similar way be used as the policy for the rollouts. Additionally, it can be depth limited to *depth* and multiple sequential rollouts equal to *rolls* will be performed whenever it is invoked.

4.5 Random Search

Two agents that play randomly were implemented. They were used in the experiments as baseline agents, but also as policies in rollouts. The first random agent simply selects a random action with a uniform probability from the set of actions returned by possibleActions(a:List<Action>). This agent will be refered to as *RandomUniform*. The policies used in rollouts should be optimized to be as fast as possible. Instead of identifying all possible actions, another method is to first make a meta-decision about what kind of action to perform. It will decide whether to make a unit action or a card action and thus simplify the search for actions. This approach is implemented in the agent that will be referred to as *RandomMeta*. If *RandomMeta* decides to take a unit action, it will traverse the game board in a scrambled order until a unit under control is found, where after it will search for possible actions just for that unit and return one randomly.

An ϵ-greedy agent was also implemented. At some probability ϵ it will pick the first action after action sorting (see Section 4.1), imitating a greedy behavior. Otherwise it will simply pick a random action. This agent is often used as policy in rollouts by some of the following agents.

4.6 Greedy Search

Greedy algorithms always take the action that gives the best immediate outcome. Two different greedy algorithms were implemented which are described in this section.

4.6.1 Greedy on Action Level

GreedyAction is a simple greedy agent that makes a one-ply search, where one ply corresponds to one action, and uses the HeuristicEvaluator as a heuristic. It also makes use of action pruning to avoid bad use of the inferno card. The Hero AIcademy engine is used as a forward model to obtain the resulting game states. *GreedyAction* is used as a baseline agent in the experiments and will be compared to the more complex implementations.

4.6.2 Greedy on Turn Level

GreedyTurn is a more complex greedy agent that makes a five-ply search which corresponds to one turn of five actions. The search algorithm used is a recursive depth-first search very similar to minimax and will also produce the exact same results, given the same depth-limit. During this search, the best action sequence is stored together with the heuristic value of the resulting game state. The stored action sequence will be returned when the search is over. This agent will find the most optimal sequence of actions during its turn, assuming the heuristic leads to optimal play. Since the heuristic used is mostly based on the health point difference between the two players, it will most likely lead to a very aggressive play. It will, however, not be able to search the entire space of actions within the six second time budget we have given the agents in the experiments. In Section 2.3.2 we estimated the average branching factor of a turn to be 7.78×10^8 which is the average number of moves this search will have to consider to make an exhaustive search. Three efforts were made to optimize the performance of the search: action pruning is applied at every node of the search, a transposition table is used as well as a simple parallelization method.

A transposition table was implemented to reduce the size of the search tree. The key of each entry is the hash code of a game state. The value of each entry is actually never used since already visited nodes are simply ignored.

Parallelization was implemented by first sorting the actions available at the root level and then assigning them one by one to a number of threads equal to the number of processors. Each thread is then responsible for calculating the value of the assigned action. The threads use the same shared transposition table that can only be accessed by one thread at the time.

In order to make this search an *anytime* algorithm, threads are simply denied access to more actions when the time budget is used and the best found action sequence is finally returned.

4.7 Monte Carlo Tree Search

The MCTS algorithm was implemented with an action based approach, meaning that one ply in the search tree corresponds to one action and not one turn. The search tree thus has to reach a depth of five to reach the beginning of the opponents turn. The search tree is constructed using both node and edge objects to handle parallelization methods. Both nodes and edges have a visit count, while only edges hold values. Action pruning and sorting are applied at every expansion in the tree, and only one child are added in each expansion phase. A child node will not be selected until all its siblings are added. The UCB1 formula were changed to the following to handle nodes and edges:

$$\text{UCB1}_{\text{edges}} = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{e_j}}$$

where e_j is now the visit count of the edge e, going from node n, instead of the child node. For the backpropagation the descent-path only approach was used because of its simplicity. In order to handle two players with multiple actions, the backpropagation implements an extension of the BackupNegamax method (see Algorithm 1). This backpropagation algorithm is called with a list of edges corresponding to the traversal during the selection phase, a Δ value corresponding to the result of the simulation phase and a boolean $p1$ which is true if player one is the max player and false if not.

A transposition table was added to MCTS with game state hash codes as keys and references to nodes as values. When an already

Algorithm 1 Backpropagation for MCTS in multi-action games

```
1:  procedure BackupMultiNegamax(Edge[] T, Double Δ, Boolean p1)
2:      for Edge e in T do
3:          e.visits++
4:          if e.to is not null then
5:              e.to.visits++
6:          if e.from is root then
7:              e.from.visits++
8:          if e.p1 = p1 then
9:              e.value+=Δ
10:         else
11:             e.value-=Δ
```

visited game state is reached during the expansion phase, the edge is simply pointed to the existing node instead of creating a new.

Both leaf parallelization and root parallelizaion were implemented. A LeafParallelizer was made that implements the IStateEvaluator interface, and is set up with another IStateEvaluator, that will be cloned and distributed to a number of threads. The root parallelization is, in contrast, implemented as a new AI implementation and works as a kind of proxy for several concurrent MCTS agents. The root parallelized MCTS merges the root nodes from the concurrent threads and then returns the best action from this merged tree.

AI agents in Hero AIcademy must return one action at a time until their turn is over. In the experiments performed, each algorithm were given a time budget b to complete each turn and thus two approaches came apparent. The first approach is simply to spend $\frac{b}{5}$ time on each action while the second approach will spend b time on finding the entire action sequence and thereafter perform each one by one. The second approach was used to make sure the search will reach a decent depth. This approach is actually used by all the search algorithms.

4.7.1 Non-explorative MCTS

Three novel methods are introduced to handle very large game trees in multi-action games. The first and simplest method is to use an exploration constant $C_p = 0$ in combination with deterministic rollouts using

a greedy heuristic as policy. It might seem irrational not to do any exploration at all, but since all children are expanded and evaluated at least once, a very controlled and limited form of exploration will occur. Additionally, as the search progresses into the opponents turn, counter-moves may be found that will force the search to explore other directions. The idea is thus to only explore when promising moves become less promising. A non-explorative MCTS is not guaranteed to converge towards the optimal solution, but instead tries to find just one good move that the opponent cannot counter. The idea of using deterministic rollouts guided by a greedy heuristic, might be necessary when no exploration happens, since most nodes are visited only once. It would have a huge impact if an action is given a low value due to one unlucky simulation.

4.7.2 Cutting MCTS

Because of the enormous search space, MCTS will have trouble reaching a depth of ten plies during its search. This is critical since actions will be selected based on vague knowledge about the possible counter-moves by the opponent, that takes place between ply five and ten. To increase the depth in the most promising direction, a *progressive pruning strategy* called *cutting* is introduced. The cutting strategy will remove all but the most promising child c from the tree after $\frac{b}{a}$ time has past, where b is the time budget and a is the number of actions allowed in the turn (five in Hero Academy). When $2 \times (\frac{b}{a})$ time has past all but the most promising child of c are removed from the tree. This is done continuously a times down the tree. The downside of this cutting strategy is that there is no going back when actions have been cut. This can lead to an action sequence in Hero Academy that will make a unit charge towards an opponent and then retreat back again in the same turn. This should however be much better than charging forward to leave a unit unprotected as a result of insufficient search for counter-moves. An example of the cutting approach in three steps is shown on Figure 4.1.

4.7.3 Collapsing MCTS

Another progressive pruning strategy that deals with the enormous search space in multi-action games is introduced called the *collapsing* strategy. When a satisfying number of nodes K are found at depth d, where d is equal to the number of actions in a turn, the tree is collapsed

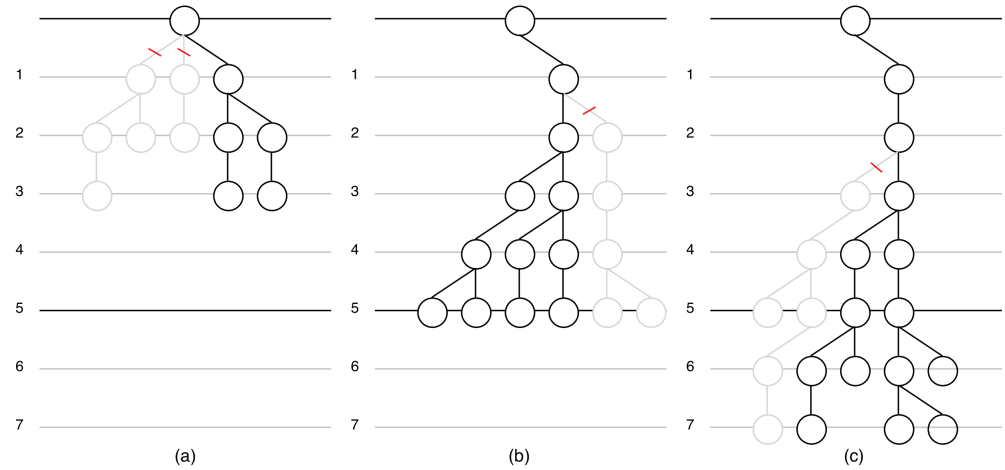


Figure 4.1: An example of how nodes are progressively cut away using the cutting strategy in three steps to force the search to reach deeper plies. (a) After some time all nodes except the most promising in ply one are removed from the tree. (b) The search continues where after nodes are removed from ply two in a similar way. (c) Same procedure but one ply deeper.

so that nodes and edges between depth 1 and d not leading to a node at depth d are removed from the tree. An example is shown on Figure 4.2 where $d = 5$. No children are added to nodes that have previously been fully expanded.

The purpose of the collapsing strategy is to stop exploration in the first part of the tree and focus on the next part. The desired effect in Hero Academy is, that after a number of possible action sequences are found the search will explore possible counter-moves.

4.8 Online Evolution

Genetic algorithms seem to be a promising solution when searching in very large search spaces. Inspired by the rolling horizon evolution algorithm an online evolutionary algorithm was implemented, where depth-limited rollouts are used as the fitness function. A solution (phenotype) in the population is a sequence of actions that can be performed in the agents first turn. A fitness function that rates a solution by evaluating the resulting game state, after the sequence of actions are performed, would be very similar to the *GreedyTurn* agent, but possibly more efficient. Instead, the fitness function is using rollouts with a depth-limit

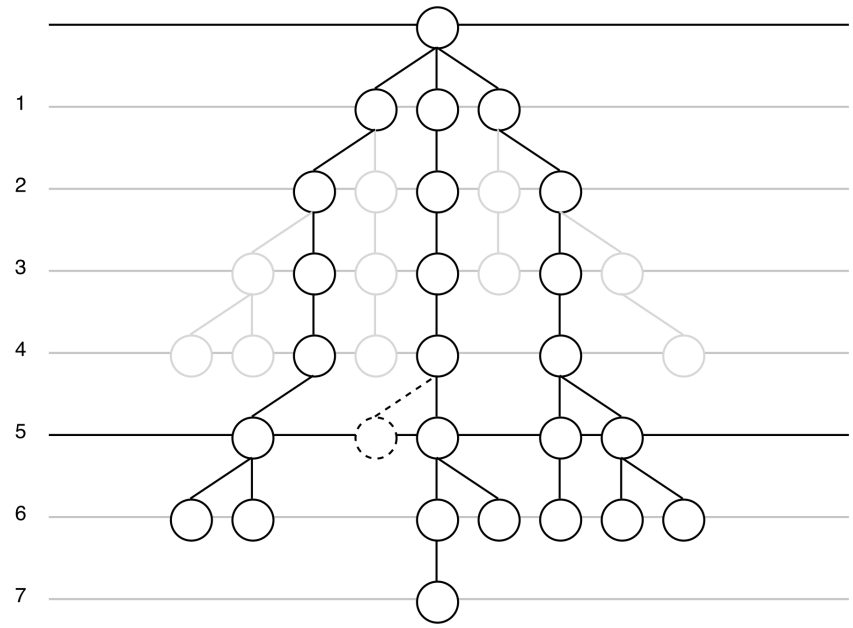


Figure 4.2: Grayed out nodes are removed due to the collapsing method, which is set up to collapse when five nodes are found in depth 5. The newly expanded node that initiates this collapse is shown with dashed lines.

of one turn to also include the possible counter-moves available in the opponents turn. The resulting game states reached by the rollouts are here after evaluated using the HeuristicEvaluator.

Because the rollouts take place only in the opponents turn, the minimum value of several rollouts are used instead of the average. In this way, individuals are rated by the worst known outcome of the opponents turn. This will imitate a two-ply minimax search as the rollouts are minimizing and the genetic algorithm is maximizing. If an individual survives for several generations, the fitness function will regardless of the existing fitness value continue to perform rollouts. It will, however, only override the existing fitness value if a lower one is found. The fitness value is thus converging towards the actual value, which is equal to the worst possible outcome, as it is tested in each generation. Individuals that are quickly found to have a low value will, however, be replaced by new offspring. The rollouts use the ϵ-greedy policy and several ϵ values are tested in the experiments.

An individual might survive several generations before a good counter-move is found that will decrease its fitness value, resulting in

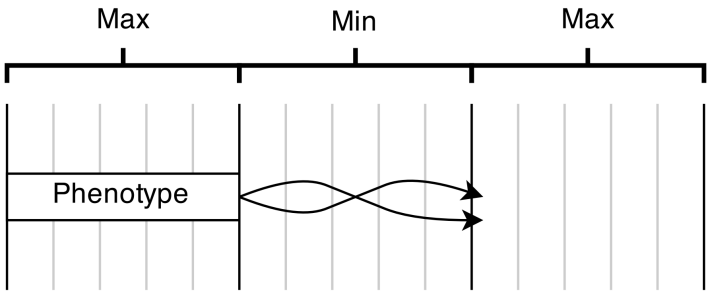


Figure 4.3: A depiction of how the phenotype spans five actions, i.e. the agents own turn, and the rollouts in the fitness function spans the opponents turn.

the solution to be replaced in the following generation. To avoid that such a solution should re-appear later and consume valuable computation time, a table is introduced to store obtained fitness values. The key of each entry is the hash code of the game state reached by a solutions action sequence, and the value is the minimum fitness value found for every solution resulting in that game state. This table also functions as a kind of transposition table, since multiple solutions resulting in the same game state will share a fitness value. These solutions are still allowed in the population as their different genes can help the evolution to progress in different directions, but when a good counter-move is found for one of them, it will also affect the others. We will refer to this table as a *history table* instead of a transposition table, as it serves multiple purposes.

4.8.1 Crossover & Mutation

Parents are paired randomly from the surviving part of the population. The implemented crossover method implements a uniform crossover, that for each gene picks from parent a with a probability of 0.5 and otherwise from parent b. This however has some implications in Hero Academy as picking one action from a might make another action from b illegal. Therefore a uniform crossover with the *if-allowed* rule is introduced. This rule will only add an action from a parent if it is legal. If an action is illegal, it will try to pick from the other parent, but if that also is illegal, a random legal action is selected. To avoid too many random actions in the crossover mechanism, the next action for each parent are also checked before a random action is used. An example of the crossover mechanism is shown on Figure 4.4.

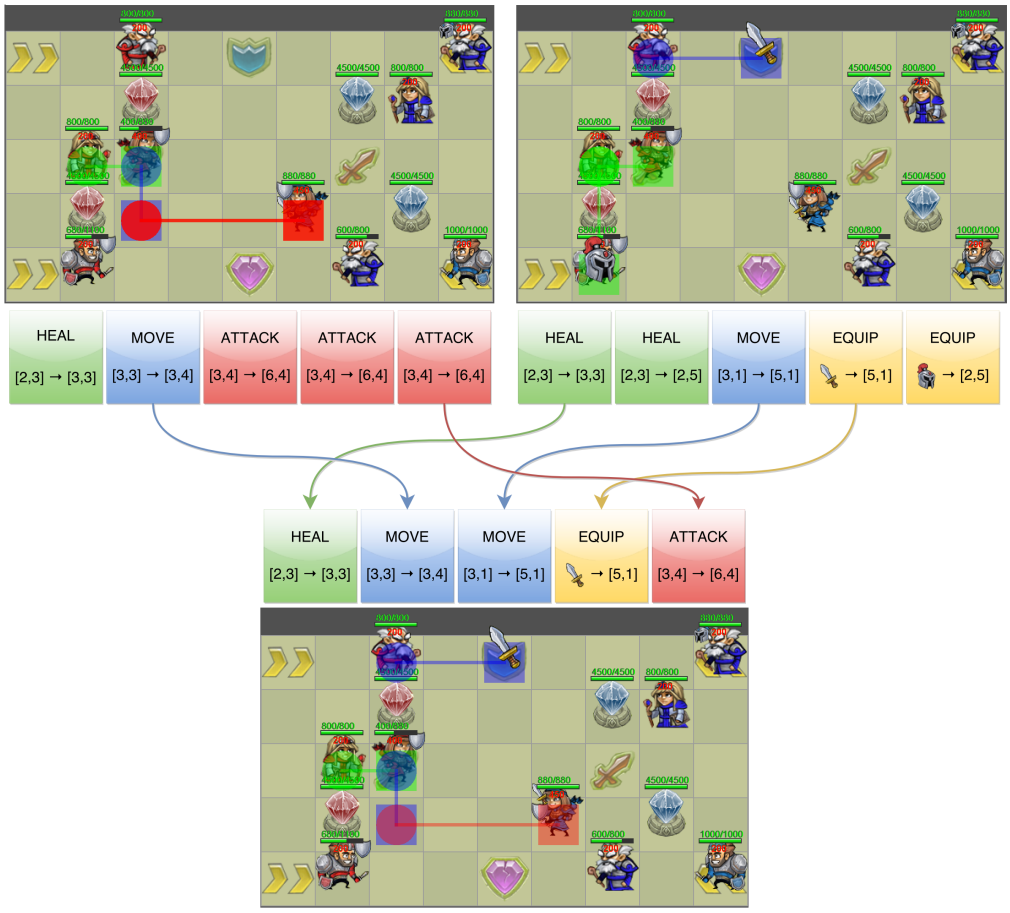


Figure 4.4: An example of the uniform crossover used by the online evolution in Hero Alcademy. Two parent solutions are shown in the top and the resulting solution after crossover in the bottom. Each gene (action) are randomly picked from one of the parents. Colors on genes represent the type of action they represent. Healing actions are green, move actions are blue, attack actions are red and equip actions are yellow.

Mutation selects one random action and swaps it with another legal action. This change can however make some of the following actions illegal. When this occur the following illegal actions are simply also swapped with random legal actions.

If illegal action sequences were allowed, the population would probably be crowded with these and only very few actual solutions might be found. The downside of only allowing legal solutions is, however, that much of the computation time is spend on the crossover and mutation mechanisms, as they need to continuously use the forward model for gene generation.

4.8.2 Parallelization

Two parallelization methods for the online evolution were implemented. The first is simple, as the LeafParallelizer, also used by the MCTS agent, can be used by the online evolution to run several concurrent rollouts. The online evolution was also implemented with the island parallelization model, where several evolutions run concurrently each representing an island. In every generation the second best individual is sent to its neighboring island. Each island uses a queue in which incoming individuals are waiting to enter. Only one individual will enter in each generation unless the queue is empty. In this way genes are spread across islands while the most fit individual remains. A simple pause mechanism is added if one thread is running faster than others and ends up sending more individuals than it is receiving. In this implementation islands were allowed to be behind with five individuals, but such a number should be found experimentally.

4.9 NEAT

This section will describe our preliminary work of evolving neural networks as game state evaluators for Hero Academy. Some discussions of other approaches are also presented in the Conclusions chapter later. Designing game state evaluators by hand for Hero Academy turned out to be more challenging than expected. The material evaluation seems fairly easy, while positional evaluation remains extremely challenging. Positioning units outside range of enemy units should probably be good to protect them, while being inside range of enemy units can, in some situations, put pressure on the opponent. Identifying when this is a good idea, and when it is not, is extremely difficult to describe manually, even though human players are good at identifying such patterns. This suggest, that we should try to either evolve or train an evaluation function to do this task. In the work presented here, the NeuroEvolution of Augmenting Topologies (NEAT) algorithm was used to evolve neural networks as game state evaluators. The first experiments were made for a smaller game board of only 5x3 squares (see Figure 4.5). This makes the size of the input layer smaller, as well as the time spend on each generation in NEAT. The software package JNEAT[1] by Stanley was used in the experiments described.

[1] http://nn.cs.utexas.edu/?jneat

Figure 4.5: The small game board used in the NEAT experiments.

Experiments with the small game board went pretty well, as we will see in the next chapter, and thus experiments with the standard game board were also made.

4.9.1 Input & Output Layer

Two different input layers were designed with different complexities that were tested separately. The simplest input layer had the following five inputs:

1. Total health points of crystals owned by player 1.
2. Total health points of crystals owned by player 2.
3. Total health points of units owned by player 1.
4. Total health points of units owned by player 2.
5. Bias equal to 1.

These inputs were all normalized to be between 0 and 1. This input layer was designed as a baseline experiment to test whether any sign of learning can be observed. A difficult challenge when it comes to designing a state evaluator for Hero Academy is how to balance the pursuit towards the two different winning conditions. Interesting solutions might be learned by this simple input layer design. The other input layer design is fed with practically everything from the game state. The first five inputs are identical to the simple input layer, where after the following 13 inputs are added sequentially for each square on the board:

1. **If** square contains archer **then** 1 **else** 0
2. **If** square contains cleric **then** 1 **else** 0
3. **If** square contains crystal **then** 1 **else** 0
4. **If** square contains dragonscale **then** 1 **else** 0
5. **If** square contains knight **then** 1 **else** 0
6. **If** square contains ninja **then** 1 **else** 0
7. **If** square contains runemetal **then** 1 **else** 0
8. **If** square contains shining helmet **then** 1 **else** 0
9. **If** square contains scroll **then** 1 **else** 0
10. **If** square contains wizard **then** 1 **else** 0
11. **If** square contains a unit **then** its health points normalized **else** 0
12. **If** square contains a unit controlled by player 1 **then** 1 **else** 0
13. **If** square contains a unit controlled by player 2 **then** 1 **else** 0

Ten inputs are added after these, describing which cards the current player has on the hand. Again, 1 is added if the player has an archer and 0 if not, and so on for each card type.

The final input layer ends up with 210 neurons for the 5x3 game board and for the standard sized game board the size is 600 neurons. Some efforts were made on reducing this number without loosing information, but no satisfying solution was found. The output layer simply consist of just one neuron, and the activation value of this neuron is the networks final evaluation of the game state.

Chapter 5

Experimental Results

This chapter will describe the main experiments performed in this thesis and present the results. Experiments were run on a Lenovo Ultrabook with an Intel Core i7-3517U CPU with 4 x 1.90GHz cores and 8 GB of ram, unless other is specified. The Hero AIcademy engine was used to run the experiments.

Experiments that compare agents consist of 100 games, where each agent plays as the starting player 50 games each. Winning percentages were calculated, where draws count as half a win for each player. A game ends in a draw if no winner is found in 100 turns. A confidence interval is presented for all comparative results with a confidence level of 95%. Agents were given a 6 second time budget for their entire turn unless other is specified. The first part of this chapter will describe the performance of each implemented algorithm individually, including experiments leading to their final configurations. In the second part a comprehensive comparison of all the implemented agents will be presented. The best agent were tested in 111 games against human players, and the results from these experiments are presented in the end.

The Hero AIcademy engine was set to run deterministic games only when two agents were compared. The decks of each player were still randomly shuffled in the beginning, but but every card draw were deterministic. This was done because no determinization strategy was implemented for MCTS. The experiments with human players was however run with random card draws, as it otherwise would be an advantage for the AI agents, as they can use the forward model to predict the future.

5.1 Configuration optimization

Several approaches exist to optimize the configuration of an algorithm. A simple approach was used for our algorithms, where several values for each setting where compared one by one. This method does not guarantee to find the best solution, as each setting can be dependent on each other, but it does at the same time reveal interesting information about the effect of each setting. Attempts to optimize the configurations of the MCTS and online evolution agents are described in this section. MCTS was the most interesting to experiment with, as several of the settings have huge impact on its performance, in contrast to the online evolution that seems to reach its optimal playing level regardless of some of its settings.

5.1.1 MCTS

The initial configuration of the MCTS implementation was set to the following:

- Rollout depth-limit: None
- Progressive pruning (cutting/collapsing): None
- Default policy: *RandomMeta* (equivalent to ϵ-greedy, where $\epsilon = 0$)
- Tree policy: $UCB1_{edges}$ with exploration constant $C_p = \frac{1}{\sqrt{2}}$
- Parallelization: None
- Transposition table: No
- Action pruning: Yes
- Action sorting: Yes

We will refer to the algorithm using these settings as *vanilla MCTS*, a term also used by Jacobsen et al. [49], as it aims to use the basic settings of MCTS. Here after, the value of each setting is found experimentally one by one. The goal of the first experiment was to find out if a depth limit on rollouts improves the playing strength of MCTS. 100 games were played against GreedyAction for each setting. Depth limits of 1, 5, 10 and infinite turns were tested. A depth-limit of infinite is equivalent

to not having a depth-limit. The results, which can be seen in Figure 5.1, show that short rollouts increase the playing strength of MCTS and that a depth-limit of one is preferable. Additionally, this experiment shows that even with the best depth-limit setting vanilla MCTS performs worse than GreedyAction.

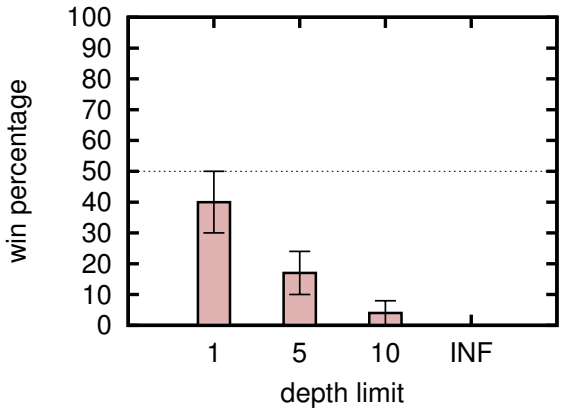


Figure 5.1: The win rate of vanilla MCTS against GreedyAction with four different depth limits. The depth limit is in turns. With no depth-limit (INF) no games were won at all. Error bars show the 95% confidence bounds.

The second experiment tries to find the optimal value for ϵ when the ϵ-greedy policy is used in the rollouts. This was tested for the vanilla MCTS as well as for the two progressive pruning strategies and the non-explorative approach. Table 5.1 shows the results of each method with $\epsilon = 0.5$ and $\epsilon = 1$ playing against itself with $\epsilon = 0$.

MCTS variant	$\epsilon = 0.5$	$\epsilon = 1$
Vanilla	55%	39%
Non-expl.	76%	82%
Cut	57%	55.5%
Collapse	48%	28%

Table 5.1: Win percentage gained by adding domain knowledge to the rollout policy against itself with $\epsilon = 0$. ϵ is the probability that the policy will use the action sorting heuristic instead of a random action.

Interestingly, the non-explorative approach prefers a $\epsilon = 1$, while the other methods prefer some random exploration in the rollouts. It was,

however, expected that the non-explorative approach would not work well with completely random rollouts.

The playing strength of vanilla MCTS was clearly unimpressive even with a depth-limit of one turn. The two progressive pruning methods and the non-explorative MCTS were tested in 100 games each against vanilla MCTS. Each algorithm, including the vanilla MCTS, used a rollout depth-limit of one. The results, which can be seen on Figure 5.2, shows that both the non-explorative MCTS and the cutting MCTS were able to win 95% and 97.5% of the games, respectively. The collapsing strategy showed, however, no improvement with only 49% wins. A $K = 20 \times t$, where t is the the budget, was used for the collapsing MCTS.

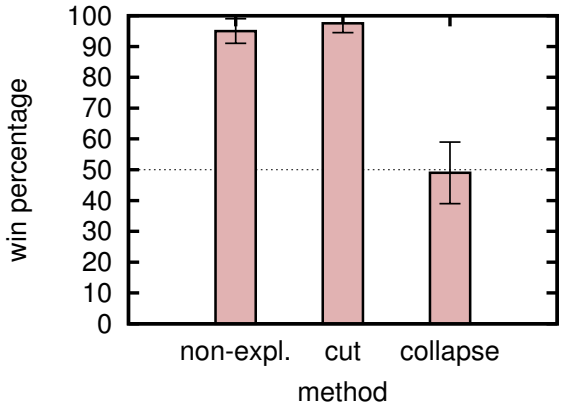


Figure 5.2: The win rates of the non-explorative MCTS, cutting MCTS and collapsing MCTS against the vanilla MCTS, all with a rollout depth-limit of one turn. Error bars show the 95% confidence bounds.

These three methods: cutting, collapsing and non-explorative MCTS, have tremendous impact on the search tree each in their own way. In each turn during all 300 games from the last experiment, statistics were collected about the minimum, average and maximum depth of leaf nodes in the search tree. The averages of these are visualized on Figure 5.3, where the bottom of each bar shows the lowest depth, the top of each bar shows the maximum depth and the line in between shows the average depth. The depths reached by the collapse strategy are very identical to the vanilla MCTS, which indicates that these two method are themselves very similar. It might be because the parameter K was set too high, and then collapses became too rare.

The number of iterations for each method are shown on Figure 5.4, showing that MCTS is able to perform hundred of thousands rollouts

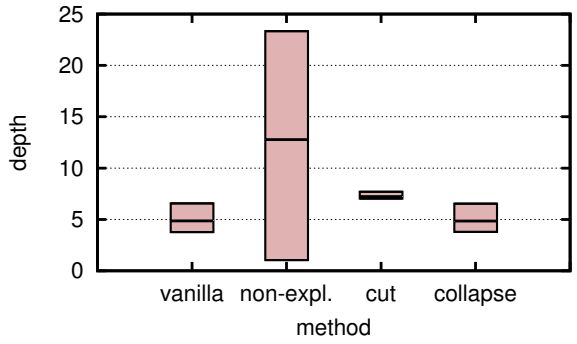


Figure 5.3: A representation of the depths reached by the different MCTS methods. Each bar shows the range between the minimum and maximum depths of leaf nodes in the search tree while the line in between shows the average depth. These values are averages calculated from 100 games for each method with a time budget of 6 seconds. It is thus the average minimum depth, average average depth and average maximum depths presented. The standard deviation for all the values in this representation are between 0.5 and 1.5.

within the time budget of six seconds. The large standard deviation is probably due to the fact that the selection, expansion and simulation phases in MCTS become slower when more actions are available to the player, and as we saw on Figure 2.6, this number is usually very low in the beginning of the game and high during the mid game.

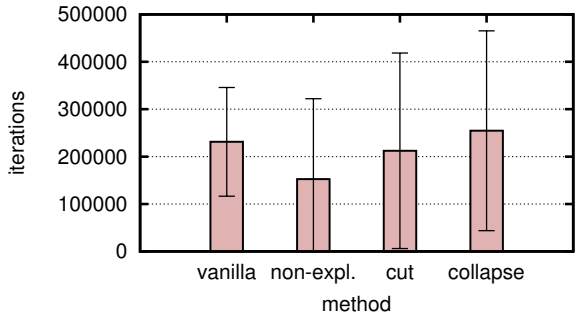


Figure 5.4: The number of iterations for each MCTS method averaged over 100 games with a time budget of 6 seconds. The error bars show the standard deviations.

Both the non-explorative MCTS and the cutting MCTS showed significant improvement. An experiment running these two algorithms against each other showed that the non-explorative MCTS won 67.0% against the cutting MCTS with a confidence interval of 9%.

Since the non-explorative MCTS uses determinstic rollouts, both leaf and root parallelization does not make sense to apply. With leaf parallelization each concurrent rollout would reach the same outcome and with root parallelization each tree would be identical, except that some threads that were given more time by the scheduler would be slightly larger. Tree parallelization were not implemented but would make sense to apply to the non-explorative MCTS, as it would add a new form of exploration while maintaining an exploration constant of zero. The cutting MCTS showed no significant improvement when leaf parallelization were applied while root parallelization actually decreased the playing strength, as it only won 2% against the cutting MCTS without parallelization.

Experiments were performed to explore the effects of the adding a transposition table with the $UCB1_{edges}$ formula to the non-explorative MCTS. A win percentage of 62.5% was achieved with a transposition table against the same algorithm without a transposition table. The average and maximum depths of leaf nodes were increased by around one ply. No significant change were however observed when applying the transposition table to the cutting MCTS.

The best settings found for MCTS in the experiements described in this section are the following:

- Rollout depth-limit: 1 turn
- Progressive pruning (cutting/collapsing): None
- Default policy: Greedy (equivalent to ϵ-greedy, where $\epsilon = 1$)
- Tree policy: $UCB1_{edges}$ with an exploration constant $C_p = 0$
- Parallelization: None
- Transposition table: Yes
- Action pruning: Yes
- Action sorting: Yes

5.1.2 Online Evolution

Several experiments were performed to optimize the configurations for the online evolution agent. These experiments were head to head matchups between two online evolution agents with a different setting, each given a three second time budget. Most of these experiments showed no increase in the playing strength by changing the settings, while some only showed small improvements. The reason might be because the agent is able to find a good solution within the three second time budget regardless of changes of its configuration.

No significant improvement over the other was found when applying 1, 2, 5 or 10 sequential rollouts to the fitness function. However, when applying 50 rollouts the win percentage decreased to 27% against a similar online evolution agent with just 1 rollouts.

The online evolution used the ϵ-greedy policy for its rollouts. No significant improvement was seen when using $\epsilon = 0$, $\epsilon = 0.5$ or $\epsilon = 1$. This was a bit surprising. Additionally, applying rollouts instead of the HeuristicEvaluation as heuristic provided only an improvement of 54%, which is insignificant with a confidence interval of 10%. The history table gave a small, but also insignificant, improvement with a win percentage of 55.5%.

Applying the island parallelization showed a significant improvement of 63%, when a time budget of 2 seconds were used, and a 61% improvement when a time budget of 1 second were used. Figure 5.5 shows the progress of four parallel island evolutions, where immigration of the next best individual happens in every generation.

With a time budget of six seconds the evolutions are able to complete an average of 1217 (with a standard deviation of 419) generations. The most fit individual found on each island had an average age of 60 generations, but with a standard deviation 339. The high deviation might be due the end game, where the best move is found early, and a high number of generations are possible, due to game states with very few units.

The best configuration found for the online evolution agent turned out to be the following:

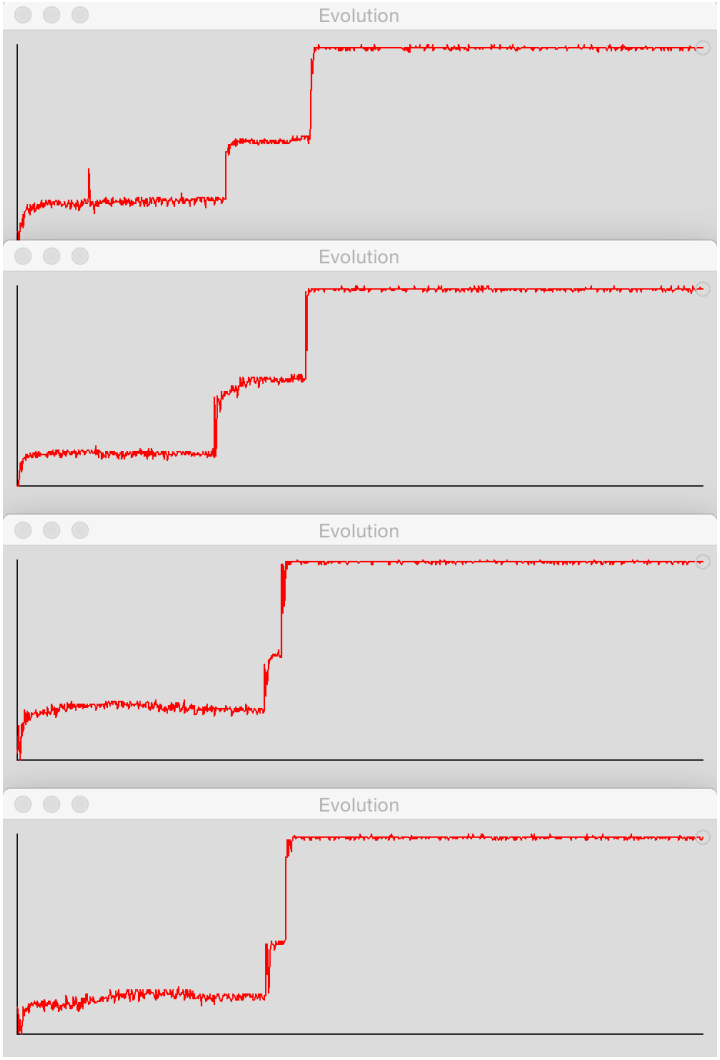


Figure 5.5: The progress of four parallel online evolutions where the x-axis is time in generations and the y-axis is the fitness value of the best solution found. The island parallelization method enables gene sharing between threads and can be seen on the figure as progressions happen almost simultaneously.

- Population size: 100
- Mutation rate: 0.1
- Survival threshold: 0.5
- Fitness function: Rollouts (use minimum obtained outcome)
  - Rollouts: 1

  - Depth limit: 1 turn
  - Policy: ϵ-greedy where $\epsilon = 0.5$.
  - History table: Yes

- Parallelization: Island parallelization with immigration of second best individual each generation.

5.1.3 NEAT

The two input layer designs described in Section 4.9 will be referred to as the *simple network*, with 5 input neurons, and the *complex network* with 210 input neurons for the small game board and 600 input neurons for the standard game board. The networks were only evolved to play as the starting player. The fitness function used in JNEAT simply played 20 games using a neural network against the ϵ-greedy agent and returned the win rate. Initially, ϵ was set to 0 to produce a completely random and thus very easy opponent to train against. As soon as a win rate of 100% was reached, ϵ would be incremented by 0.05 to give further challenge to the trained networks. This approach was inspired by how Togelius et al. evolved neural networks to play Super Mario on gradually more difficult levels [50]. An initial population size of 64 individuals were used, and a complete list of all the JNEAT parameters can be seen in Appendix A. Figure 5.6 shows how ϵ grew as the networks evolved. It is important to stress that e does not express the actual win rate of the best network against ϵ-greedy but simply tells that one individual was able to win 20 games in a row against it.

The experiment for the 5x3 game board ran on the Lenovo laptop, while the experiment on the 9x5 game board ran on a server with eight 2.9 GHz processor cores. Each experiment took three to four days before it was stopped.

The simple network was able to reach an ϵ-level of 0.85 in just 38 generations on the 5x3 game board. It is easy to imagine that this network simply tries to increase the health points of crystals and units under control while decreasing the health points of opponents. The results indicate that these features alone are not enough to reach a good playing level, as the evolution ran for 3817 generations without reaching an ϵ-level of 1.

The complex network reached the ϵ-level of 1 in 586 generations on the 5x3 game board, and the ϵ-greedy agent with $\epsilon = 1$ was finally beat

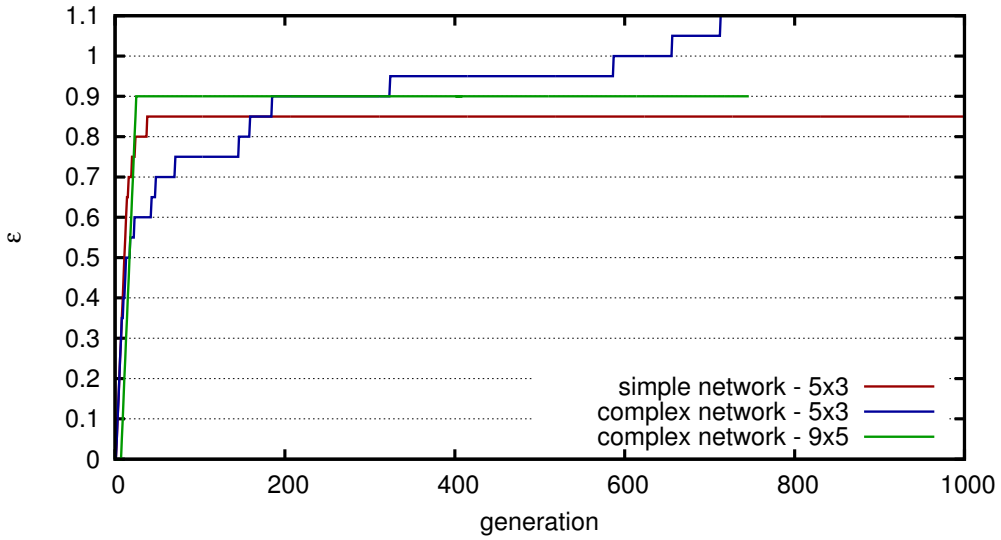


Figure 5.6: The ϵ-value used by the fitness function in each generation of the NEAT experiments. The fitness function plays the individuals 20 times against the ϵ-greedy agent and returns the win rate. When a win rate of 100% is reached, ϵ gets incremented by 0.05.

20 times in a row in the 655th generation. The best network of that generation contained 299 neurons in total with 210 in the input layer, 88 in the hidden layer and 1 in the output layer. The complex network on the 9x5 game board did, however, only reach $\epsilon = 0.9$ in the 746 generations it ran.

Agent	Network	Game board	Win percentage
NEAT	simple	5x3	32.92%
NEAT	complex	5x3	77.66%
NEAT	complex	9x5	48.0%
GreedyAction	-	5x3	51.64%

Table 5.2: Win percentages of the evolved NEAT agents and the GreeyAction agent as starting players in 10,000 games (only 100 on the 9x5 game board) against the GreedyAction agent (not starting).

To test the actual win percentage of the evolved NEAT agents, they were played 10,000 times each on the 5x3 game board and 100 times on the 9x5 game board against the GreedyAction agent. Since the NEAT agent always starts, it could have an advantage and thus the GreedyAction agent is also included in the comparison, where it also always starts. The results are shown on Figure 5.2 and clearly conclude that the

NEAT agent using the complex approach have reached a level, where it is superior to the GreedyAction agent on the 5x3 game board, while the simple network approach is much weaker. The same results was not achieved for the 9x5 game board, where the evolved network won 48% of the games. The results with the GreedyAction agents indicate, that the advantage of starting is insignificant, at least for the strategy applied by this agent.

5.2 Comparisons

Each of the implemented agents with their optimized configurations were tested against each other and against the baseline agents. The results are shown in Table 5.3.

	Random	G-Action	G-Turn	NE-MCTS	OE
GreedyAction	100%		46.5%	5%	7.5%
GreedyTurn	100%	53.5%		25%	9%
NE-MCTS	100%	95%	75%		42.5%
OE	100%	92.5%	91%	57.5%	

Table 5.3: Win percentages of the agents listed in the left-most column. G are short for greedy, NE for non-explorative and OE for online evolution.

The results show, that both the online evolution and NE-MCTS agents are significantly superior to the greedy baseline agents. We expected the GreedyTurn agent to be much better than the GreedyAction agent, but it only achieved a win percentage of 53.5%. When observing a game between the two it became apparent why, as GreedyTurn mostly charges with one unit towards the opponent to make a very effective attack. This often leaves the attacking unit defenseless in the opponent's territory. The GreedyAction agent cannot make forward planning and simply attacks if possible and otherwise tries to optimize its immediate position by deploying units, healing and equipping units with items. The difference in strategies between the GreedyAction agent and the GreedyTurn agent might also be the reason why NE-MCTS is the best agent against GreeyAction while the online evolution is the best agent against GreedyTurn. The results also show that a completely random agent is useless in Hero Academy with zero wins in all the experiments. This seems to be because it waste actions on useless move actions and

very rarely attacks. The number of available move actions are usually much higher than the number of available attack actions, and thus statistically a move action is selected by the random agents.

5.2.1 Time budget

An interesting question is how fast the different algorithms reach their optimal play, and if their playing strengths continue to increase as they are given more time. To approach an answer to this question for the NE-MCTS and the online evolution, they were matched up with various time budgets against the GreedyTurn agent, which had a constant time budget of three seconds. The final configurations from Section 5.1 were used and the results can bee seen on Figure 5.7

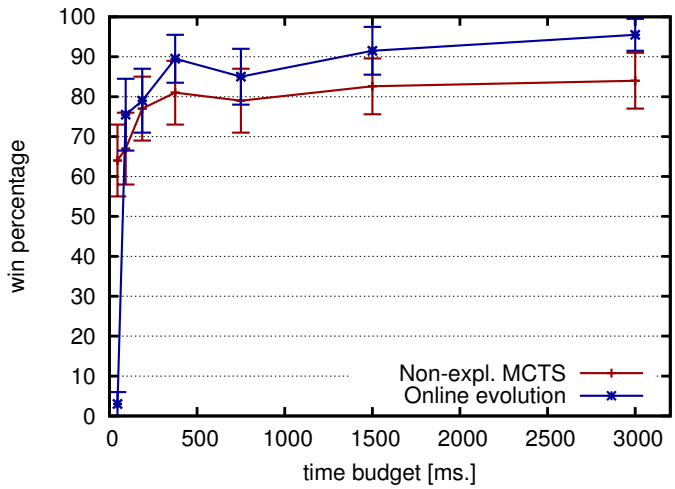


Figure 5.7: The win percentages of the non-explorative MCTS and the online evolution with various time budgets against GreedyTurn with a constant time budget of three seconds.

It is interesting to see that both algorithms are able to reach a win percentage of around 75% with a time budget of just 94 ms., suggesting that these methods could work well in real-time domains as well. The online evolution did, however, only achieve a win percentage of 3.0% with a time budget of 47 ms. possibly because it needs time to set up populations on each thread. Given more time, not a lot of progression is observed, while the online evolution achieved the best results, again.

5.3 Versus Human Players

One important and often overseen experiment is to test the implemented agents against human players. We were able to test our best agent, the online evolution agent, against 111 testers. This gives us knowledge about the current state of the implemented algorithms, and whether they are useful to implement in a real game product. One reason why such experiments are often left out, may be because of the time consuming task of finding test subjects and manually recording the results. To overcome this challenge a web server running Node.js on the Heroku cloud service was setup and connected to a MongoDB database[1]. The Hero AIcademy engine was extended to handle such tests by sending game results of each turn to the web server. Additionally, test subjects were asked about their skill level by the program before they entered the game, with options of being a beginner, intermediate or expert. If players had never played the game before they were asked to choose "beginner". Players that did choose "beginner" were supplied with an additional screen briefly explaining the two winning conditions of the game and how to undo actions during their turn. The program was exported as a runnable JAR file and distributed on Facebook among friends, colleagues and fellow students, on the Steam community page for Hero Academy[2] and on the Hero Academy subreddit page[3].

After about a month the database contained 111 records with 61 beginners, 27 intermediates and 23 experts. All the results can be seen in Appendix B. Among theses records, 55 holds data of games played until a terminal state was reached, with 26 beginners, 13 intermediates and 16 experts. In the remaining games, players simply left before it was over. Figure 5.8 shows the heuristic value, found by the HeuristicEvaluation, when they left the game, where 1 equals a win and 0 equals a loss, and the turn in which they left.

It is reasonable to think that some players left early because they simply did not want to play the game when they saw it, while others might leave because they thought it was too easy or too difficult. Results outside of the score range between -0.1 and 0.1 were treated as either a

[1]https://github.com/njustesen/heroai-testserver
[2]http://steamcommunity.com/app/209270/discussions/
[3]http://www.reddit.com/r/HeroAcademy/

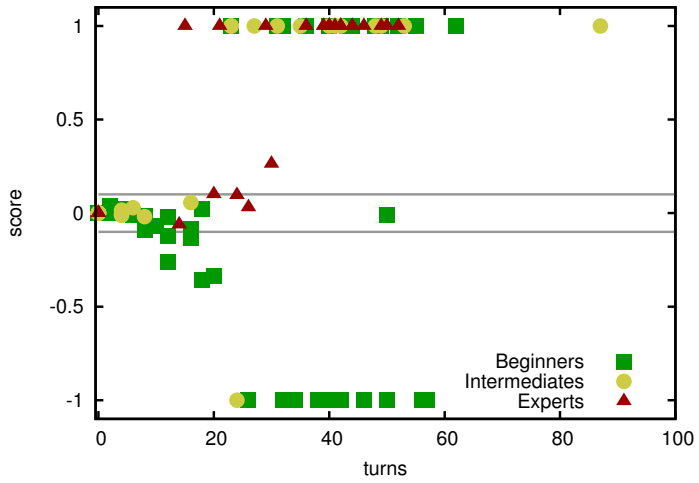


Figure 5.8: The turn in which players left the game and their score at that time. The score was calculated by the HeuristicEvaluation of the game state, where 1 equals a win and 0 equals a loss. Results outside the two grey lines are treated as a win or a loss.

win or a loss. The results for beginners, intermediates and expert players are shown on Figure 5.9, where the blue bars show results for all players outside the heuristic range between -0.1 and 0.1 while the red bars show the results of completed games only. The results show that our agent is able to challenge beginners, while it is easily beaten by intermediate and expert players.

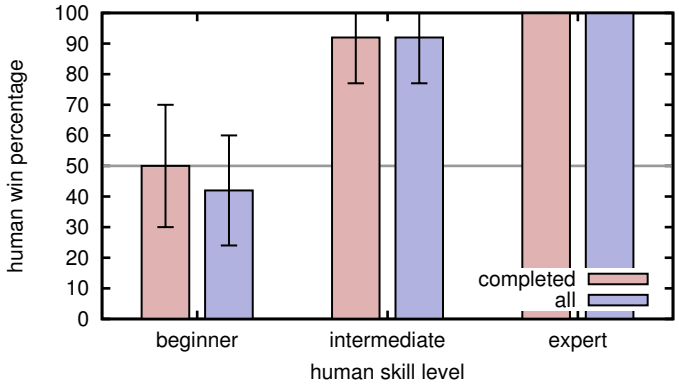


Figure 5.9: The results of human players against the online evolution agent. Error bars show the standard deviation. The red bars only show results from completed games while the blue bars also include games where players quit while being in front or behind by 10% (calculated by the HeuristicEvaluation).

The online evolution agent were using the island parallelization method to reach the best performance possible. One concern when looking at the final results was, that the number of processor cores on the testers PCs, might be the main reason why players lost and this obscured the results. In fact it did turn out that players with 4 cores won 82% of the games, while players with 8 cores only won 67%. This indicates that the winning rate is correlated with the number of processor cores used. Beginners in the experiments used an average of 6.69 cores, while intermediate players used 4.00 and expert players used 4.75. Among the winning beginners an average of 7.38 cores were used, and among the losing beginners an average of 6.00 were used, which contradicts the correlation. The correlation between the human skill level and win percentage is, however, much more evident. Still, the experiment should have been made such that the online evolution agent would use the same number of processor cores in each experiment.

Some of the testers also gave some written feedback. A few examples are shown hereunder.

> “I was beaten."

> “I also lost."

These two comments indicate that some players lost their first game and did not bother to try again.

> “I lost the first time but I had no idea what I was doing.. Second time I beat it!"

A few players expressed that it was difficult in the first game but were then able to win the second.

> “The AI can’t use the upgrades properly. It wastes the fire spell."

This comment indicates that some of the constants used by the heuristic might be suboptimal.

> “It’s far from beating an average player, however I think it’s a good piece of work."

The last comment is from the Steam community page for Hero Academy and conforms with the results on Figure 5.9, that the AI agent is on level with beginners while intermediate, or average, players are able to beat it. Still, the comment indicates that intelligent behavior is observed by saying it is a good piece of work.

Chapter 6

Conclusions

This section sums up the results of this thesis. Several algorithms have been presented that are able to play Hero Academy and even challenge human beginners. We have thus been able to answer our research question partly, while several challenges have been identified, that must be solved to reach a higher playing level. This conclusion can also be seen as our advice to others that want to implement an agent for Hero Academy or similar TTB games.

> **Research question**
> How can we design an AI agent that is able to challenge human players in Hero Academy?

A greedy search was implemented, that was barely able to beat our baseline agent, despite its use of action sorting, pruning and parallelization. The vanilla MCTS performed even worse than our baseline, while we observed that rollouts with a depth-limit of just one turn followed by a static evaluation function works best. A non-explorative variant that uses deterministic greedy rollouts and an exploration constant $C_p = 0$ outperformed the vanilla MCTS with a 95% win percentage. Two novel progressive pruning strategies were also introduced based on *cutting* and *collapsing* the MCTS search tree. The cutting strategy outperformed the vanilla MCTS with a 97% win percentage, while the collapsing strategy showed no improvement. The non-explorative MCTS was shown to be the best of the variants in head to head match-ups. Transposition tables, action sorting, action pruning and ϵ-greedy rollouts were used to enhance the performance of the MCTS agents. A parallel online

evolutionary algorithm was implemented that evolves plans each turn and uses a fitness function based on depth-limited rollouts. The minimum value found by the fitness function is used and stored in a *history table*. This agent reached the best overall performance of all the implemented agents. Uniform crossover was applied that, using an *if-allowed* rule only accepts legal action sequences. The idea of using rollouts in combination with online evolution only showed a small improvement, while it is a unique approach that, to our knowledge, has not been done before. It seems that two areas are important when creating an agent in Hero Academy. The first area is concerned with searching for plans, while the second area is concerned with game state evaluation. The online evolution agent was able to challenge human beginners, while intermediate and expert players easily beat it. As we have highlighted several methods that are able to search for actions adequate, we believe the greatest challenge now is to produce strong game state evaluators. Preliminary attempts were also made to evolve a game state evaluator using NEAT. A network was evolved, that was able to reach a win percentage of 77.66% against our baseline agent on a small game board, while a win percentage of 48.0% was achieved on the standard sized game board. The networks had 13 input neurons for each square on the game board. This suggests that other solutions should be investigated with simple input layers.

6.1 Discussion

The online evolution worked very well in Hero Academy, and it is probably also very effective in other multi-action games with more than five sequential actions such as Blood Bowl, where players can move up to 11 units each turn.

The use of rollouts as a heuristic was quite disappointing. The related work presented in this thesis and our own results suggests, that depth-limited rollouts are necessary in games with large branching factors and complex rules. The improvement of using rollouts in the online evolution was not significant, while we still think it is a very interesting approach, as it allows the evolution to take the opponents actions into account.

It was expected that MCTS would have a hard time overcoming the large branching factor of Hero Academy and that radical enhancements were needed. The cutting and non-explorative approaches rigorously

and naively ignore most of the game tree, which of course has its downsides, but enables the search to explore the opponent counter-moves very well. The collapse approach did, in contrast to our expectations, not improve the performance. Our best guess is that the remaining nodes after the collapse are too uncertain, since MCTS is not able to explore the first five plies very well. Also, the K parameter might be set too high, so that collapses almost never happen. An investigation of this should be performed, but was not possible based on the output of our experiments.

Several of our algorithms used some form of transposition table, which performance is depended on the game state hash function. Proper tests should be performed to test the collision probability of this function. We do, however, not believe that this was a problem in our implementations for two reasons. First, the MCTS search trees have been saved into XML files, where after they were manually studied and the edge connections looked correct. Second, MCTS would most likely run into an infinite loop if collisions were common, as a cycle could be produced in the tree leading to a stack overflow error. This was not observed.

The implemented agents were compared by playing the game against each other. It is important to note that each agent has its own strategy as a result of their different implementations. E.g. the GreedyTurn agent plays very aggressive and can take out human players very fast, by a crystal kill, if the opponent is unwary. It is, however, very easy to beat it, as it is way too aggressive and thus careless with its units. We believe this agent is able to beat some human beginners as well. It would have been optimal if we tested every agent against human players, but it would also require a lot more testers.

If we were to reproduce the experiments with human players, we would definitely give the agents the same number of processors in every test, as not doing so obscured some of the results.

One could argue that our agent only won against human beginners, because most of them did not know the rules. A few rules and the two winning conditions were presented to the beginners before the game began, so we do not believe, they were unaware of what was going on. Still, the beginner category of course includes players that only know some of the rules. We believe the online evolution agent could easily be used, as it is, in the actual game. It could serve as a training opponent for new players and is fast enough to work on mobile devices. It could

also help the developers with procedural content generation, such as generation of game boards and challenges.

Our NEAT approach seems to have more trouble when applied to the standard sized game board in Hero Academy. It might be a bit naive that it would actually work, since so many squares create an enormous amount of dependencies, that probably requires a network with thousands of neurons. A higher level might have been reached if the evolution was run for more generations, but it may require weeks of computation or a super computer with more than 16 cores. New approaches are probably needed to reach a very solid playing level, and some ideas are presented when ideas for future work are presented next.

6.2 Future Work

This section offers a list of future work directions representing some of the questions that remain unanswered in this thesis. It is made as a list of points that can inspire future projects.

1. Further understanding of the non-explorative MCTS is needed. Will it work better if the exploration constant is very small instead of zero? Can tree parallelization improve its performance further? In what other games could it be a useful approach?

2. Two progressive strategies were introduced in this thesis, where the cutting strategy proved to be effective. Can we design other progressive strategies to overcome large branching factors? Can the cutting and collapsing strategies be improved? How does the cutting strategy work compared to existing progressive strategies such as progressive unprunning/widening?

3. MCTS was only tested in the deterministic version of Hero Academy since no determinization method was implemented. One approach could simply be to assume that players in future game positions have all possible cards on their hand. This can violate the rules of the game, as it could result in more than six cards on a hand, but this should not be a problem in Hero AIcademy. Another approach would simply be to select the most probable cards for each players and use this fixed sample. It would be interested to see how these determinization methods would work in the stochastic version of the game.

4. Online evolution using rollouts as a fitness function with a history table was introduced in this thesis. The enhancement of using rollouts was however not significant. The idea still seems interesting and exploring other methods combining evolution and rollouts should be interesting. Few attempts were also made on using cascading fitness functions, but no useful results were achieved. It would be ideal to have an algorithm similar to minimax, but where moves for each players are evolved. If this vision is even possible it definitely requires a lot of ingenuity.

5. Applying a genetic algorithm to optimize the parameters used in the HeuristicEvaluation seems like an obvious way of improving our results. It would probably solve the issues, also pointed out by one of the testers, that the agents cannot use upgrades and spells properly. To be able to improve the positional evaluation as well the HeuristicEvaluation must be extended to also take such factors into account. This might actually be the most promising direction, to reach a higher playing level fast.

6. Potential fields and influence maps are well known methods in game AI but have not been used in this thesis. Since positional evaluation seems to be a very difficult task in Hero Academy, these methods might help and should be explored further.

7. While the online evolution agent only achieved a playing level compared to human beginners, the same approach might work even better in other TTB games. Testing this approach in a game such as the previously mentioned Blood Bowl would be very interesting.

8. Continuation of our experiments with NEAT to find the limit of our approach is interesting. Networks reaching an ϵ-level of 1 could probably be further evolved by tournament selection. A game board of 9x5 squares may be too complex, and new approaches to evolve evaluators for large maps and complex game states are very much needed. Perhaps each unit on the map can be evaluated individually with simpler networks? Or maybe HyperNEAT can be successfully used somehow.

References

[1] Niels Justesen, Balint Tillman, Julian Togelius, and Sebastian Risi. *Script-and Cluster-based UCT for StarCraft*, pages 1–8. IEEE Press, 2014.

[2] GN Yannakakis and J Togelius. A panorama of artificial and computational intelligence in games. 2014.

[3] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[4] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[5] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[6] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

[7] István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In *Advances in Computer Games*, pages 21–32. Springer, 2010.

[8] Cathleen Heyden. *Implementing a computer player for Carcassonne*. PhD thesis, Maastricht University, 2009.

[9] Sondre Glimsdal. Ais for dominion using monte-carlo tree search. In *Current Approaches in Applied Artificial Intelligence: 28th International Conference on Industrial, Engineering and Other Applications of*

Applied Intelligent Systems, IEA/AIE 2015, Seoul, South Korea, June 10-12, 2015, Proceedings, volume 9101, page 43. Springer, 2015.

[10] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon Lucas, Adrien Couëtoux, Jeyull Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition.

[11] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. 2013.

[12] Maurice HJ Bergsma and Pieter Spronck. Adaptive spatial reasoning for turn-based strategy games. In *AIIDE*, 2008.

[13] J v Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[14] Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the ACM*, 25(8):559–564, 1982.

[15] Richard D Greenblatt, Donald E Eastlake III, and Stephen D Crocker. The greenblatt chess program. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, pages 801–810. ACM, 1967.

[16] Albert L Zobrist. A new hashing method with application for game playing. *ICCA journal*, 13(2):69–73, 1970.

[17] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[18] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[19] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[20] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[21] GMJB Chaslot, Mark Winands, JWHM Uiterwijk, H van den Herik, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. In *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. Citeseer, 2007.

[22] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, J-B Hoock, Arpad Rimmel, F Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):73–89, 2009.

[23] Rémi Coulom. Computing elo ratings of move patterns in the game of go. In *Computer games workshop*, 2007.

[24] JPAM Nijssen. Playing othello using monte carlo. *Strategies*, pages 1–9, 2007.

[25] JAM Nijssen and Mark HM Winands. Monte-carlo tree search for the game of scotland yard. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 158–165. IEEE, 2011.

[26] Jean Méhat and Tristan Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):271–277, 2010.

[27] Abdallah Saffidine, Tristan Cazenave, and Jean Méhat. Ucd: Upper confidence bound for rooted directed acyclic graphs. *Knowledge-Based Systems*, 34:26–33, 2012.

[28] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71. Springer, 2008.

[29] Tristan Cazenave. A phantom-go program. In *Advances in Computer Games*, pages 120–125. Springer, 2006.

[30] J Kloetzer. Monte-carlo techniques: Applications to the game of the amazons. *Japan Advanced Institute of Science and Technology*, pages 87–92, 2010.

[31] Tomas Kozelek. Methods of mcts and the game arimaa. *Charles University, Prague, Faculty of Mathematics and Physics*, 2009.

[32] David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in starcraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[33] David Kriesel. *A Brief Introduction to Neural Networks*. 2007.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

[35] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[36] Marco Tomassini. A survey of genetic algorithms. *Annual Reviews of Computational Physics*, 3(2):87–118, 1995.

[37] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5):443–462, 2002.

[38] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 351–358. ACM, 2013.

[39] Arthur L Samuel. Some studies in machine learning using the game of checkers. iiâĂŤrecent progress. *IBM Journal of research and development*, 11(6):601–617, 1967.

[40] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[41] Kumar Chellapilla and David B. Fogel. Evolving an expert checkers playing program without using human expertise. *Evolutionary Computation, IEEE Transactions on*, 5(4):422–428, 2001.

[42] David E Moriarty and Risto Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(3-4):195–210, 1995.

[43] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[44] Keunhyun Oh and Sung-Bae Cho. A hybrid method of dijkstra algorithm and evolutionary neural network for optimal ms. pacman agent. In *Nature and Biologically Inspired Computing (NaBIC), 2010 Second World Congress on*, pages 239–243. IEEE, 2010.

[45] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Evolving competitive car controllers for racing games with neuroevolution. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1179–1186. ACM, 2009.

[46] Bas Jacobs. Learning othello using cooperative and competitive neuroevolution.

[47] Kenneth O Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007.

[48] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *arXiv preprint arXiv:1410.7326*, 2014.

[49] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. Monte mario: platforming with mcts. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 293–300. ACM, 2014.

[50] Julian Togelius, Sergey Karakovskiy, Jan Koutník, and Jürgen Schmidhuber. Super mario evolution. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 156–161. IEEE, 2009.

Appendix A

JNEAT Parameters

Parameter	Value
trait_param_mut_prob	0.5
trait_mutation_power	1.0
linktrait_mut_sig	1.0
nodetrait_mut_sig	0.5
weigh_mut_power	2.5
recur_prob	0
disjoint_coeff	1.0
excess_coeff	1.0
mutdiff_coeff	0.4
compat_thresh	3.0
age_significance	1.0
survival_thresh	0.20
mutate_only_prob	0.25
mutate_random_trait_prob	0.1
mutate_link_trait_prob	0.1
mutate_node_trait_prob	0.1
mutate_link_weights_prob	0.9
mutate_toggle_enable_prob	0
mutate_gene_reenable_prob	0
mutate_add_node_prob	0.06
mutate_add_link_prob	0.1
interspecies_mate_rate	0.02
mate_multipoint_prob	0.6
mate_multipoint_avg_prob	0.4

mate_singlepoint_prob	0
mate_only_prob	0.2
recur_only_prob	0
pop_size	100
dropoff_age	15
newlink_tries	20
print_every	5
babies_stolen	0
num_runs	1
p_num_trait_params	20
p_age_significance	1.1
p_survival_thresh	0.9
p_compat_threshold	0.2

Table A.1: JNEAT parameters used in the experiments.

Appendix B

Human Test Results

Map	AI	Time	Lvl	Win	Turns	Cores	Heuristic	Date
a	OE	6000	1	1	55	8	60000	2015-04-02T02:40
a	OE	6000	1	1	40	8	60000	2015-04-02T02:58
a	OE	6000	1		0	8	0	2015-04-02T03:08
a	OE	6000	1	1	52	8	60000	2015-04-02T04:30
a	OE	6000	1		0	8	0	2015-04-02T04:43
a	OE	6000	1	2	50	4	-60000	2015-04-02T06:17
a	OE	6000	1	1	31	8	60000	2015-04-02T06:17
a	OE	6000	1		0	8	0	2015-04-02T06:26
a	OE	6000	3	1	52	8	60000	2015-04-02T06:27
a	OE	6000	3	1	15	4	60000	2015-04-02T07:10
a	OE	6000	3	1	42	4	60000	2015-04-02T07:16
a	OE	6000	3		20	4	6058	2015-04-02T09:42
a	OE	6000	3		0	4	0	2015-04-02T09:55
a	OE	6000	2		0	4	0	2015-04-02T12:26
a	OE	6000	2		0	4	0	2015-04-02T12:36
a	OE	6000	1		0	4	0	2015-04-02T12:36
a	OE	6000	3		0	4	0	2015-04-02T12:37
a	OE	6000	2	2	24	4	-60000	2015-04-02T12:37
a	OE	6000	2	1	23	4	60000	2015-04-02T13:19
a	OE	6000	3	1	50	4	60000	2015-04-02T13:21
a	OE	6000	2		4	4	945	2015-04-02T15:00
a	OE	6000	1	1	49	8	60000	2015-04-02T15:09
a	OE	6000	1		8	4	-3005	2015-04-02T16:00
a	OE	6000	2	1	31	4	60000	2015-04-02T16:42

a	OE	6000	3	1	39	4	60000	2015-04-02T18:28
a	OE	6000	1	2	34	8	-60000	2015-04-03T01:22
a	OE	6000	1	1	49	8	60000	2015-04-03T01:32
a	OE	6000	1		0	8	0	2015-04-03T01:44
a	OE	6000	1	1	43	8	60000	2015-04-03T04:27
a	OE	6000	1	2	56	8	-60000	2015-04-03T05:27
a	OE	6000	1		0	8	0	2015-04-03T05:40
a	OE	6000	3	1	49	4	60000	2015-04-03T08:10
a	OE	6000	1	2	34	8	-60000	2015-04-03T08:46
a	OE	6000	1		0	8	0	2015-04-03T09:01
a	OE	6000	3	1	40	4	60000	2015-04-03T09:07
a	OE	6000	3	1	36	4	60000	2015-04-03T09:22
a	OE	6000	3	1	40	4	60000	2015-04-03T14:05
a	OE	6000	3	1	44	4	60000	2015-04-03T14:30
a	OE	6000	3	1	41	4	60000	2015-04-04T06:35
a	OE	6000	3	1	29	4	60000	2015-04-04T06:48
a	OE	6000	3		14	4	-3662	2015-04-04T06:57
a	OE	6000	3		26	4	1853	2015-04-04T07:05
a	OE	6000	3	1	21	4	60000	2015-04-04T15:52
a	OE	6000	3	1	42	4	60000	2015-04-04T16:00
a	OE	6000	3		24	4	5770	2015-04-04T16:15
a	OE	6000	1		4	8	520	2015-04-04T19:50
a	OE	6000	1		0	8	0	2015-04-04T19:50
a	OE	6000	1	1	32	8	60000	2015-04-04T23:06
a	OE	6000	1		0	8	0	2015-04-04T23:13
a	OE	6000	1		2	8	2175	2015-04-07T14:25
a	OE	6000	3		30	8	15820	2015-04-10T16:39
a	OE	6000	2		0	4	0	2015-04-11T13:50
a	OE	6000	2	1	27	4	60000	2015-04-11T13:51
a	OE	6000	2		4	8	-750	2015-04-11T23:24
a	OE	6000	1		12	4	-7343	2015-04-12T09:25
a	OE	6000	1	2	57	4	-60000	2015-04-12T09:44
a	OE	6000	1	1	62	4	60000	2015-04-12T10:33
a	OE	6000	1		0	4	0	2015-04-12T11:28
a	OE	6000	2		8	4	-1137	2015-04-12T12:04
a	OE	6000	2	1	87	4	60000	2015-04-12T12:10
a	OE	6000	2		0	4	0	2015-04-12T12:50
a	OE	6000	1	1	36	8	60000	2015-04-13T00:11
a	OE	6000	1		0	8	0	2015-04-13T00:25

a	OE	6000	3	1	46	8	60000	2015-04-13T00:25
a	OE	6000	1	1	48	8	60000	2015-04-13T18:41
a	OE	6000	1		2	8	0	2015-04-14T10:00
a	OE	6000	2	1	53	4	60000	2015-04-16T20:37
a	OE	6000	1	2	34	4	-60000	2015-04-18T15:58
a	OE	6000	1	2	38	4	-60000	2015-04-18T16:09
a	OE	6000	1		6	4	-640	2015-04-18T20:18
a	OE	6000	2		0	2	0	2015-04-18T22:45
a	OE	6000	2		6	2	1715	2015-04-18T22:46
a	OE	6000	2		0	2	0	2015-04-19T23:16
a	OE	6000	3	1	40	8	60000	2015-04-20T00:27
a	OE	6000	1	2	46	2	-60000	2015-04-01T10:27
a	OE	6000	1		4	8	1375	2015-04-01T11:32
a	OE	6000	1		0	4	0	2015-04-01T12:35
a	OE	6000	1		16	2	-5027	2015-04-01T12:39
a	OE	6000	2	1	42	4	60000	2015-04-01T12:45
a	OE	6000	1		2	8	1970	2015-04-01T12:50
a	OE	6000	1	2	32	8	-60000	2015-04-01T12:52
a	OE	6000	1	2	42	4	-60000	2015-04-01T12:58
a	OE	6000	2		4	4	-410	2015-04-01T13:03
a	OE	6000	1	2	39	8	-60000	2015-04-01T13:08
a	OE	6000	1		2	8	0	2015-04-01T13:10
a	OE	6000	2	1	48	4	60000	2015-04-01T13:28
a	OE	6000	2	1	41	4	60000	2015-04-01T13:48
a	OE	6000	1		16	8	-7900	2015-04-01T14:03
a	OE	6000	1	2	40	8	-60000	2015-04-01T14:45
a	OE	6000	1		50	8	-750	2015-04-01T14:48
a	OE	6000	1	1	44	4	60000	2015-04-01T14:49
a	OE	6000	1		0	4	0	2015-04-01T15:14
a	OE	6000	1		8	8	-1005	2015-04-01T15:19
a	OE	6000	1	2	26	8	-60000	2015-04-01T15:30
a	OE	6000	1		2	8	1925	2015-04-01T15:43
a	OE	6000	1		10	4	-4280	2015-04-01T16:17
a	OE	6000	1		20	4	-20205	2015-04-01T17:34
a	OE	6000	2		0	8	0	2015-04-01T18:42
a	OE	6000	2	1	49	4	60000	2015-04-01T18:45
a	OE	6000	2	1	35	4	60000	2015-04-01T19:06
a	OE	6000	1		12	4	-15795	2015-04-01T19:12
a	OE	6000	1		12	4	-1110	2015-04-01T19:16

a	OE	6000	2	1	40	4	60000	2015-04-01T19:21
a	OE	6000	2	1	23	4	60000	2015-04-01T19:21
a	OE	6000	2		0	4	0	2015-04-01T19:29
a	OE	6000	2		16	6	3390	2015-04-01T19:34
a	OE	6000	1		8	8	-5300	2015-04-01T20:43
a	OE	6000	1		18	4	1185	2015-04-01T22:11
a	OE	6000	1	1	23	8	60000	2015-04-02T00:59
a	OE	6000	1		0	8	0	2015-04-02T01:05
a	OE	6000	1		18	8	-21393	2015-04-02T02:35

Human test results against the online evolution agent.