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
2016

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

Our research focuses on researching areas of research which research has shown to be lacking in research. Research shows that the research we complete will fill certain gaps in research, making future research easier to research.

Contents

List of Figures

Chapter 1

Introduction

Our research focuses on the development of intelligent agents for playing perfect information games. Specifically, we implement agents for playing the game Go, Hex, and Sprouts, which we will thoroughly describe later. The purpose of this chapter is to outline our motivation behind researching this problem, state the goals of our research, and introduce Monte-Carlo Tree Search (MCTS), an algorithm which is very often used for game-playing agents. We also explain the rules of Go, Hex, and Sprouts, as well as common structures found within the games.

1.1 Motivation

A major long-term goal of artificial intelligence 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 human intelligence. AI systems already excel in some areas. 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 perform tasks such as 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.

However, human intelligence is far more than recognizing patterns and understanding the things we sense. The ability to react to our world, solve problems, and adapt to changes is much more important when defining true intelligence. Therefore, in order to progress towards the goal of a more generalized AI, we need to build systems which exist in and adapt to changing environments.

One's first instinct in developing 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 obvious. A quick google search yields many results of the robots failing to do what they were

designed to.

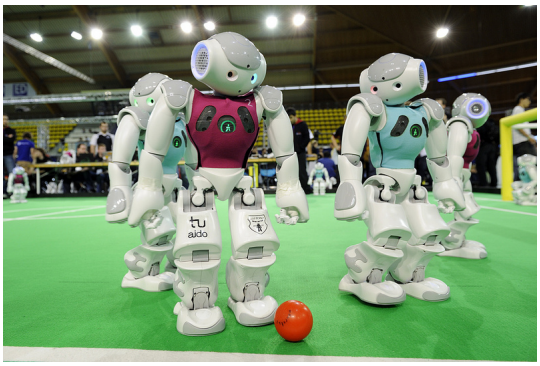


Figure 1.1: RoboCup 2015 [5]

This is an example of why robots are not ideal for developing AI algorithms — 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 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. Purely virtual environments — specifically computer board games — though, are perfect for experimentation for several reasons.

When working with 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, the 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 AI 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.

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 such imbalances may begin to even out over many trials), it can be easier to just begin with a perfect information game.

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 an evaluation function or heuristic. 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 with little success [10].

Tree search methods are another way of determining the value of a game state. A tree search method simulates a number of full games from the given state, attempting to fully construct a tree of all possible game configurations from the current state — this tree is called the game tree. 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. 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 [11]. Note this isn't actually constructing the game tree, just calculating how large it would be — it is actually theoretically impossible to even come close to constructing the whole game tree with today's computing capabilities, due to the sheer amount of storage needed to keep track of everything.

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. 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 [12][13].

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 [14][10].

MCTS is superior to other popular 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 [14]. 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 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 [17]? 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:

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

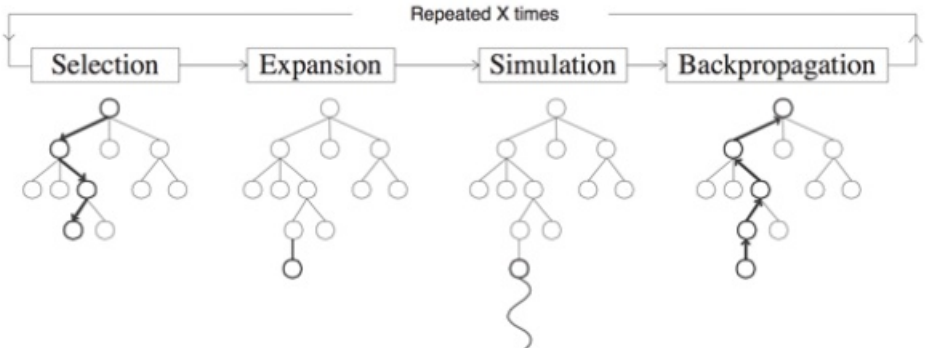


Figure 1.2: Monte-Carlo Tree Search algorithm [10]

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 [15][16].

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. These algorithms and existing implementations of game-playing agents which utilize these algorithms will be discussed in Chapter 2.

1.3 Games

In this brief section, we outline the perfect information games we use for our experiments: Go, Hex, and Sprouts. Go and Hex are two rather well-known and studied games, while Sprouts has been less popular in research. The purpose of this short section is to give the reader an understanding of how the games are played.

1.3.1 Go

Go is a two-player game usually played on a 19x19 board, although boards of size 9x9, 13x13, and 17x17 are also 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 or one player resigns.

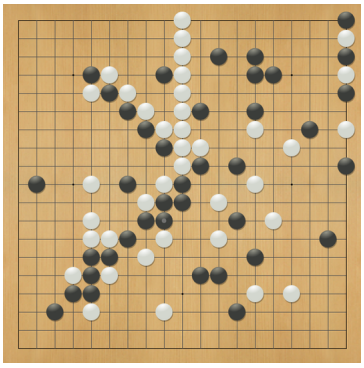


Figure 1.3: A Go board generated in GoGui

Despite its simple rules, Go has over 2×10^{170} possible playouts, making it one of the most computationally complex board games ever created [11]. 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.

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 1.4. 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.

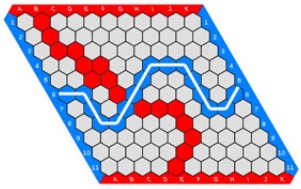


Figure 1.4: A winning game for blue [34]

1.3.3 Sprouts

Sprouts is a two person pencil-and-paper game which was created by John Conway and Micheal Paterson at Cambridge University in the 1960s. The game has two players, and begins with any number of dots on a piece of paper. The two players take turns drawing a line either between two dots or from one dot to itself. When this line is drawn, a new dot is placed on the line splitting it in half. The last player who is able to legally draw a line wins. A line is only legal if it does not cross any

other lines. Furthermore, no vertex can have more than 3 lines coming out of it — a play on a vertex which already has 3 lines is illegal. The game ends when no moves are possible, and the last player to make a legal move wins.

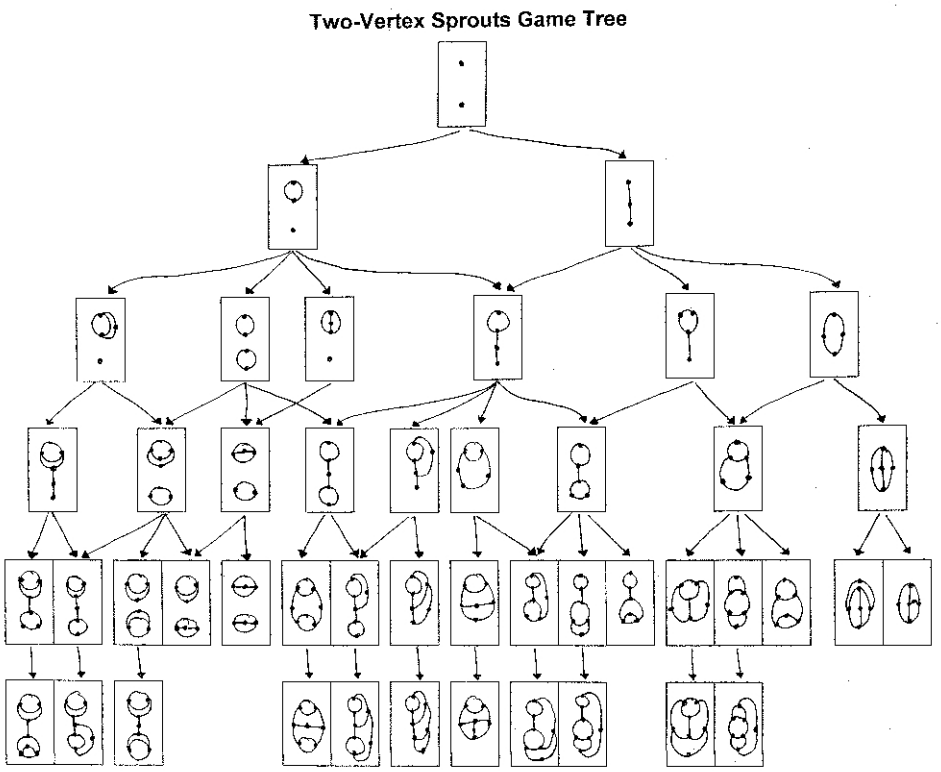


Figure 1.5: 2 vertex game tree for Sprouts [35]

Figure 1.5 shows the game tree for the simplest possible game of sprouts — one with only two starting vertices. As new vertices are added to the start of the game, the tree grows very rapidly. Because of the simplicity of games with a small number of starting nodes, Sprouts has been completely solved for up to 32 starting vertices — that is, if the number of starting nodes is at most 32, one can play a perfect game and force a win depending on whether they move first or second [30].

1.4 Goals of the Project

As stated previously, most of the best current game-playing agents utilize MCTS with some integration of another AI 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 several perfect information games: Sprouts, 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. Using multiple games helps show how this introduction of a heuristic bias affects how game-dependent the performance of the agent is.

1.5 Thesis Outline

Chapter two introduces the different AI algorithms we integrate with MCTS, and also describes current 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. Chapter three goes over our experiment design, testing methods, and any possible validity concerns are also discussed. In Chapter four, we overview Fuego, the library we use to implement our agents, as well as the actual design of each agent. Finally, we discuss our results and future research in Chapter five.

Chapter 2

Related Work

In this chapter, we discuss the different algorithms we integrate with MCTS, as well as describe existing implementations of game-playing agents which use these techniques. 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.

2.1 Genetic Algorithms

A Genetic Algorithm (GA) is an optimization or search algorithm based on Darwin's theory of evolution [19]. 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 $f(x)$; a chromosome's fitness may also just be an amount relative to the 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.

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.

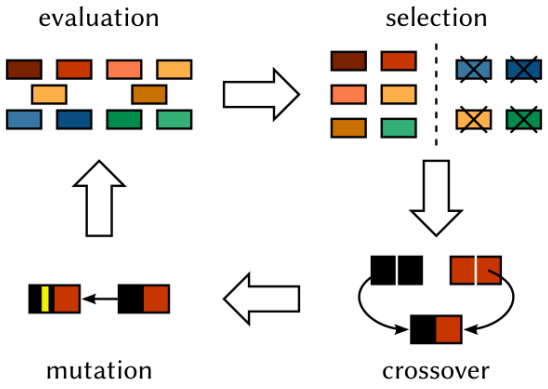


Figure 2.1: Process of a Genetic Algorithm

A GA can be integrated with MCTS to fine-tune both the tree policy and the default policy. In [20], Cazenave created a Go-playing agent which utilized MCTS whose UCT formula 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, and finally one with UCT whose bias parameter he optimized with a GA. The agent with the optimized bias parameter outperformed the other two.

More interesting is Lucas *et al.* 's fast evolution method [15]. 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 [21]. 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 [22]. A simple illustration of a feed-forward ANN with one hidden layer can be seen in Figure 2.2.

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

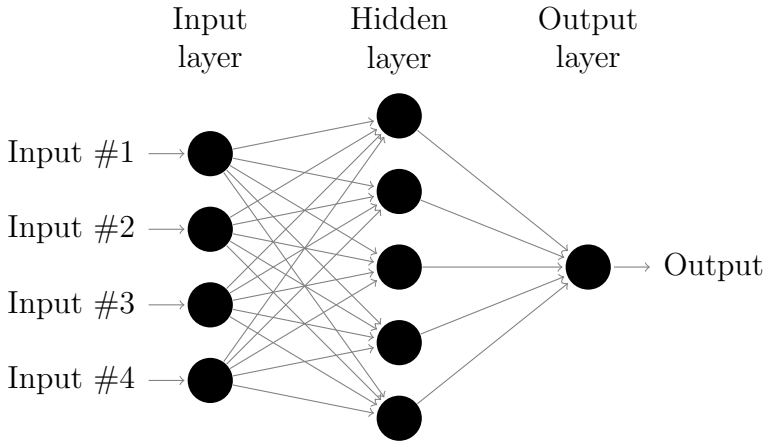


Figure 2.2: A simple feed forward ANN

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 [21].

In [23], 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 [21]. 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 [24].

In [24], 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 through evolutionary

methods. The exact methodology behind this technique is described in [24].

Hypercube-based NEAT (HyperNEAT) is an extension of NEAT. HyperNEAT trains and evolves ANNs to be able to understand the geometry of a problem through what are called *Compositional Pattern Producing Networks*, which are a type of neural network. Essentially, HyperNEAT uses the same process as NEAT, but it specifically uses CPPNs. [25]

Gauci and Stanley demonstrate how both NEAT and HyperNEAT can be used for Go in [26]. 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 HyperNEAT. Specifically, they state HyperNEAT could be used to evolve a more effective default policy for UCT. This is precisely what we do in our implementation.

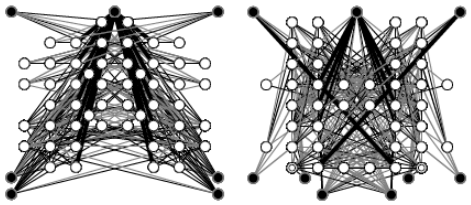


Figure 2.3: A CPPN at gen 30 (left) and gen 106 (right) evolved by HyperNEAT [28]

2.4 Deep Convolutional Networks

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 [12], 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.

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

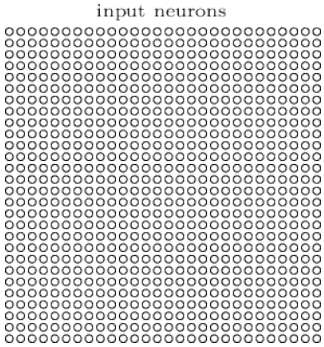
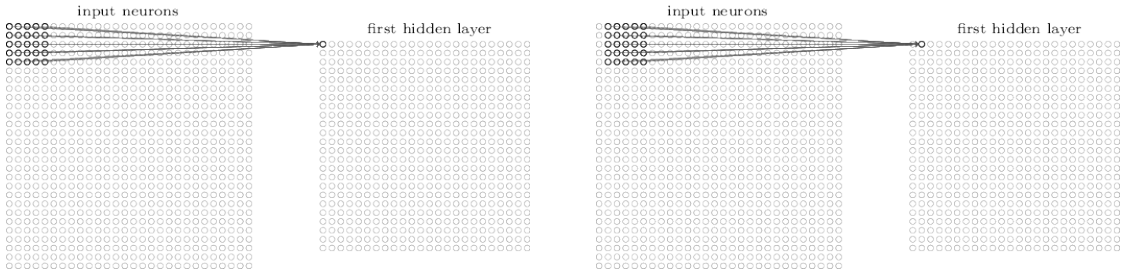


Figure 2.4: DCN input layer [27]

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[27]

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.

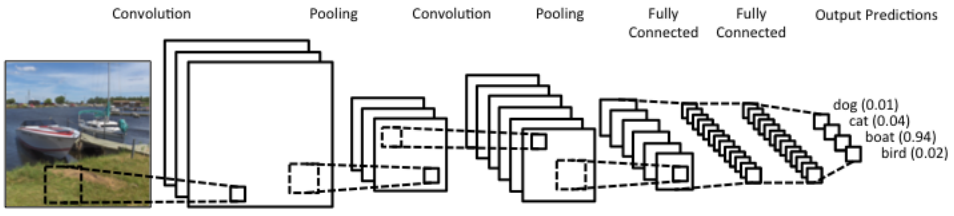


Figure 2.6: An example of a full DCN [29]

Perhaps the most famous implementation of a DCN with MCTS is Google Deep-

Mind's Alphago [12]. 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.[12]

Chapter 3

Method of Approach

This chapter demonstrates how to include short code segments, how to include pseudocode, and a few other LaTeX features.

3.1 Test Environment

Algorithm 3.1 (from [**?**]) shows a high-level description of an algorithm. There are many options for the display of pseudocode; this uses the `algorithm2e` package [**?**], but there are a number of others available at the Comprehensive TeX Archive Network (`ctan.org`). Using any of these other packages might require the additon of one or more "`\usepackage{...}`" commands in the main `AllegThesis.tex` file.

Data: this text
Result: how to write algorithm with LaTeX2e
initialization;
while *not at end of this document* **do**
  read current;
  if *understand* **then**
    go to next section;
    current section becomes this one;
  else
    go back to the beginning of current section;
  end
end

Figure 3.1: How to write algorithms (from [**?**])

3.2 Experiments

Figure 3.2 shows a Java program. There are many, many options for providing program listings; only a few of the basic ones are shown in the figure. Some thought

must be given to making code suitable for display in a paper. In particular long lines, tabbed indents, and several other practices should be avoided. Figure 3.2 makes use of the `listings` style file [**?**].

```
public class SampleProg
{
   private int dummyVar; // an instance variable

   public static void main(String[] args)
   {
      System.out.println("hello world.");
      // Avoid long lines in your program; split them up:
      dummyVar = Integer.parseInt("1234")
           + 17 * ("abcde".substring(1,3).length()
           + 11 - 1000;
      // Use small indents; don't use the tab key:
      for (int i = 0; i < 10; i++)
      {
         for (int j = 0; j < 10; j++)
         {
            if (i > j)
            {
               System.out.println(i);
            }
         }
      }
   }
}
```

Figure 3.2: `SampleProg`: A very simple program

3.3 Threats to Validity

Chapter 4

Implementation

Another possible chapter title: Experimental Results

Chapter 5

Discussion and Future Work

This chapter usually contains the following items, although not necessarily in this order or sectioned this way in particular.

5.1 Summary of Results

A discussion of the significance of the results and a review of claims and contributions.

5.2 Future Work

5.3 Conclusion

Appendix A

Java Code

All program code should be fully commented. Authorship of all parts of the code should be clearly specified.

```
/**
 * SampleProg -- a class demonstrating something
 * having to do with this senior thesis.
 *
 * @author   A. Student
 * @version  3 (10 December 2013)
 *
 * Some portions of code adapted from Alan Turing's Tetris program;
 * relevant portions are commented.
 *
 * Revision history:
 *     10 December 2013 -- added ultra-widget functionality
 *     18 November 2013 -- added code to import image files
 */
public class SampleProg
{
   private int dummyVar; // an instance variable

   public static void main(String[] args)
   {
      //=================================================
      // This section of code adapted from Alan Turing's
      // Tetris code. Used with permission.
      // http://turinggames.com
      //=================================================
      System.out.println("hello world.");

      dummyVar = Integer.parseInt("1234")
           + 17 * ("abcde".substring(1,3).length()
           + 11 - 1000;

      for (int i = 0; i < 10; i++)
      {
         for (int j = 0; j < 10; j++)
         {
            if (i > j)
```

```
            {
               System.out.println(i);
            }
         }
      }
   }

   /**
    * foo -- returns the square root of x, iterated n times
    *
    * @param    x   the value to be interatively processed
    * @param    n   number of times to iterate square root
    * @return   sqrt(sqrt(...())) [n times]
    */
   public double foo(double x, int n)
   {
       for (int i = 0; i < n; i++)
           x = Math.sqrt(x);
       return x;
   }
}
```

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

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

[12] 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

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

[14] 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

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

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

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

[18] 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

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

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

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

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

[23] 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

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

[25] 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

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

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

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

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

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

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

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

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

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

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

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

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