Technical Report CS17-05

Intelligent Monte-Carlo Tree Search for Perfect Information Games

Lucas Hawk

Submitted to the Faculty of
The Department of Computer Science

Project Director: Dr. Janyl Jumadinova
Second Reader: Dr. John Wenskovitch

Allegheny College
2017

I hereby recognize and pledge to fulfill my responsibilities as defined in the Honor Code, and to maintain the integrity of both myself and the college community as a whole.

Lucas Hawk

**LUCAS HAWK. Intelligent Monte-Carlo Tree Search
for Perfect Information Games.
(Under the direction of Dr. Janyl Jumadinova.)**

ABSTRACT

We introduce a new general-purpose game playing platform for two-player turn-based board games, and an implementation of two perfect-information games, Go and Hex, which can be played on the platform. Furthermore, we implement four intelligent game-playing agents which each use Monte-Carlo Tree Search as the base of their decision-making algorithm. Each of these agents are modified using different machine learning approaches which have been proposed as potential improvements on the Monte-Carlo Tree Search algorithm by previous researchers. Using our game playing framework and implementations of Go and Hex, we provide a direct, robust comparison of the performance of each of our agents. We find that, overall, the agent utilizing an artificial neural network to prune the search tree at an exponentially decaying rate provided a consistent, moderate improvement on standard MCTS. Additionally, we find that an agent utilizing a genetic algorithm as its default policy provides more variable performance, but sometimes provides an even larger boost in performance.

Thank you caffeine. You helped more than you know.

Contents

List of Figures

Chapter 1

Introduction

Our research focuses on the development and comparison of intelligent agents for playing perfect information games. Specifically, we implement agents for playing the games Go and Hex, which we will thoroughly describe later. The agents each include modifications to the Monte-Carlo Tree Search algorithm which have been previously implemented or proposed by researchers. Each modification of Monte-Carlo Tree Search has been empirically proven to improve upon various applications of the algorithm, but have not been directly compared to one another in the same environment. We provide a direct comparison between the agents by comparing their performance in several variations of Go and Hex.

The purpose of this chapter is to outline our motivation behind researching this problem, further state the goals of our research, and introduce the Monte-Carlo Tree Search (MCTS) algorithm, which is very widely used for game-playing agents. We also explain the rules of Go and Hex, as well as some common structures found within the games which we utilize in one of our agents. Lastly, we provide an overview of the structure of this paper.

1.1 Motivation

A major long-term goal of artificial intelligence (AI) research is to create a general intelligent agent — a machine which can act independently, and intelligently, in a variety of situations. In other words, create a machine which is able to mimic a human's intelligence. Specifically over the last decade, researchers have become increasingly close to achieving this goal; for some tasks, AI systems are able to match or even exceed a human's performance. For example, in recent years, research into deep neural networks has led to systems which perform very well at pattern recognition problems [1]. Using a large amount of training data, they are able to label objects in images [1], interpret human speech [2], distinguish faces from one another [3], and even mimic human speech [4]. Essentially, such systems can learn to mimic human senses and classification abilities.

While this is certainly impressive, human intelligence is defined by far more than recognizing patterns and understanding the things we sense. The ability to react to our world, make decisions on the fly, and adapt to changes is a better measure of human intelligence. Essentially, our decision-making abilities are not limited to a finite set of problems — regardless of the task, we are able to work towards finding a solution. Thus in order to progress towards the goal of a more generalized AI, we need to work on building systems which can exist, adapt, and function in changing environments.

One's first instinct in developing a general AI may be to use physical robots as intelligent agents and the real world as an environment. Robots have been (and continue to be) used in AI research and competitions — for example, the RoboCup Soccer [5] tournament is an annual competition which pits cooperative multi-agent robot systems against one another in games of soccer (Figure 1.1). The teams of robots consistently improve every year. Despite this, their limitations are very obvi-

ous. At RoboCup, there are several examples of robots failing to do what they were designed to, sometimes in spectacular fashion.

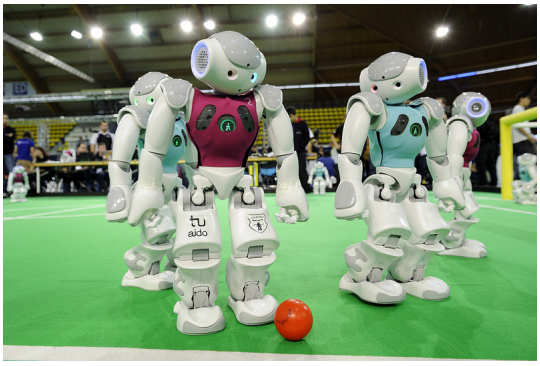


Figure 1.1: RoboCup 2015 [5]

Robots are simply not an ideal way to develop new artificial intelligence techniques — robots are expensive to build, time-consuming to test, and their effectiveness is bottlenecked by their physical limitations [6, 7]; they are not an effective way to test new algorithms. Furthermore, there are many variables of the real world which we are simply unable to control effectively, e.g. gravity. Robots do hold a very important place in the field of AI and computer science in general, but they are not suited for testing new and improving algorithms. On the other hand, purely virtual environments — specifically computer board games — are perfect for experimentation for several reasons.

When working with pure software agents rather than robots, we do not need to worry about the physical efficacy and limitations of the agents — the integration of hardware and software is not needed. Meanwhile, a board game has a set of rules which cannot be broken, and a very obvious goal for everyone in the game: to win. Furthermore, the rules and win-conditions can be tweaked as needed, and the game can be sped up to allow for quick learning and testing. Simply put, such games offer extremely controllable environments. The clear goal (to win) allows for very easy experimentation and testing of game playing agents. To see how an agent compares

against humans, we can simply have the agent play a number of humans to see how often it wins. We can also have two different agents compete head to head against one another in order to compare two different algorithms.

Perfect information games are a subset of games for which each player has access to all of the game information — nothing is hidden from either player. For example, chess and go are both perfect information board games as both players know the location of any and all pieces on the board. However most card games, or a game such as Stratego, are imperfect information as the value of each players pieces are hidden from one another [8].

When comparing two agents in a head to head game, we consider perfect information to be superior to imperfect information games. This is because perfect information ensures that the reason for an agent's win is actually a superior algorithm. In an imperfect information game, there could be pieces of information which are more valuable than other pieces of information — results of the game could be skewed towards whichever AI stumbles upon this information first [9]. For example, in Stratego, the value of a player's piece is hidden from their opponent until an "attack" is made against the piece — if one player happens to find higher level pieces than their opponent early in the game, the game may become skewed in their favor through pure luck. While it may be possible to account for such situations (or perhaps such imbalances may begin to even out over many trials), it can be easier to just begin with a perfect information game.

For any game, we can think of the "state" of the game at any time to be a snapshot of the game at that moment, or rather a recording of all of the game's information at that specific time. For turn-based games, the current game game state changes exactly when a player makes a move during his/her turn. In any intelligent game-playing agent, the most important factor is the ability of the agent to evaluate

the game state — that is, the methodology it uses to determine the probability of winning the game for a given state. For this, one might consider building a system which attempts to identify patterns among positive game states and construct a set of beliefs to approximate the value of any given state; this set of beliefs can be thought of as a heuristic or evaluation function. However, building an adequate evaluation function for a game state is a very complex task; in fact, there has been much research into using heuristic analysis for games with little success [11].

Tree search methods are another way of determining the value of a game state. A tree search method simulates a number of games from the given state, attempting to fully construct a tree of all possible game configurations from the current state. Each game state can be represented as a node in a tree, with a move which results in moving from one state to another being represented by a link between nodes — this tree representation of a game is called a game tree. We loosely refer to the rate at which each level of the game tree increases in size as the branching factor of the game. Rather than trying to directly analyze the value of a given game state, a tree search method determines value of a game state by finding how many paths to a winning position exist.

As a tree search algorithm runs, the game tree becomes fuller; once the tree is full, an agent utilizing it is able to play perfectly — that is it will always win provided it is possible. The downside of many tree search algorithms is that they are not useful until the game tree is at least mostly full. In the minimax algorithm, for example, a node's value cannot be determined until the value of every one of its children down to the leaf nodes is found [10]. For a game such as Go, this is impossible to achieve in any reasonable amount of time — a single game of traditional 19x19 Go has over 2×10^{170} possible playouts. The exact number of possible Go games was only recently calculated, and required an estimated 30 petabytes of disk I/O [12]. Due to the sheer

amount of storage which would be needed to track each possible game state, it would be impossible to even come close to actually constructing the whole game tree.

Over the last decade, game AI development has moved away from basic tree search and towards methods involving the Monte-Carlo Tree Search (MCTS) algorithm [16]. MCTS asymmetrically explores the game tree by taking into account both the current estimated value of each child node, and the number of times each child node has been visited in a simulation. The specific method of exploration can be customized by considering one of these factors more heavily than the other. MCTS is also different from other tree search algorithms as it does not need to construct an entire game tree in order to make a decision. Such methods have proven far more successful, as MCTS-based AIs have since become some of the most successful game AIs in the world [13, 14].

1.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a selective search method for finding optimal decisions via random sampling. Since its development in 2006, it has been widely used for the creation of game AI, specifically for games which can be represented as a finite tree of moves [11, 16]. Prior to its development, the minimax algorithm was commonly used as the basis for finite-game playing agents. An explanation of the minimax algorithm is outside the scope of this paper, but an in-depth explanation of the algorithm can be found in [10].

MCTS is superior to previous tree search methods such as minimax for a couple of reasons. First, while an agent utilizing minimax guarantees optimal play, minimax requires the entire game tree be explored — this often requires an impractical amount of time for games with even just a moderate branching factor. MCTS, however, can be interrupted at any time and return a node to explore. Second, given adequate

time, MCTS provides optimal play given adequate memory and time — in fact, MCTS converges to minimax [16]. So essentially, MCTS can provide accurate decision making without the potentially exponential time complexity of minimax.

MCTS works in four steps as depicted in Figure 1.2 Selection; Expansion; Simulation; and Backpropogation.

1. Selection: Starting at the root of the tree, child nodes are selected recursively until a leaf node L is reached. The way in which these children nodes are selected is called the *tree policy*, and is discussed below.

2. Expansion: if L is not a terminal node for the game tree, a child node C is created. Depending on the application of MCTS, more than one child node may be chosen.

3. Simulation: Simulate a full play of the game from node C. In the simulation step, the tree policy is not used during the simulation. Rather, a *default policy* is used — most commonly, the default policy is a simple random selection from possible moves until a terminal condition is met. Note that the nodes visited during simulation are not added to the tree.

4. Backpropogation: The tree is updated with the results of the simulation. Specifically, each node's estimated value is updated (based on the outcome of the simulation), as well as the number of times it has been visited.

During the selection phase, there is a trade-off between *exploitation* and *exploration* — should we *exploit* the nodes whose rewards we already know, and continue to expand a well searched path, or should we *explore* the less visited nodes in hopes of finding a better option [19]? A widely used tree policy to balance these options is called Upper Confidence Bounds for Trees (UCT). When using UCT as the tree policy, a child node which maximizes the following is chosen:

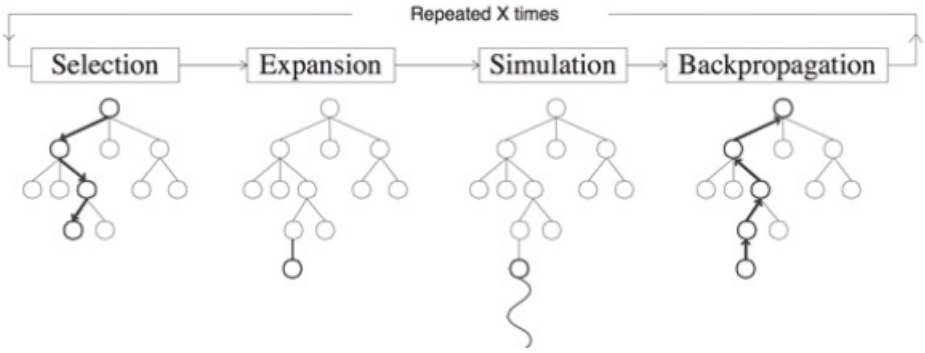


Figure 1.2: Monte-Carlo Tree Search algorithm [11]

$$UCT = v_i + C * \sqrt{\frac{2 \ln N}{n_i}} \tag{1.1}$$

where v_i is the estimated value of the node, n_i is the number of the times the node has been visited and N is the total number of times that its parent has been visited. C is a constant bias parameter which we can change as we wish. In this formula, v_i represents exploitation, while the rest of the equation represents exploration. So, by properly tuning the value of C, we are able to find a balance between exploitation and exploration [17, 18].

While MCTS can be rather effective in its most basic form, it is further enhanced using a variety of machine learning techniques. Most of the industry leading game-playing agents utilize an MCTS decision making algorithm with the integration of some other AI techniques — most often, some form of Artificial Neural Network (ANN) or Genetic Algorithm (GA) is used [16]. These algorithms and existing implementations of game-playing agents which utilize these algorithms will be discussed in Chapter 2.

1.3 Games

In this section, we outline the perfect information games we use for our experiments: Go and Hex. Due to their high branching factor, Go and Hex are two rather well-known and well-studied games in the field of AI. The purpose of this section is to give the reader an understanding of the basic rules, and overview a few common structures within the two games in order to provide a deeper understanding of strategies which players, human or software, can employ in the game.

1.3.1 Go

Go is a two-player game usually played on a 19x19 board, although any size board can be used and boards of size 9x9, 13x13, and 17x17 are particularly common. The rules of Go are very simple. The players take turns placing stones on the board, attempting to surround more territory than the other player. Stones cannot be removed, but if one or more of a player's stones are completely surrounded by the other's, the surrounded stones are removed from the board (Figure 1.3). The game continues until either neither player wishes to move one player resigns, or there are no legal moves remaining on the board.

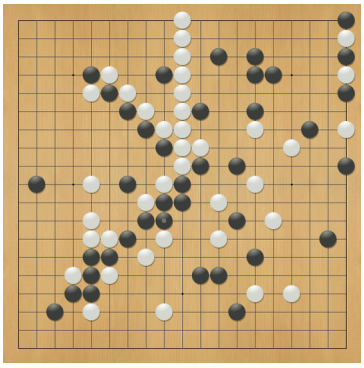


Figure 1.3: An example of a Go board generated in GoGui [20]

Despite its simple rules, Go has over 2×10^{170} possible playouts, making it one of

the most computationally complex board games ever created [12]. The combination of this fact with its popularity as a game has led to it being the focus of a large amount of research compared to other games.

Go has many well-defined board structures, but some of the most common and important concepts are that of *liberties*, *atari*, and *seki*. Each stone has between zero and four liberties, a liberty being an open space directly adjacent to the stone. For example, a stone which is placed such that it is not touching any other stones is said to have 4 liberties. A region of stones is said to be in atari if the entire region has only a one liberty — that is, if the entire region could be captured in a single move by the other player. Seki refers to empty spaces which cannot be safely played on by either player — a space is in seki if placing a stone on the space would immediately allow the other player to capture a region. In Figure 1.4, the two spaces marked in green are in seki because if a player were to place a stone in one of them, the other player would be able to immediately surround and capture a region by placing a stone in the other.

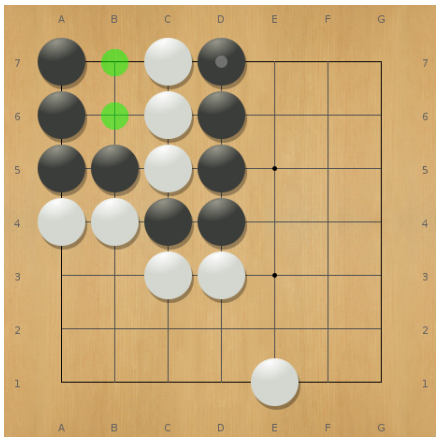


Figure 1.4: Example of seki in Go

There are various methods of scoring a game of Go. The simplest of these is the *stone scoring* method, which was popular in the past but is usually no longer used

[15]. Under this scoring method, each player's score is determined by how many of his pieces are on the board; the player with the most pieces on the board at the end of the game is the winner. This method of scoring has given way to the *territory scoring* method [15]. Under territory scoring, a player's score is determined as:

- The number of empty spaces on the board which which only the player's stones surround,
- minus the number of empty spaces on the board which are surrounded and in seki,
- minus the number number of the player's stones which have been captured;
- minus the number of "dead" stones, i.e. stones which both players agree would be captured by the end of the game.

The last aspect of territory scoring makes it very hard to determine territory score programmatically. Players can argue that any of their pieces on the board are not dead even if their removal is seemingly inevitable. At the same time, both players can agree to certain pieces being dead even if they appear to be far from being captured; territory scoring has a decidedly human element to it.

1.3.2 Hex

Hex is a two-player board game usually played on a hexagonal grid, usually a 14x14 rhombus as shown in Figure 1.5. Each player takes turns placing a stone on a cell of the grid, and simply attempts to link their two opposing sides before the opponent links the other two. The first player to connect their two sides wins. The only other rule is that, due to the first player to move having a distinct advantage, the second player can choose to switch positions with the first player after the first move.

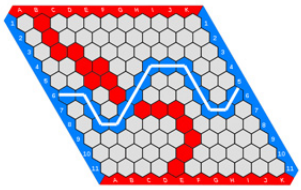


Figure 1.5: A winning game for blue [37]

A game of Hex has a branching factor similar to that of a game of Go on the same size board, but Hex games are usually far shorter, and also much less computationally expensive to simulate [15]. This is because of a simpler, better-defined end game condition (a connection between opposite sides), as well as the fact that pieces never have to be removed from the board. Games can end with the majority of the board being empty, and we never need to check if a piece should be removed from the board as we do in Go. One interesting aspect of Hex which differs from Go is that Hex cannot end in a draw; if every space on the board is filled, there must exist a path between two opposite edges [15].

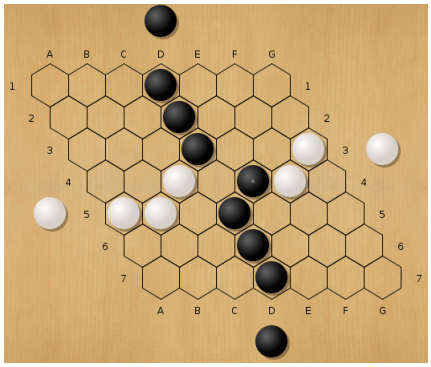


Figure 1.6: A bridge structure in Hex

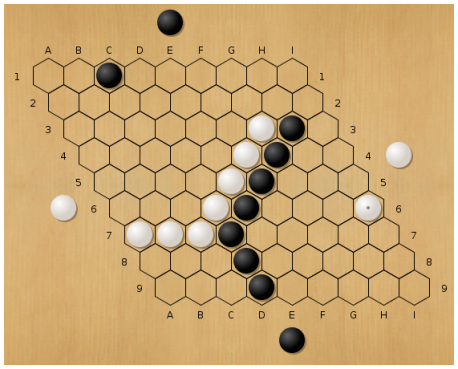


Figure 1.7: A ladder structure in Hex

Hex has far fewer well-defined board structures than Go. In fact, it only has two: *bridges* and *ladders*. A bridge refers to a structure wherein two seperate regions of a players pieces will be able to be connected in one move, regardless of where the opponent plays. A ladder refers to a structure where a string of one players stones is

placed immediately. Figure 1.7 provides an example of a bridge for black, and Figure 1.6 provides an illustration of a ladder being formed.

1.4 Goals of the Project

As stated previously, most of the current top-performing game-playing agents utilize MCTS with some integration of a machine learning algorithm. These agents are almost always shown to perform better than agents using just MCTS or other decision making algorithms. However, they have rarely if ever been directly compared to one another. This thesis helps fill this gap in research. We implement a number of intelligent game-playing agents using MCTS integrated with machine learning algorithms which have been utilized in current, leading systems. These agents are benchmarked against one another on the perfect information games Go and Hex. This provides a concrete, direct comparison of existing techniques, and a clear heirarchy between these techniques is shown.

The use of multiple games is important in these tests. While the performance of standard MCTS is game-independent, the performance of heuristic techniques is often quite dependent on the game, as stated earlier. When we integrate MCTS with these different AI algorithms, we are essentially introducing a small heuristic bias in the way MCTS performs. This addition of a heuristic bias may lead to an agent's performance to be more game-dependent — that is, the bias may affect the agent in different ways for different games. Using multiple games helps us measure how this introduction of a heuristic bias affects the game-dependency of an agent's performance.

1.5 Thesis Outline

Chapter 2 introduces the different AI algorithms we integrate into MCTS, and also describes existing implementations utilizing these algorithms. Each section of the chapter focuses on a different algorithm, and outlines a game-playing agent which has been implemented using such an algorithm. The chapter also discusses the Encog machine learning library which we use extensively in our implementation. Chapter 3 goes over the structure of our game-playing framework, the implementation of the different games and agents, and our experiment design. Chapter 4 contains the results of our experiments, and we briefly explain any trends in the data. Finally, we provide an analysis of our results and discuss any future research or improvements which could be made to the code base in Chapter 5.

Chapter 2

Related Work

In this chapter, we discuss the different algorithms we integrate into MCTS, as well as describe existing implementations of game-playing agents which use these techniques. In addition to explaining the algorithms we are using, we also describe the algorithm behind Google's AlphaGo, the current world-champion computer Go agent. The chapter is organized by algorithm, with each section discussing a specific algorithm and existing implementation of the algorithm. We discuss how each existing system has been implemented, and briefly overview how well these implementations have performed. After this, we describe the Encog machine learning library which is used in our implementation.

2.1 Genetic Algorithms

A Genetic Algorithm (GA) is an optimization or search algorithm based on Darwin's theory of evolution [21]. The algorithm borrows ideas from biology such as genes and chromosomes, as well as the concepts of crossover and mutation. In biology, a gene can be thought of an encoding of a trait, such as eye color, while a chromosome is a collection of genes which make up the "blueprint" of an organism. Crossover is the process of selecting genes from two parent chromosomes to create a child chromosome,

while mutation is a random alteration of a gene. Before explaining how a GA works, it is important to first understand what these terms, and a few others, mean in regards to the algorithm.

In GAs, a chromosome refers to a possible solution to a problem, which is often encoded as a string of bits. The genes are the various chunks of the chromosome which encode a specific element of the solution. For example, consider an agent from a GA which trades on the stock market. Every stock indicator (such as change in price, volume, etc.) may be represented by a gene consisting of 4 bits — the value of each gene determines how highly it considers that indicator compared to the others. The agent's chromosome is made up of a collection of these genes and is its overall trading strategy. Crossover might consist of randomly selecting genes from two agents to create a new chromosome, while mutation might be having a possibility of flipping a random bit in a gene.

Furthermore, the entire collection of chromosomes (i.e. potential solutions) is referred to as the *population*. The *fitness* of a particular chromosome is essentially how good of a solution it provides; the *fitness formula* $f(x)$ is the method of calculating fitness. On each iteration of the algorithm, a new population is created — each population is referred to as a *generation*.

A GA begins by generating a random population for the problem space. Then, the fitness of each member of the population is calculated by some function $f(x)$; a chromosome's fitness may also just be an amount relative to the measured fitness of other chromosomes. Next, we repeat the following steps until n new chromosomes have been created:

- Select a pair of chromosomes (parents) from the population. The probability of selection is higher for chromosomes with a higher fitness;
- Via some predetermined method, perform a crossover between the two parents

to create a child chromosome;

- Randomly mutate the child chromosome with a predetermined probability p_m.

Once n children have been created, we replace the n lowest performing children in the population with the new children — this creates our new generation. This process of evaluation, selection, crossover, and mutation repeats itself until either (a) a desired fitness is reached or (b) a certain number of generations has been produced. Figure 2.1 depicts the GA algorithm visually; each box in the figure represents a different member of the population, while the different colors represents different "genetic makeups" of each chromosome.

Note that the crossover portion of the algorithm helps improve the overall fitness of the population over time, while the mutation helps maintain diversity in the population and protect from the chromosomes becoming overfit. These concepts, as well as the random distribution of chromosomes over the search space at the start, help keep the algorithm from getting stuck at local maxima.

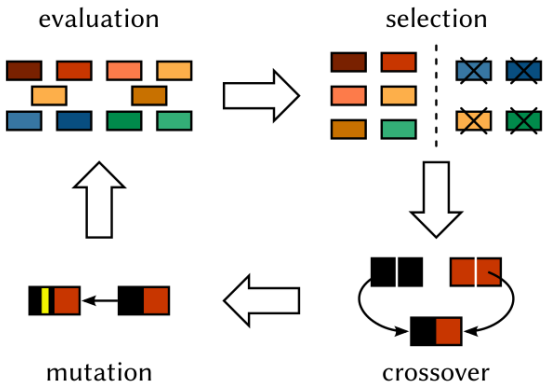


Figure 2.1: Process of a Genetic Algorithm

A GA can be integrated with MCTS to fine-tune both the tree policy and the default policy. In [23], Cazenave created a Go-playing agent which utilized MCTS whose UCT exploration parameter was optimized by a genetic algorithm. He compared three agents: one whose tree policy was a static UCT, one which used a tree

policy known as RAVE [22], and finally one with UCT whose exploration parameter he optimized with a GA. The agent with the optimized bias parameter consistently outperformed the other two.

More interesting is Lucas *et al.* 's fast evolution method [17]. Rather than pre-optimize the MCTS algorithm, Lucas *et al.* developed a system which optimizes its performance on the fly. Rather than evaluate individuals after entire game playouts, each iteration of the MCTS algorithm is followed by the evaluation of a number of individuals all working on the same search tree. At every iteration, both the tree policy and the default policy is biased towards the most fit individuals policy. The results of Lucas *et al.* 's work were positive, showing their algorithm performed better than a standard MCTS agent in 99% of runs in games called *Space Invaders* and *Mountain Car*.

2.2 Artificial Neural Networks

An Artificial Neural Network (ANN) is a machine learning method which, like genetic algorithms, has a basis in biology — it is often viewed as a simplified model of how the brain solves problems [24]. ANNs are commonly used for pattern recognition or data classification. Very abstractly speaking, an ANN is simply a cluster of nodes connected by links, with each link having a numerical weight associated with it.

More specifically, each node takes in a number of inputs and produces a decimal output (usually a number between 0 and 1). Some nodes are connected to the network's environment and are called *input nodes* or *output nodes*. As one might imagine, input nodes are those which receive data or information from the environment, and output nodes give us the a value based on the network's inputs. Any other nodes in the network are called *hidden nodes* because they cannot be directly observed/accessed from the environment — their inputs are either the output of input nodes or

other hidden nodes.

Each link between the nodes has a weight associated with it, which essentially just tells each node how much to "value" a given input. There are a few ways in which the nodes and links of an ANN can be structured, and each type of structure (or *topology*) results in different computational properties. Here, we will specifically talk about *feed-forward* networks as this is the structure we will be using in our experiments. The other main type of network is the *recurrent* ANN, and often has a more complex topology.

In a feed-forward network, links are unidirectional and there are no cycles — a feed-forward network is a directed acyclic graph. The network is organized in layers, with each node only linking to nodes in the next layer. Although this structure is much simpler than that of a recurrent network, feed-forward networks have sufficient computational abilities for most pattern recognition and classification problems [25]. A simple illustration of a feed-forward ANN with one hidden layer can be seen in Figure 2.2.

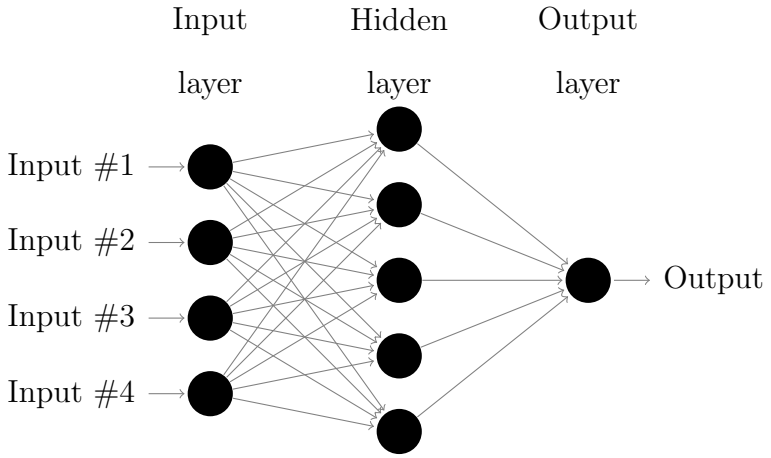


Figure 2.2: A simple feed forward ANN

With this network structure, learning is just the process of tuning the weights of the links to better fit the data in a training set. For example, suppose we have

a network which we are training to identify handwritten numbers. If, during training, it identifies a number as a '2' when it is really a '3', it can make a very small adjustment in its weights to get a little closer to thinking it is a '3'. Over a large data set, these small adjustments add up to create a rather well-generalized network. Statistically speaking, the ANN is an abstracted, finely tuneable nonlinear regression of the training data [24].

In [26], Burger *et al.* provide a method of integrating a feed-forward ANN with MCTS — specifically, they integrate the ANN into the UCT tree policy. The ANN is used to *prune* the search tree as the agent searches. That is, it determines that certain portions of the tree are undesirable and removes them from the search space. Assuming this is done accurately, the tree policy becomes more efficient as it does not spend time expanding the tree in an undesirable area. A number of pruning schemes are tested, and they determined that exponentially decreasing how much of the tree is pruned as the game goes on performs best. Burger *et al.* did not directly test their pruning algorithm against any other agents — their research only showed that an agent with an exponentially decaying pruning policy outperformed agents with other pruning policies.

2.3 Neuroevolution

Neuroevolution is a type of machine learning algorithm which uses genetic algorithms to train and evolve neural networks. In its most basic form, neuroevolution uses evolutionary methods to fine-tune the weights of an ANN whose topology has already been established [24]. This, of course, will improve the performance of a given ANN over time. However, the weights of the links between nodes are not the only thing affecting the performance of the network — just as important, if not moreso, is the actual structure of the ANN [27].

In [27], Stanley and Miikkulainen introduce a neuroevolution method called *Neural Networks through Augmenting Topology* (NEAT). As its name would imply, NEAT is a method of augmenting the topology of neural networks using evolutionary methods. In NEAT, a population of neural networks is evolved throughout the training process to improve its classification abilities. Rather than only tune the weights of a network, NEAT trains the network by also changing the structure of the network.

Prior to the evolutionary training process, the number of input and output neurons are defined — the input and output layers are not changed during training. The type of activation function to be used is also defined. No other structure is given to the network, and a population of networks are generated, each with a random structure of hidden neurons, neuron links, and weights. During training, the worst performing networks are replaced with children of the best performing networks periodically, until an appropriately performing network is found. A more in-depth explanation of the methodology behind this technique is described in [27].

Hypercube-based NEAT (HyperNEAT) is an extension of NEAT. The HyperNEAT extension uses a similar technique as NEAT, but is specifically meant to be used on very large scale neural networks. HyperNEAT trains and evolves neural networks to recognize patterns in data such as repetition or symmetry by using *Compositional Pattern Producing Networks* rather than standard ANNs. The most unique and novel aspect of HyperNEAT is its ability to create networks which understand the geometry of the problem. [28]

When a normal ANN is used to represent a game board such as Go, it has no concept of the nodes representing spaces set up in a grid; it doesn't actually know where pieces physically are in relation to one another without figuring it out on its own. HyperNEAT, though, utilizes what is called a *substrate layer* to provide the input and output layer with a representation of the board's geometry. HyperNEAT

creates network connections based on this physical board geometry. HyperNEAT evolves the network connections over time with a particular board structure already understood prior to training. Since there are hypothetically several ways to represent the geometry of the game board for any given game, there are several different ways to train a network with HyperNEAT. For a further explanation of HyperNEAT and an analysis of the algorithm's performance, see [28].

Figure 2.3 shows a neural network for an arbitrary problem at two different stages in HyperNEAT — generation 30 of the network is shown on the left, and generation 106 of the network is shown on the right. The figure shows how a grid-like structure is given to the network. Note the differences in connection weight, number of hidden neurons, and overall network structure between the two generations of the network.

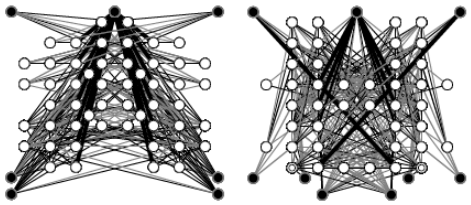


Figure 2.3: A network at two different stages in HyperNEAT [31]

Gauci and Stanley demonstrate how both NEAT and HyperNEAT can be used for Go in [29]. They demonstrate that an agent utilizing HyperNEAT is able to intelligently play Go, and more importantly, it is able to scale to larger board sizes after only being evolved on a 5x5 board. While they do not use MCTS in their implementation, they do state that it is very possible to bootstrap MCTS with a tree policy evolved by NEAT or HyperNEAT. Specifically, they state these algorithms could be used to evolve a more effective default policy for UCT. This is precisely what we do in our implementation.

2.4 Deep Convolutional Networks and AlphaGo

Deep Convolutional Networks (DCNs) are another class of machine learning algorithms closely related to ANNs. DCNs are a subset of feed-forward ANNs which contain a number of what are called *convolutional* hidden layers, and are traditionally created for image recognition. However, as seen in Google's Go playing machine AlphaGo [13], DCNs can be adapted for other tasks as well.

DCNs are far more complex than a simple feed-forward ANN, and to understand them it is perhaps best to think of each layer as a square of nodes rather than a line (Figure 2.4). Each layer still only directly interacts with the very next layer, but each hidden node will only be connected to a small region or window of the input nodes. This window then "slides" over the input layer for each hidden node, with overlap. This concept is awkward to explain in words, but Figure 2.5 represents how this works graphically.

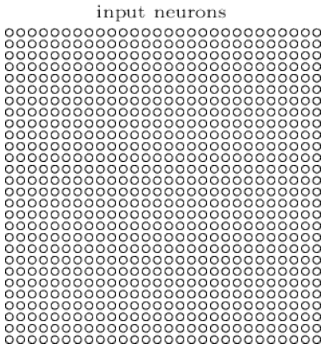
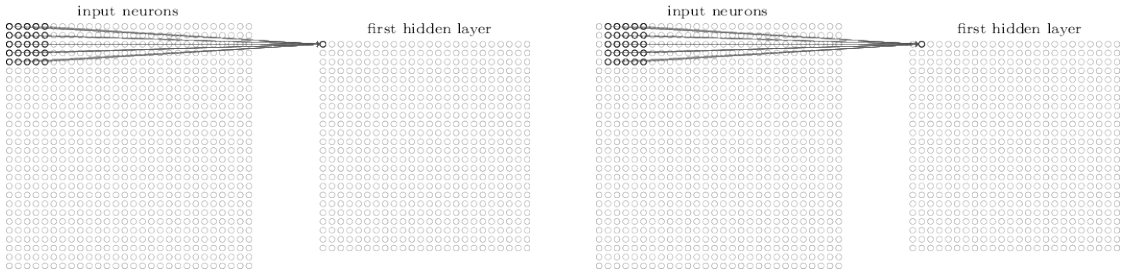


Figure 2.4: DCN input layer [30]

This mapping of small windows of nodes onto a single node through several hidden layers is repeated over a number of layers. After the convolutional layers, there are a series of *pooling* or *subsampling* layers. These are similar to the convolutional layers in that they still link a region of nodes onto a node in the next layer, but in much smaller groups, perhaps 2x2. The output of each pooling layer is usually either the

maximum or the minimum value from its grid of inputs. The goal of this process is to simplify the output of the convolutional layers.



(a) Links from input layer to hidden node 1 (b) Links from input layer to hidden node 2

Figure 2.5: Links in a convolutional network[30]

The pooling layers are linked with another layer of convolutional layers in the same way convolution was described before, and this back-and-forth between pooling and convolution is done how ever many times is wanted. This process as a whole is called *feature extraction*, because it allows the network to break a picture down into a representation of its most basic features. After feature extraction, the final layers are connected to a standard, fully connected ANN which produces the output, usually a classification of the image.

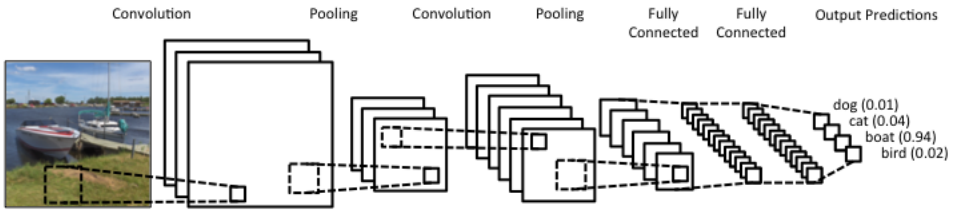


Figure 2.6: An example of a full DCN [32]

Perhaps the most famous implementation of a DCN with MCTS is Google DeepMind's Alphago [13]. AlphaGo is designed by Google DeepMind which plays the board game Go, and in 2015, it became the first computer program to beat a professional human Go player without handicaps on a full board. AlphaGo uses MCTS

in conjunction with three different DCNs — two are called "policy networks" and essentially alter the tree and default policy of MCTS, while the third is a "value network" and helps determine the value of the current game state. Each network had 13 hidden layers.

AlphaGo was trained in two phases. First, the network was trained on millions of moves from professional Go games. This training was done until the program could predict a human move 57% of the time. The second phase of training involved AlphaGo playing itself and, using reinforcement learning, discovering new strategies for itself. The result was an incredibly intelligent system — AlphaGo was able to beat the next best Go AIs in 499 out of 500 games played, even when giving the other programs a four move headstart.[13]

Unfortunately, creating an agent as sophisticated as AlphaGo was simply not feasible given our timeframe and resources. Training a deep convolutional neural network requires training sets which are many magnitudes larger than what is necessary for more simple ANNs. While Google has the resources necessary to provision such large datasets, we do not. Furthermore, assuming we did have access to an appropriate amount of training data, training a DCN with 13 hidden layers would require a massive amount of processing power and would be a long, arduous process.

2.5 Encog Machine Learning Library

To ease the integration of each machine learning algorithm into MCTS, we utilized Heaton Research's Encog machine learning library for Java [42]. The library has been in active development since 2008, and has been used in a variety of other well-cited research publications since its release (e.g. [44, 45, 46]). We chose this library because of its wide use, thorough documentation, and large number of example programs provided.

Encog provides functionality for most common machine learning algorithms, including genetic algorithms, artificial neural networks, and NEAT/HyperNEAT. Creating and training these structures is quite easy. For example, to create and train a simple ANN, you simply create a `BasicNetwork` object, define the network's structure, create a training dataset, and run `ResilientPropogation.iteration` on the network and training data to perform a training epoch. Implementing other machine learning algorithms is just as simple.

We create a general-purpose game-playing agent which uses MCTS as its decision-making algorithm. Through heavy use of the Encog library, we create three other general-purpose game-playing agents — each agent uses either a GA, ANN, or nuero-evolved NEAT network to modify the original MCTS algorithm. The next chapter outlines these agents in more detail, and also introduces our implementation of a game-playing framework which we use to compare the agents' performance.

Chapter 3

Implementation

Our implementation consists of a new game-playing framework, the games Go and Hex, and a total of five game-playing agents. While the code for each game is obviously very game-specific, the game-playing agents and games framework was made to be very general purpose. By heavily employing inheritance and polymorphism, the framework and agents are hypothetically able to run/play any perfect information game with very little change to the existing codebase, assuming a properly defined game class is made. A very high-level description of the structure of the code is given below, and Figure 3.1 provides a high-level graphical representation.

1. The games framework orchestrates the playing of two-player turn-based board games between either human or computer players; the framework hosts the master game board and agents, ensures legal play, and records data about the gameplay if desired

2. Each game-playing agent is an extension of `GameAgent.java` and employs a different general game-playing strategy; the agents interact with the master game board and utilize the game classes to help it make decisions on the board

3. The game classes are extensions of `Game.java` and provide game-specific functionality; these classes are used by both the games framework and game-playing

agents to run and play games

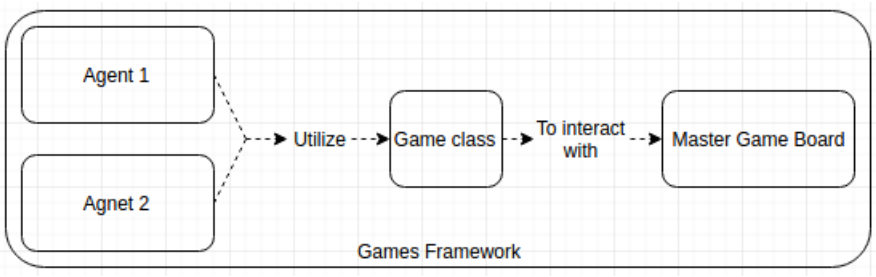


Figure 3.1: High-level structure of gameplay in our framework

This chapter provides an overview of the implementation of the Go and Hex classes, the game-playing agents, and the games framework. Furthermore, we outline our experiments and the methods of running these experiments; the results of the experiments will be discussed in the next chapter. In each section, we provide both high-level descriptions of the relevant code structure as well as in-depth, low-level discussion of any important algorithms. The chapter is organized such that the entire code structure is explained from the inside out. That is, we first describe the games, then the agents which play the games, and finally the framework which orchestrates the playing of the games. This is because an understanding of each of these parts of the implementation is beneficial to understanding the following parts.

Before the discussion of the more involved classes, we need to first explain the `Board.java` class, a very simple object which is used in almost every other class. The `Board` object provides a representation of a game-independent game board — there is no game-specific functionality in a `Board` object. A `Board` consists of a 2-dimensional `char` array which represents a physical game board, and an `int` to hold the size of the board.

The constructor for `Board.java` takes an `int` as input which sets the size of the board. The constructor then initializes the 2-dimensional `char` array with the

number of rows and columns both equal to the size of the board. Every index of the array is set to the dash character ('–') to represent an unoccupied space of the board. When a player places a piece on the board, an 'X' or 'O' is placed at the appropriate index of the array depending on whether the move is made by the first or second player, respectively. `Board.java` consists of instance methods to get/set each instance variable, print the board to the console, and place either an 'X' or an 'O' at a given index.

3.1 Games

Go and Hex each have a corresponding java class: `GoGame.java` and `HexGame.java`, respectively. Both of these classes are extentions of the abstract class `Game.java`. The purpose of these classes is to provide the game-playing agents and games framework with a set of helper methods. `GoGame.java` and `HexGame.java` provide game-specific functionality which aids the agents in their move-making process, and helps the manager track the score of the game and change the board between moves, if necessary.

`Game.java` consists of the following abstract methods which are overridden in `GoGame` and `HexGame`:

- Name: `getPossibleMoves`

  Arguments: `Board board, boolean firstPlayer`

  Returns: `ArrayList<Board>`

  Given a `Board` and a boolean indicating which player's turn it is on, this method returns an ArrayList of possible boards for the player to move to

- Name: `gameFinished`

  Arguments: `Board board, int moveNumber`

  Returns: `boolean`

Returns true if the game should end for the given `Board` and move number

- Name: `calculateScore`

  Returns the score of the given `Board`

  Arguments: `Board board`

  Returns: `int`

  Calculates and returns the score of the given `Board`

- Name: `randomPlayout`

  Arguments: `Board board, boolean firstPlayer, int moveNumber`

  Returns: `int`

  Performs a random playout from the given `Board` and returns the score

- Name: `randomBoardAfterXMoves`

  Arguments: `int boardsize, int moves`

  Returns: `Board`

  Generates a `Board` of the given size after the given number of random moves have been made on an empty board

- Name: `resolveBoard`

  Arguments: `Board board, int boardSize`

  Returns: `Board`

  Alters the board after a move is made if needed (e.g. if a move in a game of Go results in a piece being surrounded, this will remove the surrounded piece from the board) and returns the altered `Board`

For both `GoGame` and `HexGame`, the `getPossibleMoves` method is implemented by simply iterating through the board, and attempting to place a piece on each spot as seen in Figure 3.2. The `gameFinished` method is implemented differently for `GoGame` and `HexGame`. The method will return true in `GoGame` if the moveNumber

argument exceeds a set number (i.e. the max number of moves has been played; for our experiments we set this to 250), or if both players have no legal moves remaining on the board. `HexGame` returns true if the board is completely filled, or the win condition is met for one of the players (this is checked using a simple depth-first search from each edge of the board to the opposite side).

```
   input  : Board b, boolean firstPlayer
   output: ArrayList<Board>
 1 Initialize empty ArrayList<Board>;
 2 for each space on b do
 3     if char at current space on b == '–' then
 4         create copy of b;
 5         place current player's piece at current space on copy of b;
 6         copy of b = resolveBoard(copy of b);
 7         Add copy of b to List;
 8     else
 9         // do nothing
10     end
11 end
12 return ArrayList of boards;
```

Figure 3.2: getPossibleMoves pseudocode for both `GoGame` and `HexGame`

For `HexGame`, the `calculateScore` method simply returns 1 if the first player wins and -1 if the second player wins — recall that it is impossible for a game of Hex to end in a tie. For `GoGame`, score is calculated using the stone-scoring scheme described in Chapter 1, and has a trivial implementation. The `randomPlayout` and `randomBoardAfterXMoves` methods also have a trivial implementation which utilizes

`getPossibleMoves` and Java's `util.Random` for both games, and `resolveBoard` simply returns the `Board` argument in `HexGame` (there is never a need for board resolution in Hex as pieces are never removed).

However, the `resolveBoard` method for `GoGame`, which is used to remove surrounded regions of stones from the board, is more involved. The algorithm we created for this is based on the flood-fill algorithm [41] which is commonly used for the "paint bucket" tool of image editing software to fill similarly colored regions with a new color. A recursive approach is commonly used for this type of algorithm [41], but for the problem of resolving Go boards, we decided an iterative Queue-based approach would be a better method. A board is resolved by running the algorithm outlined in Figure 3.3 for each occupied space of the board. Note that in Figure 3.3, a "pair" refers to an (x,y) coordinate pair corresponding to an index of the board.

Essentially, we move outwards along the row of the starting Pair in both directions until the char at both points on the board is no longer equal to the char at the starting point. We replace every point we visit with a placeholder (the char 'A'), and then repeat these actions for the Pairs above and below each visited Pair if their char is equal to the starting Pair. We do this until we have no more Pairs on which to perform these actions. If any Pair we visit contains a '–', we know the region is not surrounded and we can return the original Board. If we never visit a Pair containing a '–' before exhausting all the connected Pairs, we know the region is surrounded; in this case we remove all the 'A's from the board, and return.

3.2 Game-playing Agents

Similar to our games, each of the game-playing agents are the children of an abstract class called `GameAgent.java` (Figure 3.4). A `GameAgent` contains a `Game` object to help it make moves, a boolean tracking whether the agent is the first or second player,

```
   input  : Board b, Pair p
   output: Board
 1 char c = char at p;
 2 Initialize empty Queue<Pair> q;
 3 q.enqueue(p);
 4 Create Pairs w and e;
 5 while (!q.isEmpty()) do
 6     Pair pair = q.dequeue();
 7     Set w and e equal to pair;
 8     Decrement x-value of w until char at w != c;
 9     Increment x-value of e until char at e !=c;
10     for each Pair n between w and e do
11         if char at n == '–' then
12             return original board (area not surrounded);
13         end
14         Replace char at n with 'A' (placeholder);
15         Enqueue Pair north of n if its char == c;
16         Enqueue Pair south of n if its char ==c;
17     end
18 end
19 Replace every 'A' on the board with '–';
20 return resolved Board;
```

Figure 3.3: Board resolution algorithm for `GoGame`

an int to track how many iterations of its search algorithm it has gone through (if applicable), and a Random object. In total, we have five agents extending `GameAgent`. The first is a trivial agent which simply makes random moves, called `RandomAgent`. The second is `MCTSAgent` and uses MCTS with a UCT tree policy to inform its move selection. The other three use MCTS in conjunction with either a genetic algorithm (`GAAgent`), an artificial neural network (`ANNAgent`), or the NEAT algorithm (`NEATAgent`) to modify the search and move selection process. Each non-trivial agent contains a `makeMove` method which spends a given amount of time deciding what move to make on a given `Board`, using various helper methods within the agent.

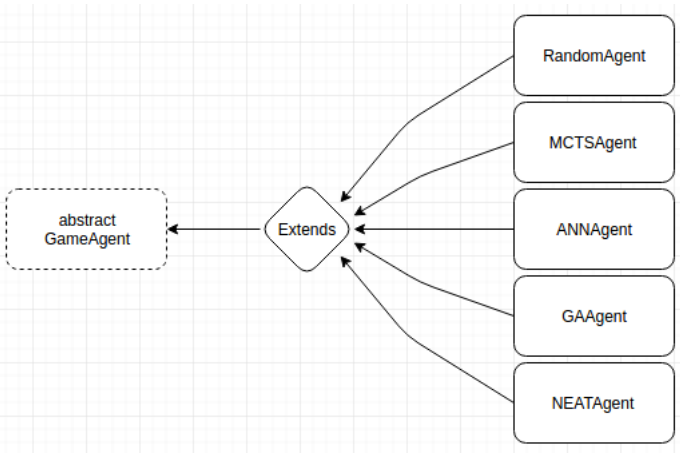


Figure 3.4: Hierarchy of agent implementation

The four non-trivial agents utilize MCTS as the base of their decision-making algorithm, and each have a different Tree policy or Default policy from one another. Each of the four non-trivial agents follow the four steps of MCTS as outlined in Chapter 1, but each agent may have a slight alteration or addition to the algorithm. In this section, we describe the way in which we implemented the base MCTS algorithm and `MCTSAgent`, and then we will look at how each agent modifies this algorithm. Note that in this next subsection, we will refer solely to the implementation of the `MCTSAgent` — however, everything stated directly applies to the other three non-

trivial agents as well unless directly stated otherwise when discussing the modified agent.

3.2.1 Implementing Monte-Carlo Tree Search

The main function of MCTS involves incrementally building a tree of possible moves within the game. For this, we created a `Node.java` class which behaves slightly differently from how one may expect a generic tree-node to, since it must seperate the children which have been visited in MCTS from those which have not. Each `Node` object has the following instance variables:

- `double score`

  Tracks the number of random playouts from this node resulting in a win

- `double games`

  Tracks total number of random playouts performed from this node

- `ArrayList<Node> unvisitedChildren`

  Contains all children which have not been visited by MCTS

- `ArrayList<Node> children`

  Contains all children which have been visited by MCTS

- `ArrayList<Node> prunedChildren`

  Contains nodes pruned (only applicable for ANNAgent)

- `Node parent`

  Points to node's parent

- `Board b`

  The `Board` associated with the node

- `boolean firstPlayer`

  Tracks who's turn it is on the `Board` represented by the node

`Node.java` has two constructors, one for the root node of the tree (which is only called on the first iteration of MCTS each turn), and one to create all other nodes. The root constructor takes a `Board b` and `boolean firstPlayer` as arguments and sets the respective instance variables to these values; `score` is set to 0.0, `games` is set to 1.0, the children ArrayList is initialized, and its parent is set to `null`. The other constructor takes a `Board b` and `Node parent` as arguments, and sets the respective instance variables to these. The only other difference from the root constructor is `firstPlayer` is set to the opposite of its parent's value. Each `Node` has instance methods to find and set its unvisited children, recursively backpropogate scores from random playouts up the tree to the root node, find the UCT value of a node (given an exploration bias parameter as described in Equation 1.1), and prune nodes from the tree (i.e. remove them from children and/or unvisitedChildren).

Utilizing the `Node` object, the `MCTSAgent` is able to build a game tree using MCTS as described in Chapter 1. In addition to all the instance variables defined in `GameAgent`, the `MCTSAgent` also has a `double explorationConstant` for use in the MCTS algorithm — for our experiments, this constant was set to $\sqrt{2}$ which provides a rather equal balance between exploration and exploitation [16]. Its `makeMove` method takes a `Board b`, `int timeAllowed`, and `int moveNumber` as arguments, which are the current Board, amount of time it is allowed to take to decide on a move, and how many moves have been made in the current game (which is used for random playouts), respectively.

Within the `makeMove` method is a while-loop which repeatedly calls the agent's `select` method over its given time allowance. This method runs through a single iteration of MCTS using various other helper methods — it selects the node in the

tree with the highest UCT value, visits a random one of the node's unvisited children, finds the score of a random playout, and backpropogates it the score up the tree. Note that with our chosen scoring scheme for Go, the scores of the random playouts can theoretically fall anywhere in the interval $[-s, s]$ where s is the number of cells on the board (e.g. a 9x9 board can have scores anywhere from -81 to 81); however, UCT requires estimated node scores to be in the interval $[0, 1]$ [16]. Furthermore, for player two, lower scores are better. Thus after finding the score of a random playout from a node, we must alter the score before backpropogating it up the tree.

This was a simple process. First, if the player performing the random playout was the second player, we multiplied the estimated score of the playout by -1. Then the score was normalized using Equation 3.1, where s is the score of the game on a $b \times b$ Go board. s' is then the score which is backpropogated up the tree.

$$s' = \frac{s + b^2}{2b^2} \tag{3.1}$$

After the agent spent its alotted amount of time searching the tree, it makes its final move selection. For final move selection, the agent simply chooses the child of the root with the highest average playout score (i.e. the child with the highest score divided by total games).

3.2.2 ANNAgent

The `ANNAgent` behaves exactly the same as the `MCTSAgent` with one caveat: prior to building the game tree with MCTS, it prunes a number of the root's immediate children from the tree. This is done by training a simple feed-forward network to rank a node's children in order of how likely they are to be repeatedly visited by MCTS — those with the lowest scores are not included in the search process. An exponentially decaying pruning scheme is used as outlined in [26], meaning the number of nodes

pruned from the tree exponentially decays as the game progresses. Using equation 3.2 to determine the number of nodes to prune, the agent removes 50 percent of children at move 0, 10 percent of children by move 70, 4 percent by move 125, etc.

$$(\%\text{ pruned}) = 50 - 48.6 * (1 - e^{(-0.0244 * x)}) \tag{3.2}$$

Seperate neural networks were created and trained for each boardsize of both Go and Hex prior to any experimentation. In total, 8 neural networks were made (9x9, 11x11, and 14x14 Hex; 5x5, 7x7, 9x9, 11x11, and 13x13 Go). For this, the Encog machine learning library was used [42]. The neural networks were designed as follows for each game on each $n \times n$ board:

- Input layer consisting of $n \times n$ neurons — each neuron corresponds to exactly one spot on a board;
- A single hidden layer consisting of the number of neurons in the input layer divided by 3;
- Output layer consisting of a single neuron, representing the estimated value of the input board.

Only sigmoid activation functions were used in the neural networks, as this was found to provide the best performance for evaluating Go boards by Burger *et al.* [26]. We chose a single hidden layer as Heaton, the creator of Encog, suggests that more than a single layer is completely unnecessary for training simple feed-forward networks [42]. Finally, we settled on the size of the hidden layer empirically, by briefly testing several hidden layer sizes between that of the input size and output size.

After designing the networks, each network was trained on a seperate set of 2500 randomly generated boards of the network's respective game type and board size. Each set contained an equal number of boards representing random games after 5

moves, 10 moves, 15 moves, etc. up to 105 moves. In order to estimate the value of each of these boards, an `MCTSAgent` performed 10,000 iterations of MCTS for each board; the UCT value of each board after 10,000 iterations was used as the ideal output value for the input board.

We chose to train each network on this dataset until the network's error was below 0.0001 — this seems like an incredibly low amount of error, but such a low error was actually necessary to guarantee any level of accuracy from our networks. Because of our randomly generated inputs for each dataset and the large number of MCTS iterations, most of the ideal values were found to be between 0.45 and 0.55. Such a small range in most of the training data meant that a calculated level of error would in actuality be magnitudes higher. For example, if the network were to classify every single input as having a value of 0.5, the error would likely only read to be around 0.05 since most of the ideal values are between 0.45 and 0.55 — such a network is clearly far from being accurate, despite having a seemingly low error over the training set.

After being trained, the networks were each serialized using Java's serialization library. The `ANNAgent` has a `BasicNetwork` instance variable (imported from the Encog library) which is initialized upon creation of the agent. Before gameplay, the agent chooses the appropriate network to deserialize based on the gametype and size of the game board. On each turn, the agent then uses this network to evaluate moves and remove a percentage of them from the game tree, as described earlier.

The process of evaluating each child of the current board, sorting them by estimated ranking, and removing a set number of them from the tree obviously takes some of the agent's alotted time that could be spent performing more iterations of MCTS. So, what is expected to be gained by this process? Essentially, the idea behind the `ANNAgent` is that, while it performs a lower number of iterations of MCTS, more

of the iterations it does perform will be done exploring high quality moves. If we can identify which children will not be explored heavily before even starting MCTS, the number of iterations lost from time spent evaluating children may be overcome by the fact that any iterations of MCTS that would have been wasted exploring those moves can now be spent on deciding between fewer, better options.

3.2.3 GAAgent

The `GAAgent` utilizes a genetic algorithm to bias both its tree policy and its final move selection. The agent's genetic algorithm consists of a population of `GAWeight` objects, which each hold a weight vector of five doubles $(x_1 \cdots x_5)$ between zero and one to represent the weight given to a hueristic property of the game board. Tables 3.1 and 3.2 give the names and descriptions of each of the games' heuristic properties which are used. Each `GAWeight` also contains methods to evaluate each heuristic property of a given board.

Property name	**Definition**
netStones	Score of resulting board
goodLibs	Number of own liberties on board
badLibs	Number of opponent's liberties on board
goodAtari	Number of own stones in atari
badAtari	Number of opponent's stones in atari

Table 3.1: Properties used for Go genetic algorithm

For example, consider Figures 3.5 and 3.6. These are representations of small boards from Go and Hex, after several moves have been made — the boards were generated using GoGui [20] and HexGui [14], respectively. On the Go board, the blue dots represent white stones' liberties while the red dots represent black stones'

Property name	Definition
netBridges	Net number of bridges on board
goodConnected	Size of largest connected region of own stones
badConnected	Size of largest connected region of opponent's stones
goodDeepest	The smallest number of agent's moves needed to win
badDeepest	The smallest number of opponent's moves needed to win

Table 3.2: Properties used for Hex genetic algorithm

liberties; stones marked with green dots represent stones in atari. Meanwhile on the Hex board, the blue shaded region represents the largest region of black stones, the red shaded region represents the largest region of white stones, green lines represent any bridges, and the empty spaces marked with 'X' represent the spaces where either player would need to play to end the game.

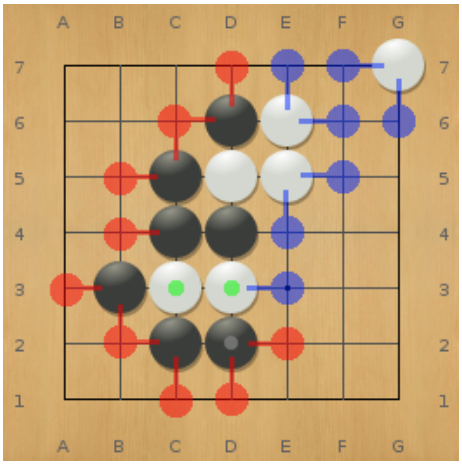


Figure 3.5: Go heuristic properties

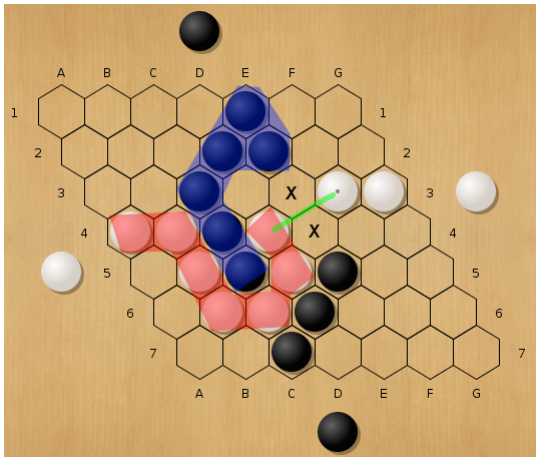


Figure 3.6: Hex heuristic properties

So, suppose a `GAAgent` is playing as black in both of these games and one of its `GAWeights` is evaluating the boards. For Go, the `GAWeight` would find a netStones value of -1 (6 white stones and 7 black stones), a goodLibs value of 9, a badLibs value of 7, a goodAtari value of 0, and a badAtari value of 2. If $(x_1, \cdots, x_5)$ are the

respective weights for each property, the `GAWeight` would find the overall value v_1 of the Go board to be

$$v_1 = -1x_1 + 9x_2 + 7x_3 + 0x_4 + 2x_5.$$

For the Hex board, the `GAWeight` would find a netBridges value of 1 (1 white bridge and 0 black bridges), a goodConnected value of 6, a badConnected value of 7, a goodDeepest value of 2 (both spots marked x needed to win for black), and a badDeepest value of 1 (only one spot marked x needed to win for white). So, the `GAWeight` would find the overall value v_2 of the Hex board to be

$$v_2 = 1x_1 + 6x_2 + 7x_3 + 2x_4 + 1x_5.$$

When a `GAAgent` is created before the start of a game, it is given a small population of 10 `GAWeights` with randomized weight vectors $(x_1 \cdots x_5)$. While, traditionally, a genetic algorithm requires each member to fully complete its task (e.g. play a full game of Go) in order to evaluate a member's fitness, the nature of MCTS allows us to to evolve the weight vectors many times during a single game [17] — rather than use a full, real playout of a game the fitness of a `GAWeight`, we can use the simulated playouts of a game during each iteration of MCTS.

We evaluate, select, crossover, and mutate the population of `GAWeights` after every 100 iterations of MCTS; each of the ten `GAWeights` is used ten times to bias the `GAAgent` before evaluation. So, when the `GAAgent` is asked to make a move, it selects a node of the tree to expand taking into account both the UCT scores and estimated board values of each node. A random playout is performed from the chosen node as usual, and, for the purpose of fitness evaluation, the score of the playout is stored in whichever `GAWeight` was used. After every 100 iterations of MCTS, each `GAWeight` will have been used 10 times and we can perform the evolutionary algorithm

on the population.

Evaluating the fitness of the population is simple; we look at the saved playout scores of each `GAWeight`, and a higher average score equates to a higher fitness. The two `GAweights` with the lowest fitness are removed from the population and are replaced by two children of the highest performing `GAWeights`. These children are made by simply averaging the weight vectors of the parents — prior to mutation, the two children are exactly the same. Mutation of each child is done independently on each of its weights — if a weight is chosen to be mutated, it is averaged with a random double between 0 and 1.

For the first child, each weight has a 20% chance of being mutated, while each weight of the second child has a 60% chance of being mutated. So, on average, one of the first child's weights will be mutated while three of the second child's weights will be mutated. We chose these high mutation rates because of the small population size and the small number of random playouts performed relative to the branching factor of each game. A high mutation rate keeps the population diverse and allows the agent to more easily discover heuristic strategies far different from those in the starting population.

After the `GAAgent`'s alotted time to decide on a move is up, it biases its final move selection as well. The agent assigns a score to each possible move in the same way as the `MCTSAgent`, biases each move with the highest performing `GAWeight` in its population, and selects the move with the highest final score. During both the selction phase of MCTS and final move selection, the `GAAgent` essentially introduces a small amount of heuristic analysis into its decision making in order to differentiate between nodes in the tree with similar UCT scores. We observed that during MCTS, there are oftentimes several clusters of nodes which have similar UCT scores. Hypothetically, the `GAAgent` should be able to choose between similarly scoring moves

more accurately; this could be particularly helpful when the agent is only able to run through a small number of iterations of MCTS (e.g. early in a game of Go on one of the larger boards).

3.2.4 NEATAgent

Rather than alter the MCTS tree policy or final move selection like the previous agents, the `NEATAgent` uses an evoloved neural network to alter the default policy of MCTS as proposed by Gauci and Stanley in [29]. The logic behind this agent is that, when playing against a non-trivial player, a random playout will likely not result in a very accurate representation of a game playout. Thus, a default policy which chooses moves more intelligently will allow the MCTS algorithm to better estimate the scores of playouts during the simulation phase.

Gauci and Stanley proposed using NEAT to create a network which acts as an action selector — given a board, the network provides each empty space with a score based on how valuable placing a piece at that spot would be. The Encog framework [42] provides NEAT functionality and was used to implement this system. Similar to the ANNs created for the `ANNAgent`, a different network was created for each boardsize of each game, and each have an input layer size equal to the number of spots on the board (e.g, 81 input neurons for a game on a 9x9 board). Because these networks are acting as an action selector, the size of the output layer is also equal to the number of spots on the board — each output neuron represents the value of placing a piece at the spot represented by the input neuron in the same place. The networks were limited to using only sigmoid activation functions.

To train the networks, we again randomly generated 2500 boards of the appropriate game and board size for each. For each board, an `MCTSAgent` performed 5000 iterations of MCTS using the board as the root of the tree. This allowed us to es-

timate the value of each legal move by looking at the score of each of the board's children. The ideal value for the output of the board was set to the corresponding move's score; any output neurons which represented spaces which were not valid moves had the ideal value set to 0.

After the training dataset was created, each network was trained until its error was below 0.1. The networks were then serialized using Java's serialization library, and the `NEATAgent` contains a `NEATNetwork` instance variable, similar to the `ANNAgent`. Upon creation, the `NEATAgent` can pick the appropriate network to use, and assign the network to this instance variable. During MCTS, the `NEATAgent` has a standard selection and expansion phase. For the simulation phase, though, the agent uses a built-in playout method called `biasedPlayout` rather than the random playout functionality provided in the games classes.

The `biasedPlayout` method takes a `Node` as input (the node which is expanded to during MCTS), and returns a score of a playout from this `Node`. During each move of the simulated playout, the agent evaluates the current board with its network and places a piece at the spot corresponding to the output neuron with the highest value. The agent's MCTS process is otherwise the same as the standard `MCTSAgent`, calculating the score at the end of the playout and backpropogating it up the tree. The agent's final move selection is also the same, selecting the move with the highest ratio of average-playout-score to total-games-simulated.

3.3 Games Framework

`GameController.java` is the "runner" class of our framework and orchestrates the actual playing of games — it is used to create game and agent objects, run the games, and record data. In order to run this program, note that you must include the `encog-core-3.3.0` and `jcommander-1.48` JARs included in the `/lib` directory

of the project. The class contains a `Game` object, a `Board` object which acts as the master game board, two `GameAgent` objects, and several `FileWriters` to record data. We parse command-line inputs using the JCommander library [43] to determine gametype, board size, which agents are playing, etc. (Table 3.3). Our framework also supports human play by printing the game board to the terminal between moves, and allowing human players to make moves by typing the row and column of the space on which they wish to place their piece.

Parameter	**Default**	**Description**
String –agent1	"human"	Player one's agent type
String –agent2	"human"	Player two's agent type
int –size	9	Size of the game board
String –game	"go"	The game being played
int –time	3000	How many milliseconds can be spent making a move
boolean –record	false	Record game data
boolean –help	false	Display JCommander usage

Table 3.3: JCommander parameters for `GameController.java`

Following the creation of all of the necessary objects, `GameController` simply enters a `while-loop` which executes until the game is complete; the game over condition for whichever game is being played is checked following every move. While the game over condition is not met, each of the two players are simply asked to make a move. If the player is human, the game halts until a valid move is typed into the terminal; otherwise, the agent's `makeMove` method is called for the current `Board`, movenumber, and time allowance. After each move, if `--record` has been set to true, the current game score and number of iterations of MCTS the agent was able to complete (if applicable) is written to a .csv file. Figure 3.7 illustrates the entire structure of the

framework during gameplay between two agents.

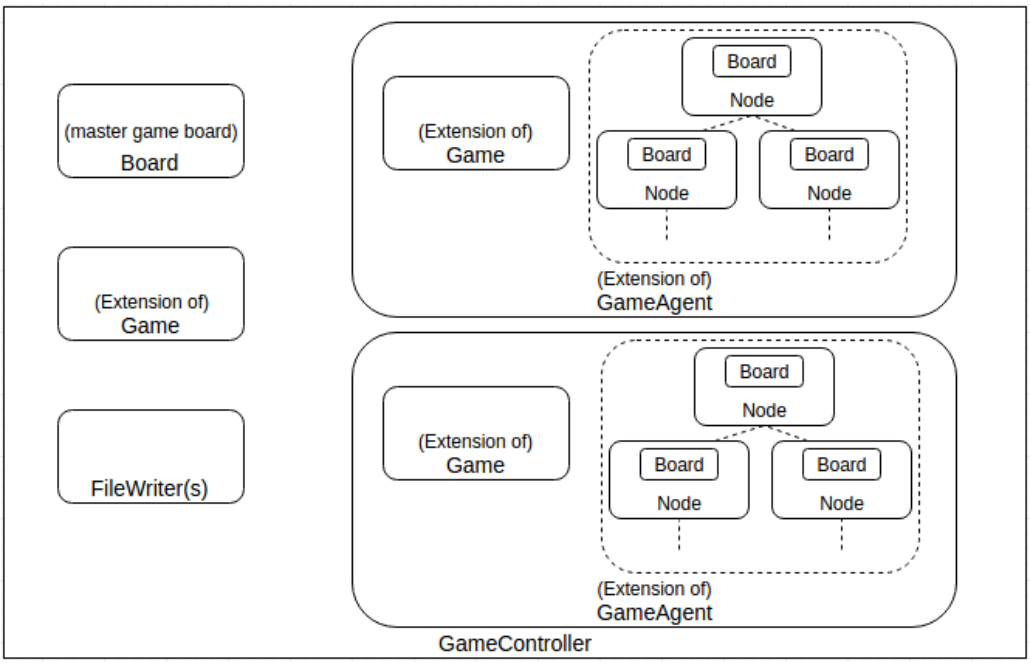


Figure 3.7: Structure of the game-playing framework

The entire framework was written with expandability in mind. The extensive use of object-oriented programming paradigms allows for new games and agents to be integrated into the framework with ease. Adding support for a new game or agent would require no changes to the existing Java classes, other than adding a few lines to instantiate the new object in `GameController.java`.

3.4 Experimental Design

Using our framework, we ran a series of experiments comparing the different agents on equal footing. Each discrete pair of agents played one another multiple times on various configurations of the Go and Hex games, and we recorded both the game score and the number of MCTS iterations the agents were able to complete per turn (if applicable). Oftentimes experiments involving improvements to MCTS are done with

the agents having an equal computational budget — that is, each agent is allowed to take as much time as is needed to perform a set number of iterations of MCTS. Rather than following this experimental structure, we chose to give each agent a time budget.

We believe that this decision makes our experiments much more interesting, and much more applicable to the real world. Performing our experiments with the agents having a computational budget rather than a time budget would indeed tell us which algorithms provide better absolute performance per iteration of MCTS. Limiting the agents by time, though, allows us to look at the tradeoff between the performance and the increased computational complexity of each agent. In other words, a computational budget would allow us to measure an increase in computational power, but a time budget allows us to measure whether or not this increase in power is worth it. In our opinion, this is a much more realistic standard to compare the agents' performance with.

Because of the large number of experiments, we chose to distribute them over multiple machines in Alden Hall and run them concurrently. We achieved this by manually `ssh`ing into a different machine for each pair of agents, and running a simple `bash` script on each to run 20 games between the two agents for each variation in board size and time allowance. Each `bash` script simply sets the classpath to include the necessary JARs, and loops through playing the different games with the `--record` option set to true. We discuss the results of these experiments in the following chapter.

Chapter 4

Experimental Results

Our experiments consisted of having every agent play one another multiple times on different sized boards of each game under different time allowances. Each pair of agents played games on 5x5, 7x7, 9x9, 11x11, and 13x13 Go as well as 9x9, 11x11, and 14x14 Hex. For each of these gametypes, 20 games were played with time allowances of 500ms, 1000ms, 2000ms, 4000ms, and 8000ms. The different board sizes provide different branching factors (e.g. 13x13 Go has a much higher branching factor than 7x7 Go), while the different time allowances allow for each agent to run through more iterations of MCTS on each turn. In the course of our experiments, we were able to uncover several interesting performance trends.

This chapter is organized by game; there is one section covering the reseults from the Go gameplay, and another on the results from the Hex gameplay. In each section, we first note the agents' performances against the trivial `RandomAgent`, then overview the results from each experimental agent's games against `MCTSAgent`, and then finally look at the head-to-head games between the experimental agents. Note that due to the size and number of our data visualizations, we will often cite Appendix A, which houses all of our data visualizations in full, when referencing data. While most of the full visualizations will not be in this chapter, some small snippets will be included.

For some context on the Go score visualizations, consider Figure 4.1 as an example.

Each colored line represents the score of a different game of Go, whose size and time allowance is specified at the top of the Figure. Each line plots the score of a single playout over the course of a 250 move game. A positive score is a good score for the first player, while a negative score is a good score for the second player — so here, a positive score is good for `ANNAgent`, while a negative score is good for `MCTSAgent`. The smooth, gray curve shows the overall score across all plays of the game at each move for the given configuration.

Figure 4.1: Example Visualization

4.1 Go

The results of the games between each intelligent agent and the `RandomAgent` are surprisingly significant. We assumed each of the intelligent agents would vastly out-

Agent Type	Losses to RandomAgent	Total Games
`MCTSAgent`	5	400
`ANNAgent`	6	400
`GAAgent`	7	400

Table 4.1: Intelligent Agent Losses to `RandomAgent`

perform the `RandomAgent`, and the raw win/loss numbers for these games (Table 4.1) seem to indicate exactly this — each agent won between 98% and 99% of their games against the `RandomAgent`. However, the plots tracking the game scores over the course of the games tell a different story. The `MCTSAgent` and `ANNAgent` appear to have a normal performance — they usually take a quick lead and rarely give up points, regardless of board size or time allowance. This, however, was not the case for the `GAAgent`.

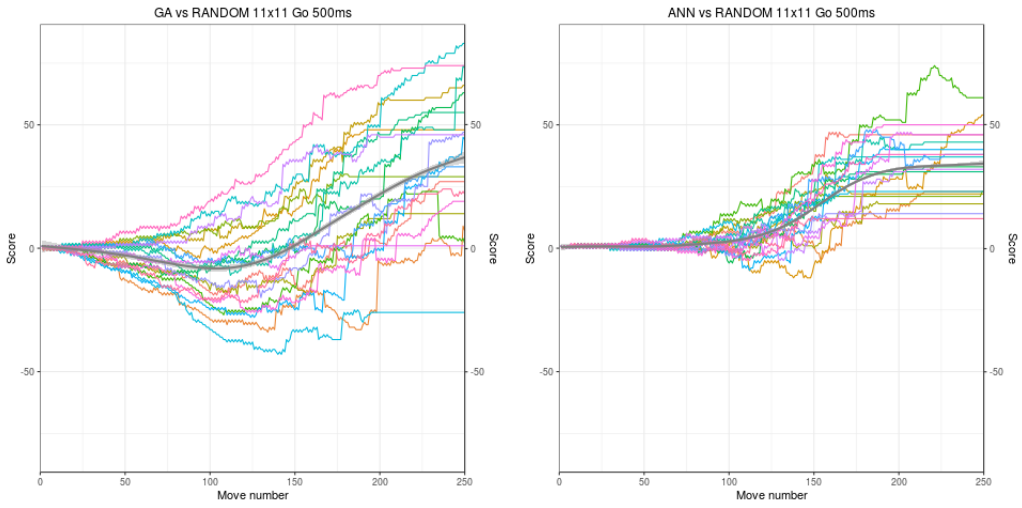


Figure 4.2: `GAAgent`'s and `ANNAgent`'s performance against `RandomAgent`

For large boards and/or a small time allowance, the `GAAgent` has a tendency to actually fall behind the `RandomAgent` for a significant portion of the game. The larger the board and smaller the time allowance, the longer the agent performs poorly;

the agent's performance always seems to vastly improve later in the game, though, leading to a seemingly higher performance. Note the shape of the two graphs in Figure 4.2, which compares the performance of the `GAAgent` and `ANNAgent` against the `RandomAgent`.

The `GAAgent` only loses a single game of this configuration, even ending with a higher average score than the `ANNAgent`. However at turn 150 out of 250, on average the `GAAgent` is actually losing by a significant amount against the `RandomAgent`. Meanwhile the `ANNAgent` is consistently winning almost all of its games at move 150. This can be observed happening between other game configurations in Figure A.7 of Appendix A. This type of performance ended up being a trend for the `GAAgent`.

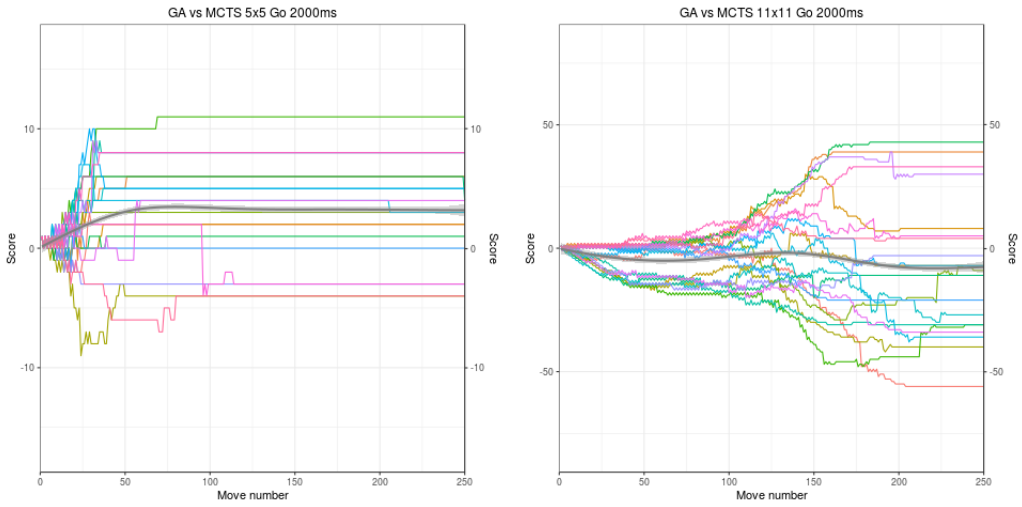


Figure 4.3: `GAAgent` vs `MCTSAgent` performance on different board sizes

The results of the games between `ANNAgent` and `MCTSAgent` (Appendix A Figure A.2) are quite conclusive. The `ANNAgent` consistently slightly outperforms the `MCTSAgent` regardless of board size and time allowance. While the `MCTSAgent` did manage to win a number of games in each board configuration, we can see that the average score favors the `ANNAgent` in 18 out of 20 board configurations. On the two configurations for which the `ANNAgent` did not outperform the `MCTSAgent`, the average

score was 0, favoring neither agent.

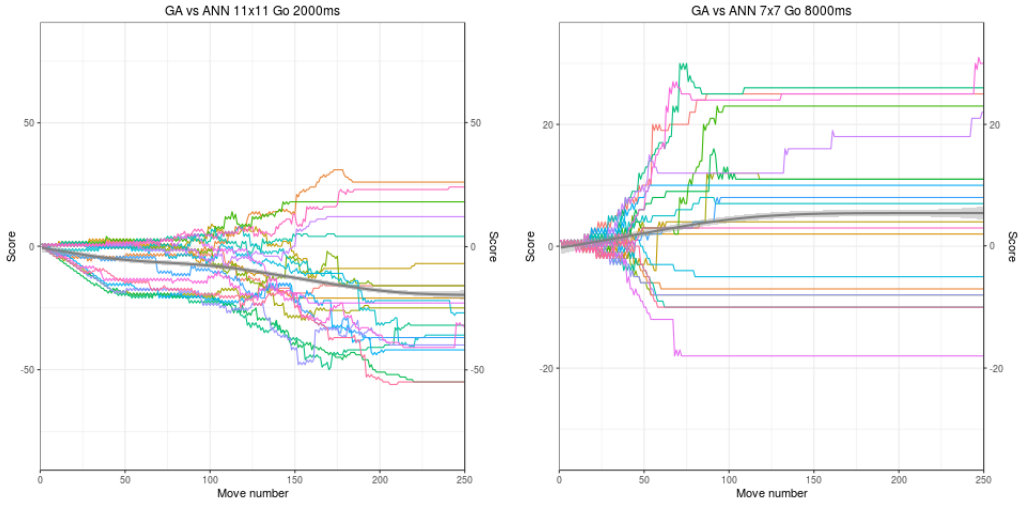


Figure 4.4: `ANNAgent` and `GAAgent` on their more dominant board configurations

The `GAAgent`'s results against the `MCTSAgent` are rather consistent with what was observed against the `RandomAgent` — essentially, performance trends upwards as the time allowance increases, and performance trends downwards as board size increases. An example of this can be seen in Figure 4.4, which shows the scores for two board sizes at the same time allowance. Each row of graphs in Appendix A Figure **??** clearly shows this trend — scores favor `MCTSAgent` more and more as board size increases. The `GAAgent`, though, does tend to outperform the `MCTSAgent`. While not as dominant as the `ANNAgent`, it was able to achieve a higher average score for the majority of board configurations, especially those with smaller boards or higher time allowances.

The head-to-head matchups between `GAAgent` and `ANNAgent` provide the most obvious patterns in our data, and further confirms our suspicions about the `GAAgent`'s performance trend. The `GAAgent` was able to achieve a higher average score on every set of games which had either a 5x5 game board (the smallest size used), or a time allowance of 8000ms (the highest time allowance used). Meanwhile the `ANNAgent`

tended to have a much better performance on boards with a lower time allowance and larger board size, most notably on the 11x11 and 9x9 boards (excluding those with an 8000ms time allowance).

The configurations which had both a moderate board size and moderate time allowance led to the games being somewhat of a tossup between the two agents. Some configurations saw the `GAAgent` with the higher average score, while others gave it to the `ANNAgent`. In general, the more moderate the board size and time allowance, the closer the average game score tended to 0.

4.2 Final Analysis

The results of the experiments painted a shockingly clear picture. We found that the `ANNAgent`'s performance was rather consistent regardless of board size or time allowance. It's tree pruning technique provides a moderate boost in performance over a standard Monte-Carlo Tree Search, and does not noticably change with such changes in board configuration. This demonstrates the value of having certain nodes in the game tree be visited a higher number of times, even at the expense of a smaller number of overall MCTS iterations. Because of this, it can be used to reliably improve on MCTS without a need to worry about the structure of the game.

The `GAAgent` can also provide a demonstrable boost in performance over standard MCTS, but its algorithm's performance is much more dependant on the board size and time allowance it is given. Its performance has a strong positive correlation with time allowance, and a strong negative correlation with board size. Overall, this leads its performance to be much more variable. It can outperform the `ANNAgent` in some configurations, but in others it can struggle against even the trivial `RandomAgent`.

We can explain the `GAAgent`'s performance trend rather easily. Because of the random starting point of its population of heuristic weights, it requires a large number

of iterations through MCTS before its population can find a decent hueristic function. With a low time allowance or a large board size, it simply cannot find an appropriate heuristic function until rather late in the game. This does, however, speak to the potential power of this type of agent. Despite its notably bad performance early on in many games, it was often able to outperform its opponent and win by the end of the game.

The agent, though, is likely very dependant on starting with an appropriate hueristic. If the things the agent is measuring on the board as its hueristic do not represent a realistic strategy, it is very likely the agent would perform poorly — its performance may even be lower than a standard MCTS.

Chapter 5

Summary

5.1 Future Work

Most improvements to our work are needed in the games classes. Since each agent utilizes methods in the game classes for the MCTS process, any improvements to the algorithms within the games classes will immediately lead to a performance boost in each agent. The most needed improvement is on the `boardResolution` method of the Go class. The algorithm has a very high time-complexity, and is called in every iteration of MCTS; this combination leads to an agent's performance becoming exponentially worse as board size increases.

Beyond this, we would like to test agents with better trained neural networks. To train each neural network, we used only 2500 pieces of data; this is an admittedly small training set. These networks were able to provide the agent with a noticeable performance boost, but it would be interesting to see how much better performance would be with a more reliable network.

5.2 Conclusion

We have implemented a new general-purpose game playing framework for two-player turn-based board games. The framework supports both human and computer players, and is completely uncoupled from any game-specific functionality; new games can be designed and played on the framework with almost no change to the existing codebase. We also implemented the games Go and Hex for the framework. We then created one trivial and four intelligent general-purpose game playing agents. Similar to the entire framework, the agents contain no game-specific functionality and could be used with any games that run on our platform.

The trivial agent is called `RandomAgent`, and simply performs random moves on the board. The first intelligent agent is called `MCTSAgent`, and utilizes Monte-Carlo Tree Search as its decision-making algorithm. After these agents were implemented, we created the following three agents: `GAAgent`, `ANNAgent`, and `NEATAgent`; for this, we heavily employed the Encog machine learning framework. Each of these agents utilize their respective machine learning technique to alter the standard MCTS algorithm, following the proposals of several different previous researchers.

We used our framework and game implementations to perform novel research on these agents. While they're respective algorithms have been compared to a standard MCTS algorithm and were found to improve on its performance, these experiments were often done in different environments and under different conditions. Furthermore, agents utilizing these algorithms had never before been directly compared to one another — we provide this direct comparison. We ran a series of experiments between each agent on the games Go and Hex with a variable board size and MCTS time allowance. These parameters provide different branching factors for the games, and overall MCTS iteration restrictions, respectively. Varying these parameters provides the most applicable real-world results.

The data from our experiments uncovered some very clear performance trends. Particularly, the `ANNAgent` provided a moderate boost in performance that was independent of both board size and time allowance. This makes it a reliable choice with which to improve MCTS. The `GAAgent`'s performance was more variable. While it had the potential to provide a higher performance boost than the `ANNAgent` under certain circumstances, its performance was very weak in others. Specifically, we found that the `GAAgent`'s performance had a positive correlation with time allowance, and a negative correlation with board size.

Our research provides new data on an algorithm which is currently heavily used in the field of artificial intelligence. Beyond this, the framework we have implemented has the potential to make similar future research in this area much more straightforward.

Appendix A

Data Visualizations

This Appendix contains the full set of data visualizations. The graphs are presented in grids, with with each grid containing the playouts between two different agents; each grid contains all board sizes and time allowances for these agents. They are organized with board size increasing left to right, and the time allowance increasing from top to bottom. For example, 5x5 Go games with a 500ms time allowance are in the top-left corner of each grid, while 11x11 Go games with an 8000ms time allowance are in the bottom right.

For a demonstration of how to read a graph containing the score information for a board configuration of Go, consider FigureA.1. In addition to giving the board size and time allowance of the game, the graph is titled in order of player; that is, it is formatted as 'Player one' vs 'Player two'. This distinction is important to properly understand the score. The x-axis is how many moves have been made in a game, and the y-axis notates the score of the game at that point. Each colored line on the graph represents a single, distinct playout of the game. A positive score is good for Player one, and a negative score is good for Player 2. The grey smooth curve represents the average score at each move across all games of that type.

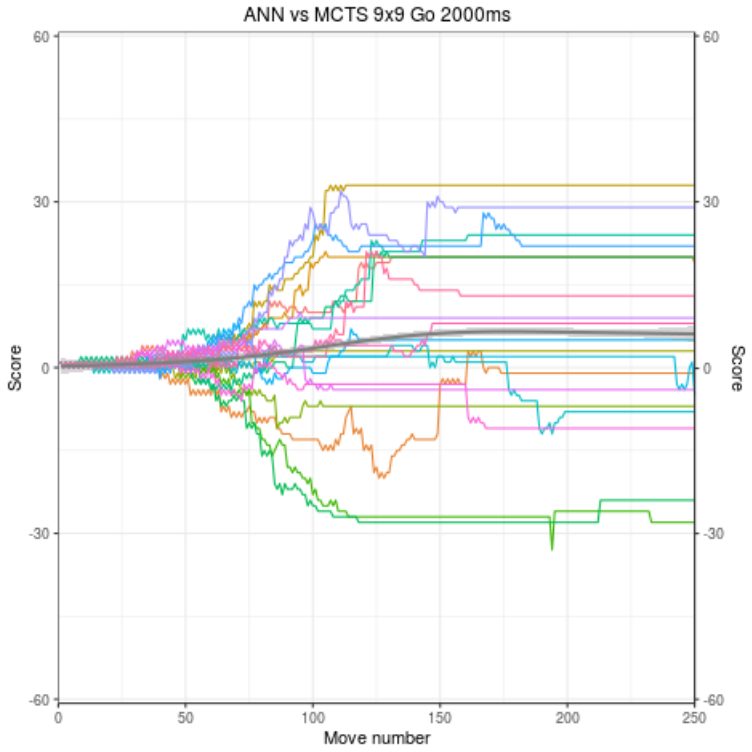


Figure A.1: Go Score Sample Graph

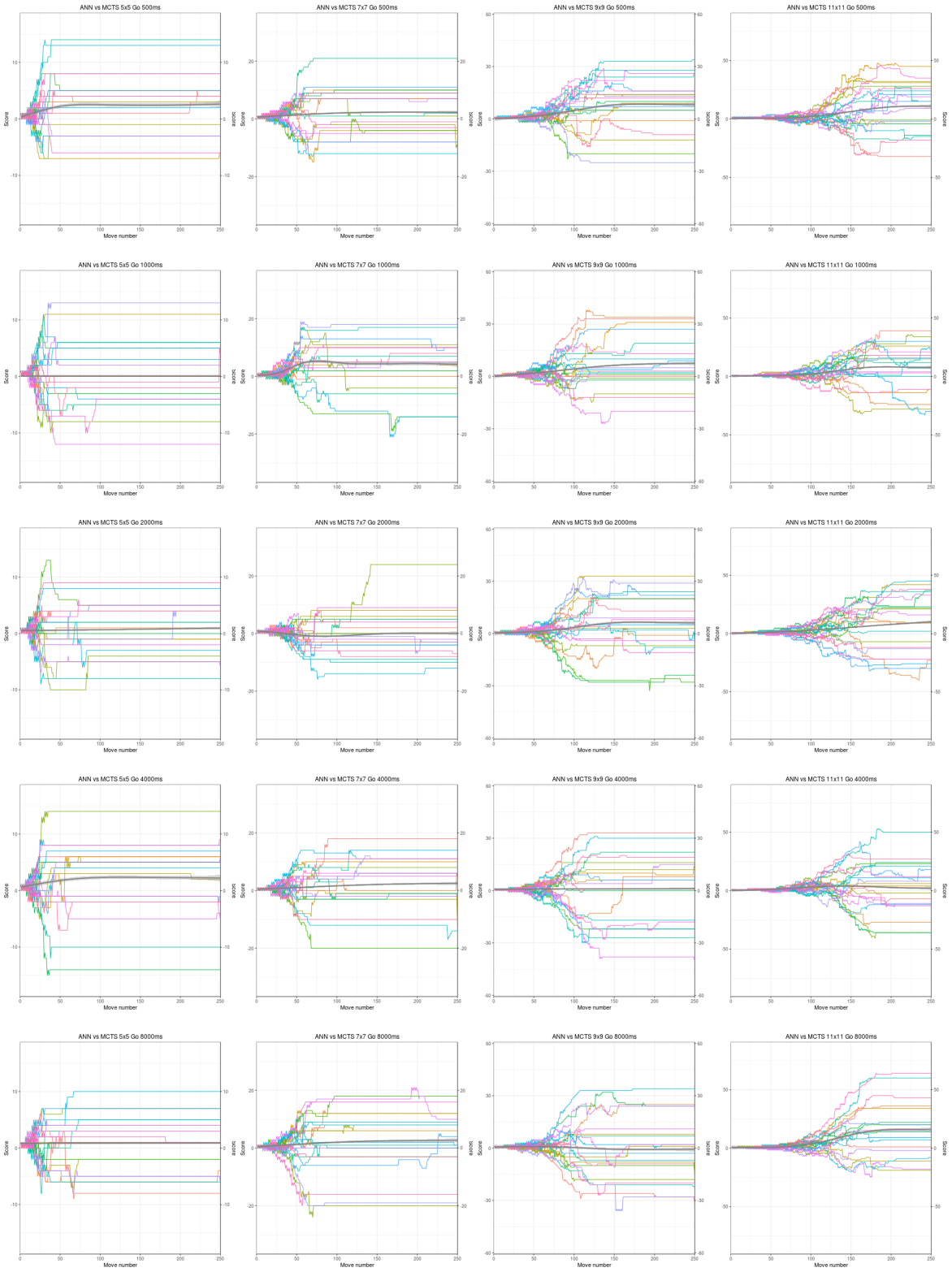


Figure A.2: `ANNAgent` vs `MCTSAgent` Go scores by board size and time allowance

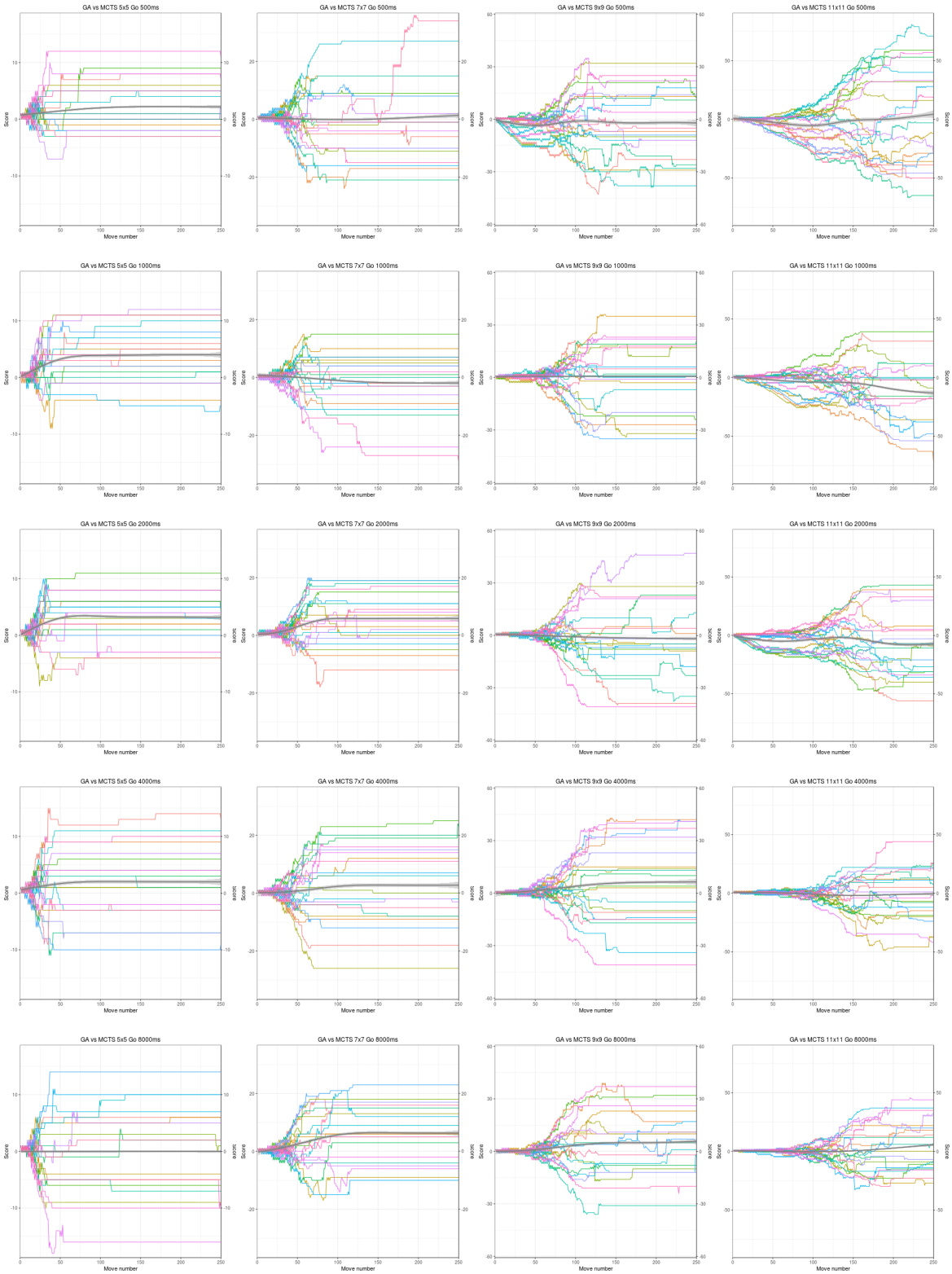


Figure A.3: `GAAgent` vs `MCTSAgent` Go scores by board size and time allowance

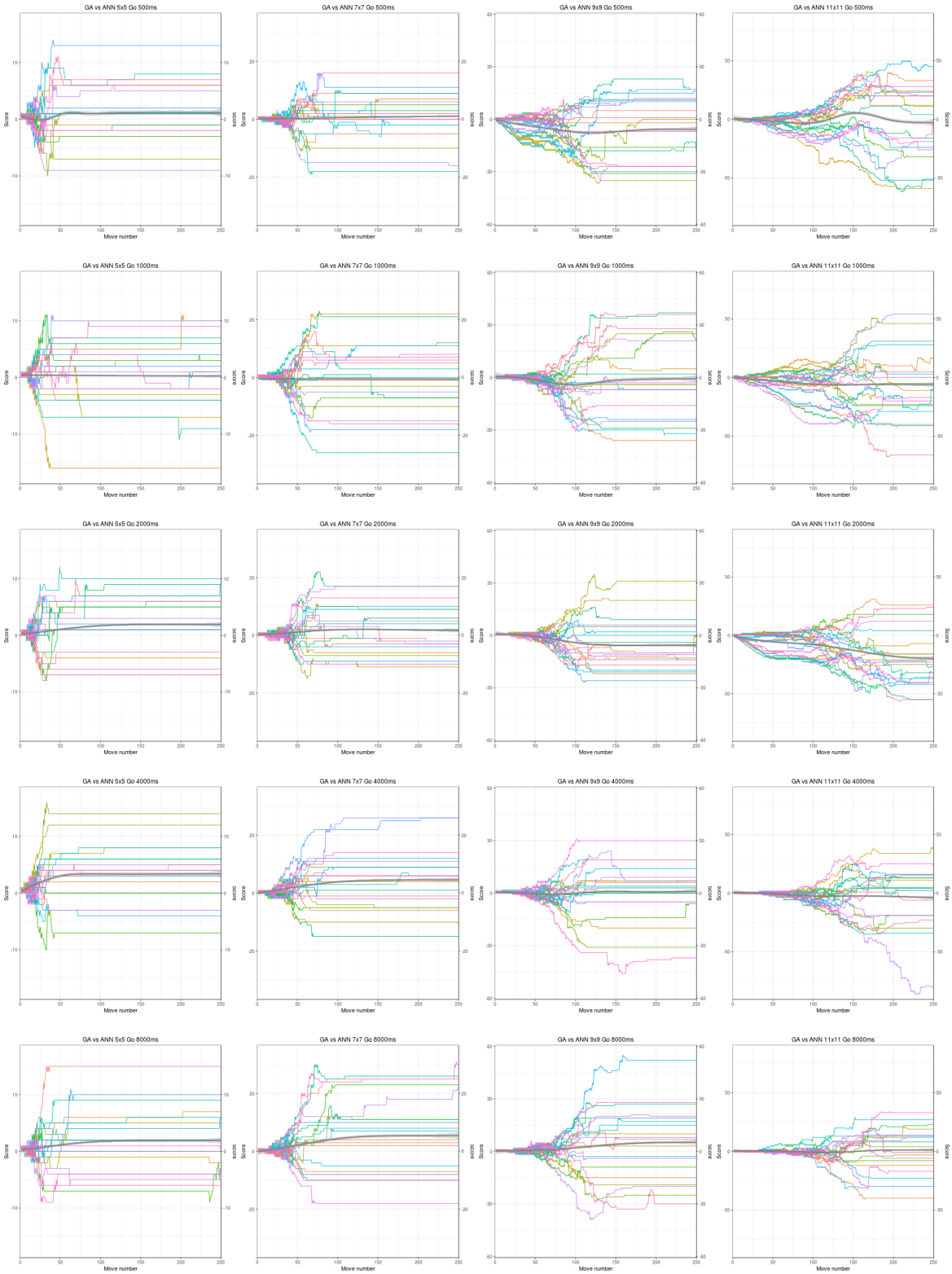


Figure A.4: `GAAgent` vs `ANNAgent` Go scores by board size and time allowance

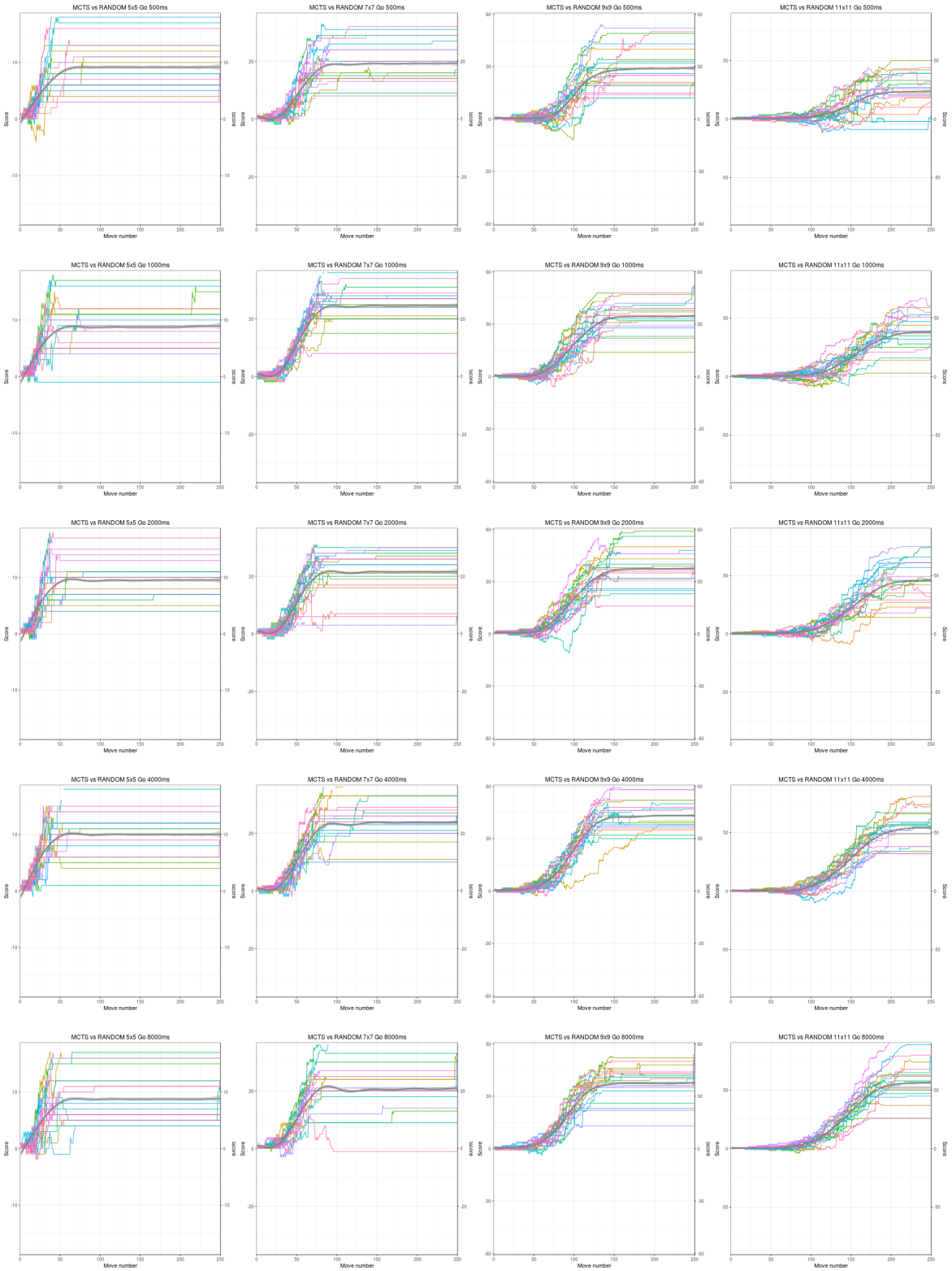


Figure A.5: `MCTSAgent` vs `RandomAgent` Go scores by board size and time allowance

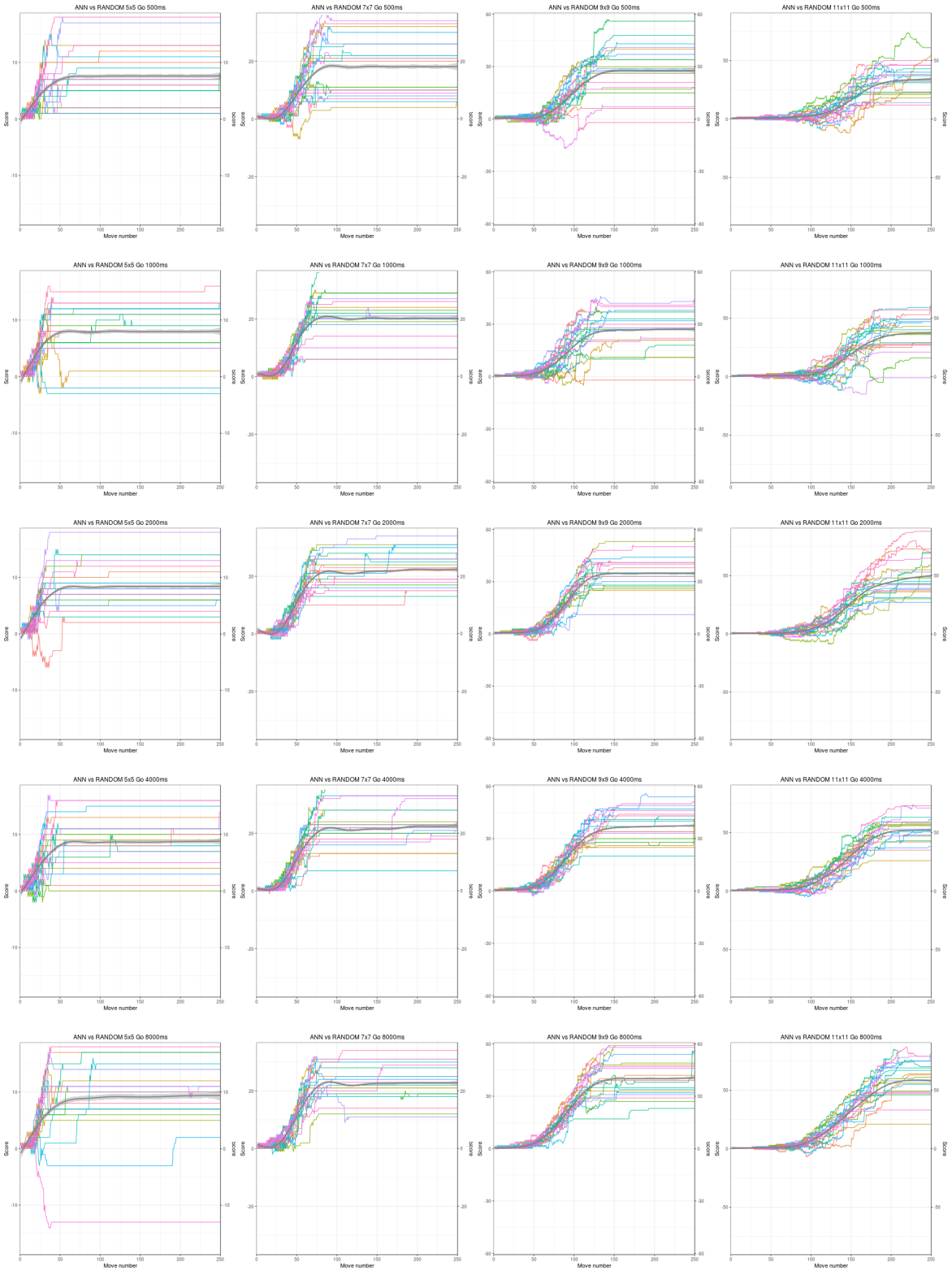


Figure A.6: `ANNAgent` vs `RandomAgent` Go scores by board size and time allowance

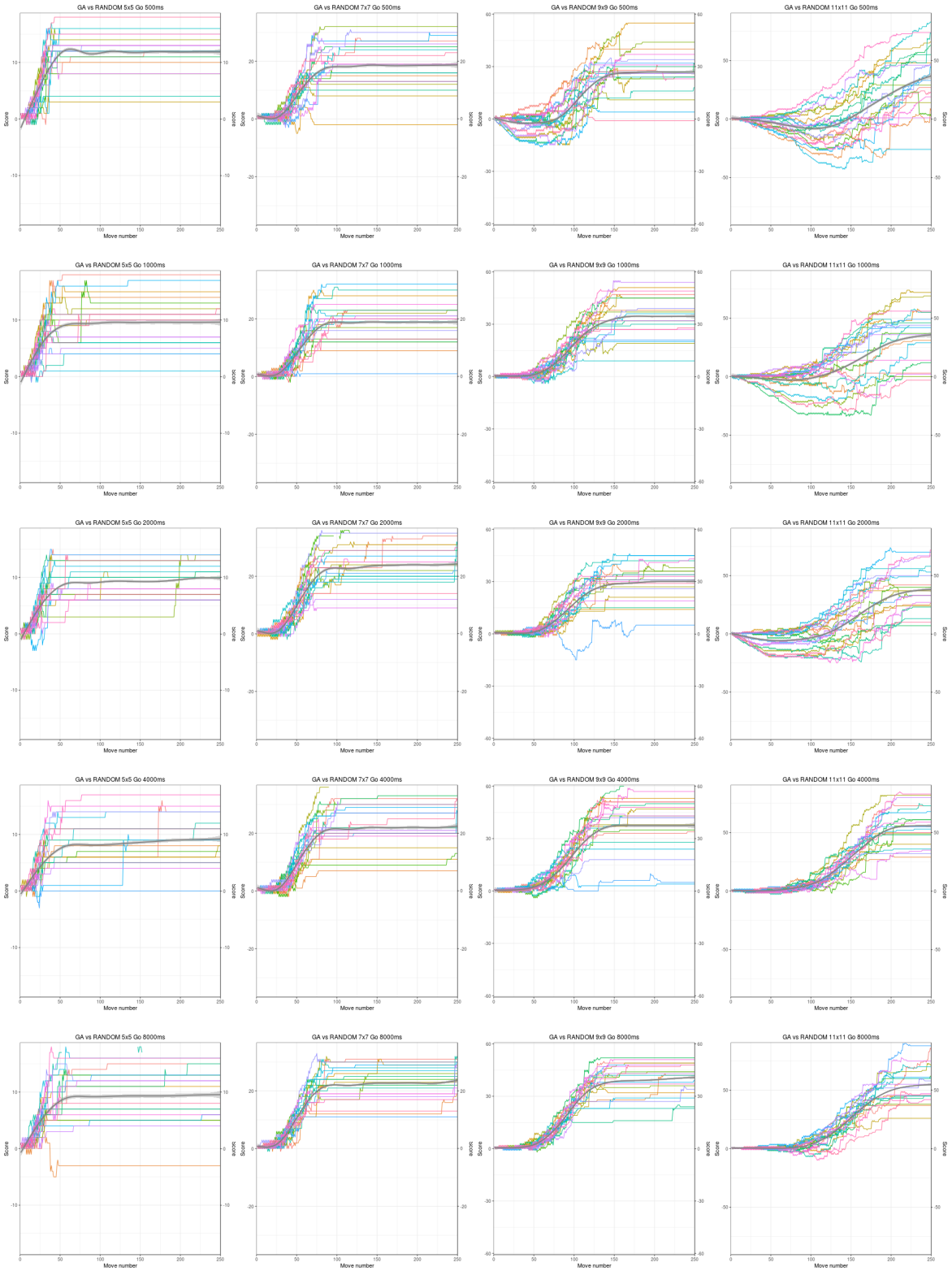


Figure A.7: `GAAgent` vs `RandomAgent` Go scores by board size and time allowance

Bibliography

[1] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E.; *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in neural information processing systems, IEEE, 2012.

[2] Hinton, Geoffrey; Deng, Li; Yu, Dong; Dahl, George; Mohamed, Abdel-rahman; Jaitly, Navdeep; Senior, Andrew; Vanhoucke, Vincent; Nguyen, Patrick; Sainath, Tara; Kingsbury, Brian; *Deep Neural Networks for Acoustic Modeling in Speech Recognition*, IEEE Signal Processing Magazine, Pearson Education, 2012.

[3] Parkhi, Omkar M.; Vedaldi, Andrea; Zisserman, Andrew; *Deep Face Recognition*, Robotics Research Group, 2015.

[4] Van den Oord, Aaron; Dieleman, Sander; Zen, Heiga; Simonyan, Karen; Vinyals, Oriol; Graves, Alex; Kalchbrenner, Nal; Senior, Andrew; Kavukcuoglu, Koray; *WaveNet: A Generative Model for Raw Audio*, Google DeepMind, 2016.

[5] RoboCup Soccer Tournament, `http://www.robocup2016.org/en/`.

[6] Gunderson, Louise F.; Gunderson, James P.; *Robots, Reasoning, and Reification*, Springer, 2008

[7] Bihlmaier, Andreas; Worn, Heinz; *Robot Unit Testing*, Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots, ACM, 2014

[8] `http://www.policonomics.com/perfect-imperfect-information/`, Policonomics, 2012

[9] Gilpin, Andrew; Sandholm, Tuomas; *Lossless abstraction of imperfect information games*, ACM, 2007

[10] Flying Machine Studios, *An Exhaustive Explanation of Minimax, a Staple AI Algorithm*, `http://www.flyingmachinestudios.com/programming/minimax/`, 2012

[11] Chaslot, Guillaume; Bakkes, Sander; Szita, Istvan; Spronck, Pieter; *Monte-Carlo Tree Search: A New Framework for Game AI*, AIIDE, 2008

[12] Tromp, John; Farenback, Gunnar; *Combinatorics of Go*, Lecture Notes in Computer Science, Vol. 4630, Springer, 2016

[13] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis, *Mastering the Game of Go with Deep Neural Networks and Tree Search*, Nature, 2016

[14] Henderson, Philip Thomas, *Playing and Solving the Game of Hex*, University of Alberta, 2010

[15] Hollosi, Arno; Pahle, Morten, *Sensei's Library: Scoring*, `http://senseis.xmp.net/?Scoring`, 2017

[16] Browne, Cameron B;; Powley, Edward; Whitehouse, Daniel; Lucas, Simon M; Cowling, Peter I; Rohlfshagen, Philipp; Tavener, Stephen; Perez, Diego; Samothrakis, Spyridon; Colton, Simon; *A survey of monte carlo tree search methods*, Computational Intelligence and AI in Games, IEEE, 2012

[17] Lucas, Simon M; Samothrakis, Spyridon; Perez, Diego; *Fast evolutionary adaptation for monte carlo tree search*, Applications of Evolutionary Computation, Springer, 2014

[18] Audibert, Jean-Yves; Munos, Rémi; Szepesvári, Csaba; *Exploration–exploitation tradeoff using variance estimates in multi-armed bandits*, Theoretical Computer Science, Elsevier, 2009

[19] Nakhost, Hootan and Müller, Martin, *Monte-Carlo Exploration for Deterministic Planning*, IJCAI, 2009

[20] Enzenberger, Markus; Müller, Martin; Arneson, Broderick; Segal, Richard; *Fuegoan open-source framework for board games and Go engine based on Monte Carlo tree search*, Computational Intelligence and AI in Games, IEEE, 2010

[21] Melanie, Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, 1999

[22] Silver, David, *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*, Elsevier, 2011

[23] Cazenave, Tristan, *Evolving Monte Carlo tree search algorithms*, L.I.A.S.D., 2007

[24] Russell, Stuart and Norvig, Peter *Artificial Intelligence: A Modern Approach*, 3rd ed., Prentice Hall, 2009

[25] Engelbrecht, A. P., *Computational Intelligence: An Introduction*, 2nd ed., Wiley Publishing, 2007

[26] Burger, Clayton; Plessis, Mathys C. du; Cilliers, Charmain B.; *Design and Parametric Considerations for Articial Neural Network Pruning in UCT Game Playing*, SAICSIT 2013, ACM, 2013

[27] Stanley, Kenneth O. and Miikkulainen, Risto, *Evolving Neural Networks through Augmenting Topologies*, Evolutionary Computation, MIT Press, 2002

[28] Stanley, Kenneth O.; D;Ambrosio, David; Gauci, Jason; *A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks*, Artificial Life Journal, MIT Press, 2009

[29] Gauci, Jason and Stanley, Kenneth O. *Indirect Encoding of Neural Networks for Scalable Go*, PPSN 2010, MIT Press, 2010

[30] Nielsen, Michael A., *Neural Networks and Deep Learning*, Determination Press, 2015

[31] ES-HyperNEAT, `http://eplex.cs.ucf.edu/ESHyperNEAT/`, 2012

[32] Clarifai visual recognition, `https://www.clarifai.com/technology`, 2016

[33] Lemoine, Julien and Viennot, Simon, *Computer analysis of Sprouts with nimbers*, Games of No Chance 4, MSRI, 2015

[34] Marcolino, L.S. and Matsubara, H., *Multi-Agent Monte Carlo Go*, Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems, 2011

[35] S. Takeuchi, T. Kaneko, and K. Yamaguchi, *Evaluation of Monte Carlo Tree Search and the Application to Go*, CIG 08, 2008

[36] Computer Go — Past Events, `http://www.computer-go.info/events/`, 2016

[37] Jean-Luc, W. *Wining situation on a Hex Board*, 2009

[38] Cornell Math Explorer's Club, `http://www.math.cornell.edu/mec/2003-2004/graphtheory/sprouts/sproutsgametree.html`, 2003

[39] *Length of a Go Game*, `http://homepages.cwi.nl/~aeb/go/misc/gostat.html`

[40] Baird, Leemon and Schweitzer, Dino, *Complexity of the Game of Sprouts*, 2010

[41] Torbert, Shane, *Applied Computer Science*, 2nd Ed., Springer, 2012

[42] Heaton, Jeff, *Encog: library of interchangeable machine learning models for Java and C#*, Journal of Machine Learning Research, Volume 16, 2015

[43] Beust, Cedric, *Jcommander command line parsing framework*, 2017

[44] Menden, Michael P and Iorio, Francesco and Garnett, Mathew and McDermott, Ultan and Benes, Cyril H and Ballester, Pedro J and Saez-Rodriguez, Julio, *Machine learning prediction of cancer cell sensitivity to drugs based on genomic and chemical properties*, PLoS one, 2013

[45] Awan, Shahid M and Aslam, Muhammad and Khan, Zubair A and Saeed, Hassan, *An efficient model based on artificial bee colony optimization algorithm with Neural Networks for electric load forecasting*, Neural Computing and Applications, Springer, 2014

[46] Ramos-Pollán, Raúl and Guevara-López, Miguel Ángel and Oliveira, Eugénio, *A software framework for building biomedical machine learning classifiers through grid computing resources*, Journal of medical systems, Springer, 2012