University of Nebraska at Omaha
DigitalCommons@UNO

Student Work

5-2011

WOPR in search of better strategies for board games

Ben Horner
University of Nebraska at Omaha

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Part of the Computer Sciences Commons

Recommended Citation
Horner, Ben, "WOPR in search of better strategies for board games" (2011). *Student Work*. 2864.
https://digitalcommons.unomaha.edu/studentwork/2864

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

WOPR
in search of better strategies for board games

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment
of the Requirements for the Degree

Masters of Science

University of Nebraska at Omaha

by

Ben Horner

May, 2011

Supervisory Commitee:
Dr. Victor Winter (chair)
Dr. William Mahoney
Dr. Hesham Ali

UMI Number: 1490758

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 1490758
Copyright 2011 by ProQuest LLC.
All rights reserved. This edition of the work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

WOPR

in search of better strategies for board games

Ben Horner, M.S.

University of Nebraska, 2011

Advisor: Dr. Victor Winter

Board games have been challenging intellects and capturing imaginations for more than five thousand years. Researchers have been producing artificially intelligent players for more than fifty. Common artificial intelligence techniques applied to board games use alpha-beta pruning tree search techniques. This paper supplies a framework that accommodates many of the diverse aspects of board games, as well as exploring several alternatives for searching out ever better strategies. Techniques examined include optimizing artificial neural networks using genetic algorithms and backpropagation.

Contents

1 Introduction **1**

2 Board games **4**
- 2.1 What do board games have in common? 4
- 2.2 What do board games *not* have in common? 7
  - 2.2.1 Hidden information 7
  - 2.2.2 Chance 8
  - 2.2.3 Innumerable choices 9
  - 2.2.4 Simultaneous turns 9
  - 2.2.5 Cooperative play 10
- 2.3 The board games framework 10
- 2.4 The code 16
  - 2.4.1 The basics of Scala 16
  - 2.4.2 Implementing the board games framework 21
  - 2.4.3 Some simplifications 31

3 Strategy **33**
- 3.1 Game theory 33
- 3.2 Current methods 38
- 3.3 A different approach 41
- 3.4 The simple neural network implementation 42
- 3.5 The search space 47
- 3.6 How the search might proceed 51

4 The Search **55**
- 4.1 Genetic algorithms for search 55
- 4.2 The genetic algorithms framework 59
- 4.3 Tic-Tac-Toe as a test case 62
- 4.4 Real valued genetic algorithms 70
- 4.5 Backpropagation in neural networks 74

5 Next steps **80**

List of Figures

3.1 This diagram depicts collective choices being made from the root node, an artificial node where the decision between Heads and Tails is made by the pseudo-player chance, and a node that has an infinite number of choices. 35

3.2 The solid lines depict the subgraph defined by a strategy for the circle player, each circle has only one solid outgoing edge. The nodes connected by dashed lines cannot be reached if the strategy is used. Since the strategy only specifies the decisions of the circle player, all choices for the square player must be considered. 37

3.3 This diagram shows the nodes and weights of a simple network whose output at b is just a function of it's inputs at i1 and i2. 45

3.4 This diagram shows a feed-forward network, laid out first circularly, then in layers. Any missing edge can be thought of as having a weight of zero (i.e. BC or BD). 46

3.5 This diagram shows the tree structure of a subgraph strategy for Tic-Tac-Toe for the first player to a depth of five, and for the second player to a depth of four. The first player has included a single initial move, then all eight second player responses, then one first player response per second player response, etc. The second player strategy has included all nine initial first player moves, then one second player response for each, then all seven first player responses, etc. 48

4.1 This diagram shows a Tic-Tac-Toe state being mapped to a numerical value. First the sensor breaks down the state into a binary description, then this binary string is fed into a complete feed forward neural network (of two nodes), which outputs a number. The number is interpreted by the system as the player's evaluation of the state. The sensor and neural network are wrapped in a SimpleMaxPlayer which, when given a set of states, returns the one that evaluates to the highest value. 64

4.2 Allowing our weights to range from -1.0 to 1.0 with a precision of 1 decimal place requires 21 distinct values. A 4-bit Gray code supplies only 16, so a 5-bit code, which supplies 32, must be used. The binary string is sliced into substrings of length 5, then the Gray code is applied to obtain the real valued string. The weights on the edges in our neural network are then taken directly, in sequence, from the real valued string. The lower node in the pictured network has 20 inputs, and the higher has 21, so a binary chromosome that could fully define the weights of this network would need to be 5 * (20 + 21) = 205 binary digits long. 66

4.3 A plot of the data collected to analyze samples of random binary chromosome players with different numbers of neurons. The horizontal axis is the number of neurons, the vertical axis is the number of losses. For each value for the number of neurons, a sample of 500 random chromosomes was taken. It can be seen that none of the statistics varied by much, the bottom row of points represents the best of each sample, the top represents the worst, the squares in the center the median, and the x's in the center the mean. 67

4.4 A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 200, and population size remained at 5. . . 68

4.5 A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 10. . 68

4.6 A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 40. . 69

4.7 A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 2 to 40, as population increased from 5 to 100, and the number of generations was always 200. The missing quadrant was to computationally intensive to compute. 70

4.8 A plot of the data collected to analyze samples of random real chromosome players with different numbers of neurons. The horizontal axis is the number of neurons, the vertical axis is the number of losses. For each value for the number of neurons, a sample of 500 random chromosomes was taken. It can be seen that none of the statistics varied by much, the bottom row of points represents the best of each sample, the top represents the worst, the squares in the center the median, and the x's in the center the mean. 71

4.9 A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 200, and population size remained at 5. 72

4.10 A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 10. 72

4.11 A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 40. 73

4.12 A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 2 to 40, as population increased from 5 to 100, and the number of generations was always 200. The missing quadrant was to computationally intensive to compute. . 74

4.13 This is a plot showing the error reduction by backpropagation for sample networks with from 5 to 40 nodes, with a learning rate of 0.005, through 30,000 training cycles. 78

4.14 This is a plot showing the number of losses for sample networks with from 5 to 40 nodes as the error was reduced by backpropagation through 30,000 training cycles. 79

Chapter 1

Introduction

The famous Turing test proposes that if a human judge, in a textual question and answer session with two subjects, one human and one machine, cannot distinguish the human from the machine, then the machine can be said to be intelligent. There are many questions that can be asked that muddy the waters, about how intelligent the human subject is, and how intelligent the human judge is. The salient concepts, however, are testability, and the use of humans as the standard for comparison. The Turing test is unfortunately subjective, but it acknowledges that a test is needed. Without a test, attempting to determine whether or not a machine is intelligent cannot commence. Once it is acknowledged that a test is needed, and the task to find one is undertaken, the difficulty becomes apparent. An endless list of specific criteria can be drafted and then erased, never giving satisfaction. An exact boundary between what is intelligent and what is not cannot be found. It seems the word, as with many words, cannot be precisely defined. So instead of measuring intelligence, the test measures the machines ability to fool a human into thinking that the machine is also a human. Humans are intelligent by definition, we invented the word. Even if we can't agree on a precise definition, we all sort of "get it", and this capacity for imprecision may be one aspect of intelligence.

This comparison of the abilities of machines to the abilities of humans is a running theme in the field of artificial intelligence, specifically in head to head competitions. One highlight of this was the chess match between Deep Blue and Gary Kasparov in which the machine defeated the then world champion. Could Deep Blue be said to possess at least some measure of intelligence? Current Chess playing computers are far better than Deep Blue was then, humans no longer can compete with them. A recent noteworthy event was Watson's performance on the game show Jeopardy, easily defeating Brad Rutter, the winner of the most money in the history of the show, and Ken Jennings, the contestant with the longest winning streak. Watson was asked questions out loud as were the human contestants. Could Watson be said to posses some level of intelligence? These are examples of machines dominating humans at specific tasks. Deep Blue, nor any Chess computer today would have a chance of passing the Turing test. Watson is a step closer with it's natural language processing abilities, but it is very doubtful that it could pass either. Given a specific enough competition, and powerful enough hardware, there is little doubt that a machine would defeat the best humans, but passing the Turing test is still a ways off.

To pass the Turing test, to have a conversation indistinguishable from a human, a machine would need to be able to converse about the weather, tell you what it had for dinner last night, and learn a new game. To pass the Turing test, a more general intelligence is required. For example, a machine that could play competitively at any board game would be more generally intelligent than any machine that could only play a single board game, wouldn't it? It would still only address a fairly specific domain, that of board games, but much more general than a single game. If you had a machine with general enough intelligence to defeat all humans at all board games, you would have a fairly amazing tool on your hands. There are many real world problems that can be formulated as games, and a machine player capable of defeating all humans at all games would by definition be able to solve the real world problem

better than any human. This work is in pursuit of that very goal.

First we perform an analysis of strategy board games in general as opposed to party games, trivia games, or dexterity games, though games that don't necessarily involve a board, such as card games are included, to draw out their similarities and differences. In order for the system to be general, so that as little programming as possible would be required to produce a player for a new game, it was necessary to isolate the general nature of a game. Once this level of abstraction was captured, general placeholders for specialized components could be created, and any algorithms that were part of the general nature could be implemented in terms of the placeholders. Implementations for specific games would be specialized components that could be plugged in to the system.

Next we consider strategy in general, game theoretic concepts such as pure and mixed strategies are discussed. We discuss current methods for implementing strategies including heuristic based alpha-beta tree searching methods, and we see how a strategy can be implemented in terms of a function that evaluates the state of a game. Then we explore using a complete feedforward neural network as the function that evaluates a game state and implements a strategy, and the structure of that strategy as an indicator of how difficult it might be to find a good one.

Finally we describe the optimization of the neural network using genetic algorithms. We discuss how strong traits of parents are inherited by offspring and how this can lead to good strategies being extended forward from the opening of the game, and backward from the end of the game. Both binary and real valued genetic algorithms are used, and also the backpropagation algorithm within the neural network. We show that even using these powerful artificial intelligence techniques that this is a difficult problem to tackle.

Chapter 2

Board games

2.1 What do board games have in common?

So what is a board game? A large set of disparate examples was considered to determine what they all had in common. A fairly diverse subset is Chess, Stratego, Yahtzee, Texas hold'em, and Bridge, but the set actually considered is much larger. After much reflection a set of commonalities surfaced:

- Games all have rules.
- Games all have players.
- The goal of the players is to win the game.
- The players make decisions.
- Games all have state, which provides context for the rules.
- Games all progress in turns (of some kind).
- Games all always end (else how can they be won?).

While the rules of different games may not have much to do with each other, the fact that they all have rules is obviously vital. The rules are the framework that the players play within. While a game can exist sitting on a shelf, we are concerned with games in action, being played. If a game is being played then players are just as obviously vital as the rules. The goal of all players is to win the game and they act

in their own self interest, rationally. There may be value in producing less skillful opponents, in order to give a human the satisfaction of victory, or to provide a human with a training target, but the goal of this system is to produce the best automated player possible.

The measure of the quality of a player may be in terms of it's win/loss record against other players, or in terms of a rating which attempts to predict the winner of a game between a set of rated competitors. Examples of rating systems would be the Elo and Glicko rating systems used in Chess[3]. The source of the quality of a player though, is the quality of the decisions made at each turn, improving these decisions is the only way to improve a record or a rating. If a player's decisions always lead to the best possible outcome, that player can be considered to be perfect, such a player is always the ultimate goal.

Games all have pieces, or cards, or some form of physical components, but there are no components common to all of them. The disposition of the components provides a context within which the rules can be used to determine what actions can be taken. In addition to the positions of the physical components, there may be events in the history of a game that affect what actions are available. In games between two players in which turns alternate, one important historical event that affects available actions is which player moved last. Who moved last may not be apparent from the state of the components, but it must be tracked in order to use the rules to determine what actions are available to the players (one player has none). The present disposition of physical components of a game can be thought of as a concise representation of the moves made throughout the history of the game up to the present. The state of a game that is required to utilize the rules includes the physical state of the game (which is essentially historical data), as well as any other important historical events not represented by the physical components.

The type of game that we're concerned with has turns. The cycle of the game

proceeds in discrete steps, turn after turn. The alternative would be that a player could take an action at any time, an event driven kind of game, like a multiplayer video game would be. The choice was made to exclude this possibility, our interest is in strategic decision making, not in action. If such a game were found to have strategic merit, it would be possible to simulate an event driven nature by asking all players for an action at each turn, and allowing players to choose an action, or to do nothing. A "turn" would become just the time step in an event driven simulation.

Another aspect of the type of game that we're concerned with is that they end. This requirement excludes some things that may normally be thought of as games, such as the so called "cash game" in Texas hold'em. In a cash game, a player may buy in to a table and play for a while, and then, after several hands, leave the table (either having won or lost some money) while other players remain. Also, if a player loses all the money they sat down to play with, they may still put up more money to keep playing, they haven't "lost" the game, just lost some money. A game such as this doesn't really end, and for that reason, we are not considering it for our system. Our system is meant to produce players that win games, but a game that doesn't end can't be won, so our system applies only to games that end. There is the possibility of a player sitting down to a cash game and imposing extra rules on themselves so that the game will end for them, such as placing an artificial limit on the amount of time at the table, or the number of hands that will be played. The player could then try to win more money than any other player within this additional constraint.[1]

[1] Note that an optimal strategy for playing under an artificial constraint may be suboptimal without the constraint, leaving the player at a disadvantage to the players who are not imposing it on themselves.

2.2 What do board games *not* have in common?

A side effect of considering what games have in common with each other was the realization that there are several general concepts, that not all games exhibit, but which a general solution would have to provide for:

- Hidden information
- Chance
- Innumerable choices (either infinite, or impractically many)
- Simultaneous turns (multiple players moving at once)
- Cooperative play

2.2.1 Hidden information

Hidden information doesn't show up in all games, but when implementing an abstract, most general tier of a games framework it must be considered, else it may be impossible to specialize that framework for a game that includes hidden information. In Chess, all players know the entire state of the game at all times. The state of a game of Chess is almost entirely represented by the positions of the pieces on the board. The only ancillary bits of data needed are for special moves. In order for a player to make the special move called "castling", a player may not have previously moved their king, or the rook involved in the castling move. There is also a move called an "en passant" capture which is only available under the circumstances of a specific previous opponent's move, so it must be known if the opponents previous move made an en passant capture available. Though information corresponding to these two special cases is not evident from the information visible on the board, it is known to the players who saw the game unfold (similar to knowing which player has the next move). Hidden information on the other hand, is something like the contents of an opponents hand in a card game, the information is a matter of fact, but not known to all players. The system needs to keep track of the true state of a

game, but also what information is available to each player, which may be a subset.

2.2.2 Chance

In games without chance, that is, games in which a decision made by a player leads to a deterministic change in the state of the game, a player is basically choosing the next state for the game. Chess is such a game, and in Chess choosing a piece to move and what space to move it to, is equivalent to setting up a different board for each possible move, and allowing the player to walk among them to decide which they would like to continue the game on. In games with chance the situation can be different. There is a difference between a players choices being limited by a chance event, and a player deciding on an action, the outcome of which is based on chance. The key is in whether the chance event takes place before or after the player makes their decision. In Backgammon a player has no choice, they must roll dice at the beginning of their turn, but once the result of the roll is known they can make a deterministic choice about the next state of the game, in this case the players choices are limited by chance. In Yahtzee, after a player's initial roll, they have to decide which dice to set aside, and which dice to re-roll. In this case the player is presented with the results of the initial roll of the dice, they are not able to choose the result of their roll, their choices are limited by chance. Then the player must make their decision, which dice, if any, to re-roll, but the result of their decision depends on a random event, the re-roll of selected dice. If the player decides to re-roll one or more dice, they are not making a deterministic decision about what the values of those dice will be, they are leaving that to chance, and must deal with the consequences whatever they may be. There is a dichotomy between the chance event occurring before and after the players decision. When the chance event occurs post decision, the player's decision is not deterministic, they cannot choose the next state of the game, only an action to take.

2.2.3 Innumerable choices

In order for a player to make a decision, a set of choices has to be generated. For many games such as Chess it is possible to generate all possible choices, and make a decision from among them. For some games however, this is either impractical due to a combinatorial explosion, or in the case of a game with an infinite number of moves, impossible. An example of a game with an infinite number of moves is Calculus which requires the placement of a piece on a board that is not divided into spaces, the piece may be placed anywhere. The natural representation of this placement would seem to be two dimensional real valued coordinates, which presents a problem, there is a continuum of choices rather than a discrete set. One way of handling this is to consider that in order for a digital computer to play this game, some discretization of values is required, and also that at some point further precision is not valuable. Given these realizations, we can use a method such as splitting the board into quadrants for the player to decide between, then again splitting the selected quadrant into sub-quadrants until the desired precision is achieved.

There are other ways to cope with this, one of which is to heuristically generate only a small group of choices to decide from. Another method for games with a finite number of moves, which doesn't sacrifice completeness, is to change the single large decision into a series of sub-decisions. An example of this possibility with Chess might be to first let the player decide which piece to move, and after that decision has been made, to let the player decide which space to move the piece to.

2.2.4 Simultaneous turns

Games that incorporate sealed bid auctions require players to make decisions about how much to bid simultaneously, and then the rules resolve the bids into an outcome. In RoboRally players simultaneously construct very simple programs from instruction cards which are revealed simultaneously and the instructions are executed, adjudicat-

ing conflicts as instruction execution progresses. The possibility of multiple players making decisions simultaneously will have to be accommodated.

2.2.5 Cooperative play

Some games such as Bridge incorporate cooperative play which may affect a players decision making, but doesn't really affect the mechanics by which decisions are made. The affect on decision making is due to the difference in the goal, if a player can help a certain other player to win, they help themselves to win as well which means that the game can have multiple victors.

2.3 The board games framework

The idea of trying to implement the most general functionality is appealing because even if only a small amount of functionality can be extracted and implemented abstractly, it only has to be implemented once. It also gives the special cases of this general functionality, the individual games, some common structure, and serves to establish the boundaries of what must be implemented in the player. Constructing a general tier of functionality boils down to creating a set of super class (or interface) components that an instance of a game can sub-class and plug into the framework.

In addition to concerns for generality, it is desirable to isolate the player as much as possible in order to simplify it's automation. It was reasoned that the more things a player is responsible for, the more difficult it is to implement an artificially intelligent player. Anything that isn't absolutely required of a player should be moved elsewhere. One thing (perhaps the only thing) that is absolutely required of a player is that they must make a decision from among several choices when it's their turn. Allowances must be made so that the rules can provide the player with all possible choices, to relieve the player of the responsibility of generating choices.

After a few days with the Socratic method, several component concepts settled out:

- Player position
- Player choice
- Player action
- Game state
- Player view
- Player
- Rules
- Driver

The “Player position” is probably the simplest of these. It is often natural to refer to players in a generic way. In different games, players are generically referred to in different ways. In Bridge players are referred to by the four cardinal directions corresponding to the seats the players sit in, North and South versus East and West. In Chess players are referred to as black or white corresponding to the color of the pieces they are playing with, and in Tic-Tac-Toe as X or O, the symbol they use to claim boxes. This was left as a marker trait which should be implemented by an enumerated type, which could just be a set of integers if no more descriptive game specific option presents itself.

The “Player choice” is a little more complicated. A player choice represents a possible action that a player can take, or a set of such actions. The presence of this concept allows a component other than the player to determine the set of choices that a player has to choose from. The concept was included so that the responsibility for choice generation could be removed from the player. The rules certainly determine whether or not a choice is legal, so it doesn’t seem too much of stretch to have the rules pass the player all legal choices when practical. It will not always be practical however, sometimes there will just be too many choices to enumerate. In these cases the rules can still make it easy on the player and pass a set of descriptions of legal choices. In Chess, the set of all possible moves can be generated and passed. In

contrast, when betting in Texas hold'em there are a large number of distinct possible values for the bet amount. Rather than passing an explicit set containing each possible value, an implicit set can be passed, a description of the range. The player would need to understand the requirement of returning a result within the range.

Another example of a game in which it is impractical to pass all choices is Arimaa, which has an average of around 17,000 choices at each turn. The way Arimaa achieves this very high branching factor is through very complex turns. To deal with situations like this when they arise, a series of sub-decisions can be implemented in the rules. This would manifest as the same player taking multiple turns in order to make the small decisions which make up the larger decision required. In the more familiar example of Chess, this might entail the player being presented with a choice of which piece to move, and after that decision is made, being presented with a choice of which spot to move the piece to. In this way the communication channel (the amount of data that needs to be passed to the player) can be kept from being overloaded. Implementing sub-decisions may require some artificial game state to keep track of what stage of the decision making process the player is in. The purpose of the "Player choice" concept is to reduce the complexity of the player which in turn simplifies it's automated construction.

The "Player action" represents a decision that a player has made. In many cases a player action can be identical to a player choice, meaning that the player can select one of the choices it was offered as it's decision. More complex possibilities such as the bet amount in poker mentioned above may necessitate a separate type for the response defining the decision that the player has made. In the poker example, the player may be passed a range which implies the set of choices, it's bet must be between 100 and 10,000 chips, the player cannot respond with a range, but must decide on an individual value from this range, such as 500. This situation requires that different data types be used for the choices passed to the player, and the actions

that the player decides on.

The "Game state" is probably the most interesting component of any game (with the possible exception of the rules), but unfortunately it is not generalizable. There is nothing common to the state of all games, so the game state is left as a marker concept. It actually ends up being passed to nearly every method of the rules as it defines the context under which the rules must govern. A closely related concept is the "Player view", which represents those parts of the game state that a player has knowledge of. Obviously all state must be maintained in the game state, but not all state is known to all players. For example, it might be that no players know what the top card on a draw pile is, but it's still part of the game state. In Gin Rummy a player can know how many cards their opponent has in their hand, but not the denominations or suits of those cards. Unfortunately as a player's view of a game is a subset of the game state, and the game state is not generalizable, neither is the player's view, so it too remains a marker. In many games a player must make a decision without complete information, and the player view represents the partial information that is available to them.

A "Player" is obviously a very interesting component of any game. The player has been kept as simple as possible here to aid in automated construction. The sole responsibility of a player is that when presented with a decision, given the current state of a game (or a view of it), and a set of choices, it must respond with an action to take. Producing high quality (competitive) decisions automatically is what this work is all about.

The "Rules" are where the generic meat is. Though like a game state, there is not much that is common to the rules of different games, there is some that can be found. The rules are independent of instances of players, though the rules do need to refer to players in generic ways, for this purpose the rules use the player positions described above. An instance of the rules corresponds to a set number of players, that

is, the same game played with a different number of players would require a different instance of the rules. However, the same "set" of three player rules can be reused for games between any group of three player instances. The rules define the lion's share of the functionality for the games framework. The simpler responsibilities of the rules include the setup of the initial state of a game, determining which player's turn it is to make a decision, determining if the action that a player decides on is legal, whether or not a game as ended, and if so what players have won. Perhaps the primary function of the rules is producing the next state of the game in response to an action a player has taken. They must also be able to determine what a particular player is privy to among the information in the game state (they must produce player views). Finally, the slipperiest bit of "responsibility", they should provide a player with a simple set of choices to decide between. In the simplest case this can just be an enumeration of all possibilities, in more complex cases a description of a set of choices (either finite or infinite).

The Driver is the concrete implementation of the games framework. Given an instance of some rules, and a mapping of player positions to actual implementations of players (like a seating chart for players), it begins the game, requires decisions of players in turn, and plays the game out to it's conclusion.

To gain a higher precision in the explanation of the framework, a formalization is presented here. The aspects most closely related to players and strategies have not been covered in depth, but will be in later sections.

- Let P be the set of all player positions.
- Let G be the set of all game states for a particular game.
- Let a game state $g \in G$ be equivalent to the set of all information that defines the game state.
- For $g \in G$, $p \in P$, let $v(g,p) \subseteq g$ be player p''s view of the game state g.
- Let A be the set of all actions that can be taken by a player (possibly infinite).
- For $g \in G$, $p \in P$, let $c(g,p) \subseteq A$ be the choices function which maps a game state and player to the set of that player's legal actions (possibly infinite).
- For $g \in G$, $p \in P$, let $s_i(v(g,p_i), c(g,p_i)) = D(c_n)$ be the strategy function for player i, which maps player i''s view of the state $v(g,p_i)$ coupled with player i''s choices from the state $c(g,p_i)$ to a probability distribution of those choices $D(c_n)$ where
  - $\bigcup_k \{c_k\} = c(g,p_i)$, the choices available are indexed
  - $Q(c_k)$ is the probability assigned by player i (through s_i) to choice c_k,
  - and finally the distribution is formed by $D(c_n) = \sum_{k=1}^{n} Q(c_k)$, $D(c_n)$ was only used for symbolic brevity and consistency with the continuous case, note that $Q(c_k) = D(c_k) - D(c_{k-1})$
  - This treatment does not handle cases where an infinite number of choices are available. If a single continuous variable is involved an analogous probability density function will suffice, and if there are multiple continuous variables involved a multivariate density function would be necessary.
- For $g \in G$, $p \in P$, let $a(s_i(v(g,p_i), c(g,p_i)), r) \in c(g,p_i) \subseteq A$ be the action function which applies a randomly generated number r to the probability distribution given by s_i and produces the action that player i will take.
- For $g \in G$, let $p(g) \subseteq P$ be the player function which maps a game state to the set of players that must take actions this turn.
- For $g \in G$, let $m(g) = \bigcup_{p_i \in p(g)} \{(p_i, a(s_i(v(g,p_i), c(g,p_i)), r))\} \subseteq P \times A$, be the "group action", the collected actions of all players that must take an action this turn (m is for move). If $m = \oslash$ then g is a terminal state, and the game is over.
- For $g \in G$ let $t(g, m(g)) \in G$ be the state transitioned to as a result of all players taking their actions. The result of the function t may be probabilistic, depending on random events.
- Let $g' \in G$ be the initial state of the game.
- Cycling through the states of a game from start to finish is now possible. If we number the states of the game $g_0, g_1, ..., g_n$ where g_n is a terminal state of the game, we can define $g_0 = g'$,and recursively $g_i = t(g_{i-1}, m(g_{i-1}))$.

The correspondence between the concepts and the formalization are as follows:

Player position	the elements of the set P
Game state	the elements of the set G
Player view	the function $v(g \in G, p \in P) \subseteq g$
Player action	the elements of the set A
Player choice	the function $c(g \in G, p \in P) \subseteq A$
Player	the functions $s_i(v(g, p_i), c(g, p_i)) = D(c_n)$ and $a(s_i(v(g, p_i), c(g, p_i)), r) \in c(g, p_i) \subseteq A$
Rules	the tuple of (P, g', p, v, c, m, t)
Driver	the definitions $g_0 = g'$, and $g_i = t(g_{i-1}, m(g_{i-1}))$

As previously stated the Player concept including the details of $D(c_n)$is the subject of the remainder of the paper.

2.4 The code

2.4.1 The basics of Scala

The language chosen for the implementation was Scala[1], a fairly new language. Scala achieves platform independence by producing executables that run on the Java virtual machine. In fact, Scala is interoperable with Java, it can produce "jar" files just like Java, and jar files written in either language can be used in either language. Scala has the advantages of being a functional language in addition to being an object oriented language, as well as having far superior support for generics.

Scala has no primitive types, everything is an object. Things that are primitive types in Java such as '`int`' and '`boolean`' can only be expressed as their object equivalents '`Int`' and '`Boolean`' in Scala. There is not even a primitive array type in Scala, what in Java would be declared '`String[] strings = new String[5];`' is declared '`val strings: Array[String] = new Array[String](5)`' in Scala. This

difference in usage of the '`[]`' symbols may be a little confusing, in Java the square brackets are special symbols related to arrays, where in Scala they signify *type parameters*. Scala's `Array` class must be provided with a type parameter which specifies the type of object that can be stored in the `Array`, in Java the array type is primitive and the type of the object that can be stored precedes the '`[]`' symbols. The final bit of the call to the `Array` constructor, the '`(5)`' is the argument to the constructor of the `Array` class, and reserves space for five `String`s.

You may note that the Scala declaration of an array is longer than the Java declaration. It is actually one of Scala's design goals to remove as much "boilerplate" code as possible, and in general Scala code is much shorter than Java code. One element that helps make Scala code more concise is it's strong type inference mechanism, in most cases type annotations can be left out of the code altogether, and the compiler infers them for the programmer. The above declaration of the array can be reduced to '`val strings = new Array[String](5)`', Scala infers that the type of the variable `strings` is the same as the object being assigned to it. Scala goes so far in removing the burden of boilerplate code that even the statement terminator character '`;`' is not even required. Code listing 1 illustrates basic class and member declarations as well as examples of the use of type inference.

Code listing 1 This listing demonstrates class and member declarations, the elimination of unnecessary boilerplate clutter, and the advantages of type inference.

```
class Verbose{
  val variable: Int = 4;
  // The keyword 'val' is used to declare variables.
  // variable has type Int and is assigned the Int 4.

  def method(i: Int): String = {
    "s";
  }
  // the keyword 'def' is used to declare methods
  // every code block returns the final value encountered in the block
  // method takes an Int argument and returns the String "s"

  val function: (String => Int) = (s: String) => { 5; };
  // function is assigned a function which takes a String
  // and returns the Int 5

  val tuple: (Int, String, Int) = (1, "s", 2)
  // tuple is assigned a value compatible with it's type
}

class Concise{
  val variable = 4
  // the type of variable is inferred to be Int.

  def method(i: Int) = "s"
  // the type of the argument i cannot be inferred so must be provided
  // the return type of method is inferred to be String

  val function = (s: String) => 5
  // the type of function is inferred to be (String => Int)

  val tuple = (1, "s", 2)
  // the type of tuple is inferred to be (Int, String, Int).
}
```

One great example of the brevity afforded by Scala is it's tuple syntax. The word *tuple* is an *n-ary* reference to the sequence: single, double, triple, quadruple, quintuple, sextuple etc. A *tuple* in Scala is an ordered list of values which may

be of different types. A tuple is written in code as the list of values in between parenthesis, i.e. '`(1, "hello", 14.2)`'. The type of this tuple would be written similarly, `(Int, String, Double)`. Most container classes such as `Array`s can only hold one type of object, but tuples can hold many types. You can access the elements of a tuple individually. Most containers' first element is at index 0, but for tuples the natural numbers are used, the nth element is accessed by the `_`n member of the tuple (i.e. `(1, "s")._2 == "s"`).

Perhaps the greatest example of the brevity of Scala is it's function syntax. The type of the function is written in terms of it's argument types and return type. The type of a function that takes an integer and a string and returns a character would be written '`(Int, String) => Char`'. Parenthesis surround the argument types and the '`=>`' symbol separates the arguments from the return type. Even better, a function can be *implemented* with much the same syntax: `(i: Int, s: String) => s.charAt(i)` has the previously mentioned type, '`(Int, String) => Char`', and is an actual function which can be evaluated and will return the character at the ith position of the string `s`. In Java you would have to write a new class:

```
class F{char apply(int i, String s){return s.charAt(i);}}
```

Scala has classes, as Java does, but it also has something called a *trait*. Both classes and traits can have member variables and methods. Like Java, Scala does not allow multiple inheritance, but it does allow you to "mix in" multiple traits [2] which is similar to multiple inheritance. Usually when we think of inheritance, it is with a class extending another class. In Scala, a class can also extend a trait (and a trait can also extend either a class or trait). An important difference between classes and traits is that only one class or trait may be extended (single inheritance), but multiple traits may be mixed in, while no classes can be mixed in at all. Either classes or traits can have declared members that have no implementations, these members are

[2] Traits are mixed in using the keyword "with".

called *abstract*. If a class contains abstract members, then the class must be declared to be abstract, however, a trait is inherently abstract, so no special declaration is needed. You can only create an instance of a concrete (non-abstract) class, not an abstract class, and not a trait. Subtypes can implement the abstract members of their supertypes, and if the subtype contains no abstract members, either declared or inherited, then it is concrete, and can be instantiated. An abstract member of a class can only be implemented by a subtype of the class, but the abstract members of a *trait* can be implemented either by a subtype, or by a *supertype*. Traits can be mixed in at the leaves of the inheritance hierarchy, and can add functionality to a class. Code listing 2 is a concise example of some of the interplay between traits and classes.

Code listing 2 This code listing shows examples of a trait, a class extending a trait, an abstract class, a trait with concrete and abstract members, and a trait mixed in to a class declaration.

```
trait A{
  def a = "a"
}
// Trait A doesn't need any abstract members.

abstract class B extends A{
  def b: String
}
// Class B has inherited A's concrete method a.
// Class B must be abstract since the method b has no implementation.

trait BCallsA{
  def a: String
  def b = a
}
// Trait BCallsA has no relationship to either class A or class B.
// It declares a method b which is implemented
// by calling an abstract method a.
// traits need not (in fact, may not) be declared abstract.

class C extends B with BCallsA
// Class C has no class body, so the {} are omitted.
// C extends B so it has a concrete method a, and an abstract method b,
// C then mixes in BCallsA which implements the abstract method b.
// The trait's abstract method a is implemented by class B's method a.
// C need not be declared abstract, it has no abstract members.
```

2.4.2 Implementing the board games framework

The general code, or framework code, written in Scala, is listed and described here. The size of the code is small, but it effectively expresses the commonalities between games, as well as the methods of communication between the roles played by the components. Generic type parameters are used extensively throughout this code. This is done so that the framework code doesn't hide types from code that interacts with it. If external code passes a component in to the framework, and framework

code passes the component around internally, and then back out to the external code later, the same type that was passed in is returned. It doesn't return a limited view of the component that only exposes the members necessary to the framework (an interface), instead it returns the full blown type that was passed in, including any members that the external code depends on that the framework never needs to know about.

The marker traits

Code listing 3 The marker traits.

```
trait PlayerPosition
trait GameState
trait PlayerView
trait PlayerChoice
trait PlayerAction
```

The traits in code listing 3 are declared for use as markers, so that type arguments aren't accidentally used in inappropriate contexts. A class or trait must explicitly subtype or mix in the appropriate above trait if it will be used in the corresponding role within the framework. These traits are not generalizable for all games, so no more precise definition can be given to them.

`PlayerPosition` is meant to be implemented as a natural generic reference to a player, such as black and white in Chess, or X and O in Tic-Tac-Toe. `GameState` and `PlayerView` are closely related, `GameState` is the true state of the game while `PlayerViews` are used to communicate to a player the information that they have available to them for making decisions. In games with no hidden information the same class or trait can be used in both of these roles.

`PlayerChoice` and `PlayerAction` are also closely related traits, the reason for the dichotomy is that in games with very large (or infinite) branching factors, the

rules cannot enumerate all possible choices for the player. In games that are simple enough, the rules can enumerate all possible actions that the player can take, and the player can respond with the action they decide upon. In this case the same class or trait can be used in both of these roles. If the branching factor is too large (or infinite) though, it will only be possible for the `PlayerChoice` to be a description of the set of actions a player can take. In this case the player cannot respond with a `PlayerChoice`, as this is not an action but a description of a set of actions. The player must respond with something different, a `PlayerAction`.

The Player

Code listing 4 The Player trait.

```
trait Player[PV <: PlayerView,
             PC <: PlayerChoice,
             PA <: PlayerAction] {
  def apply(game: PV, choices: Set[PC]): PA
}
```

The '`<:`' symbol is the subtype operator. An example use from code listing 4, '`PV <: PlayerView`', restricts the type parameter `PV` of the `Player` trait to be a subtype of the `PlayerView` marker trait. The `Player` trait is implemented as a function object, which means that the `apply` method allows an instance of this trait to be called like a function. For example, if we have: `val player: Player[...]`, then we can call: `val action = player(game, choices)`, which uses the instance of the object as a function. This definition of the `Player` component allows for hidden information since it accepts an instance of the `PlayerView` marker trait instead of the conceptually full blown `GameState` trait. This definition also allows for games with innumerable choices since the `PlayerAction` returned isn't required to be of the same type as the `PlayerChoice`s that it is passed.

Strictly speaking the "choices" argument could be ignored by implementations of the `Player` as long as the returned `PlayerAction` was legal according to the `Rules`. However, this definition allows the `Player` to be supplied with choices, either explicitly or implicitly defined in order to remove some or all of the responsibility for generating choices from the `Player`. There is an indirect dependency here in that `Player`s will only work with `Rules` that meet their expectations of the `PlayerChoice` type provided. In most cases this is taken care of by the requirement that they must both use the same `PlayerChoice` type. If you tried to use a `Player` that expected all possible choices to be given, with a set of `Rules` that only provided a description, you would get a compilation error due to the different `PlayerChoice` type parameters used. The only situation which wouldn't provide a compilation error for diagnosis would be when the same `PlayerChoice` type was used by both the `Player` and the `Rules`, but they made different assumptions about what the internals of that type meant.

The Rules

Code listing 5 The Rules trait.

```
trait Rules[PP <: PlayerPosition,
            GS <: GameState,
            PV <: PlayerView,
            PC <: PlayerChoice,
            PA <: PlayerAction] {
  def players: Set[PP]
  def newGame: GS

  def nextPlayers(    game: GS): Set[PP]
  def playerView(     game: GS, player: PP): PV
  def choices(        game: GS, player: PP): Set[PC]
  def legal(          game: GS, player: PP, choice: PA): Boolean
  def executeChoices( game: GS, groupChoice: Map[PP, PA]): GS

  // game is over if there are no players with choices
  def gameOver( game: GS): Boolean =
    nextPlayers(game).flatMap(choices(game, _)).isEmpty
  def draw(     game: GS): Boolean
  def winners(  game: GS): Set[PP]
}
```

The `Rules` are the core of the definition of a game. The core responsibilities of the `Rules` corresponding to the trait's methods are:

- keep track of the set of `Player`s through `PlayerPosition` references
- provide newly initialized `GameState`s for play
- be able to tell what set of `Player`s need to make decisions next turn
- provide a useful set of `PlayerChoice`s for `Player`s
- determine whether a particular decision made by a `Player` is legal
- execute the `GameState` transition given the `Player`s' decisions
- know when the game is over
- know if there has been a draw
- know who the winners of the game are

This definition of the `Rules` given in code listing 5 allows for hidden information

by providing different `PlayerView` instances for each `Player` instead of `GameState` instances. The `Rules` allow for innumerable choices by providing choices for `Player`s in terms of `PlayerChoice`s, and accepting the decisions of `Player`s in terms of `PlayerAction`s. The `Rules` allow for simultaneous turns by providing a set of `nextPlayers` instead of a singular `nextPlayer`, as well as by accepting the decisions of multiple `Player`s at once when transitioning to the next `GameState`. And finally, the `Rules` allow for cooperative play by providing a set of winning `Player`s rather than a singular winning `Player`.

The `gameOver` method is actually implemented at this level of abstraction, when there are no `Player`s with choices, then the game is over. The implementation is very concise, and even readable to the trained eye. The `gameOver` method first calls the abstract `nextPlayers` method which returns a `Set[PP]`. The Scala `Set` type has a method: `def flatMap[B](f: (PP) => Traversable[B]): Set[B]`. This method has a type parameter `B`, but it need not be specified, it will be inferred. In the case of the `gameOver` method the type `B` will be `PC`. The method takes an argument `f` whose type is a function that takes a `PP` and returns a `Traversable[PC]`. It may not look like it, but we are using the `choices` method to form a function of this type, and passing it in. In this context the call `flatMap(choices(game, _))` is equivalent to the call `flatMap((pp: PP) => choices(game, pp))` in which a function is created inline and passed as the parameter `f`. The `choices` method returns a `Set[PC]` which is a subtype of `Traversable[PC]`, so the function we are passing has type `(PP) => Set[PC]` which is a subtype of the function the `flatMap` method is expecting. The compiler knows the type of the function that must be passed, and in this case that it has only a single argument, so it allows us to use the '`_`' symbol to stand in for the argument in the implementation of the function that we provide. This is called *lifting*. The `flatMap` method executes the function `f` on each element in

the `Set[PP]` returned by `nextPlayers` and concatenates[3] the results into a `Set[PC]` which is the list of all `PlayerChoices` for all of the `Players` that need to move this turn. If this set is empty, then the game is over.

The GameDriver

Code listing 6 The GameDriver object.

```
object GameDriver {
  def apply[PP <: PlayerPosition,
            GS <: GameState,
            PV <: PlayerView,
            PC <: PlayerChoice,
            PA <: PlayerAction,
            P  <: Player[PV, PC, PA],
            R  <: Rules[PP, GS, PV, PC, PA]]
           (rules: R, players: Map[PP, P]): GS = {
    var game = rules.newGame;
    while(!rules.gameOver(game)){
      val onTurn = rules.nextPlayers(game)
      val choices = Map() ++
        onTurn.map(seat =>
          { val view    = rules.playerView(game, seat)
            val options = rules.choices(game, seat)
            val choice  = players(seat)(view, options)
            require(rules.legal(game, seat, choice))
            (seat, choice)
          }
        )
      game = rules.executeChoices(game, choices)
    }
    game
  }
}
```

[3]Note that there is another method of the Set class called 'map', which does not concatenate the results of the function calls, it's return type would have been Set[Set[PC]]. flatMap "flattens" out the result of map for us, into a Set[PC].

Through the facilities of the `Rules`, we can actually set a game in motion and play it out to it's conclusion at this level of abstraction (given an appropriate set of components), this is the responsibility of the `GameDriver` shown in code listing 6. It never needs to be reimplemented for any specific game.

As you can see, the `GameDriver` is declared as an *object* rather than a class or a trait. An object in Scala is not a type (like a class or trait), but a singleton object, that is automatically constructed. Since objects are concrete instances (not types), they cannot have abstract members, constructor parameters (they have no constructors), or type parameters. Objects can extend other classes and traits, but nothing can extend an object since it's not a type. It's not possible to create new instances of an object, but the singleton instance can be passed around since it is an instance itself. An object can be referenced by it's name, in this case `GameDriver`. Objects are commonly used as repositories for code that would be *static* in Java, code that is global. Scala allows you to reuse the name of a class as the name of an object, when this is done the object and class are called *companions*. Typical functionality implemented in a companion object would be factory methods for constructing new instances of the companion class.

In the case of the `GameDriver` object there is no companion class, and the object implements only the `apply` method which allows it to be used as a function. The function accepts as arguments the rules of a game, and a map from the playing positions for that game (such as black or white in chess) to the instances of players playing in each position. The function uses the rules and players to play the game to it's conclusion and returns the completed `GameState`.

The first thing the `GameDriver` does is to assign the variable `game` a new `GameState` using the `rules.newGame` method. The `game` variable is declared with `var` instead of `val` which hasn't been discussed yet. Variables declared using `val` can never be assigned a new value, while variables declared using `var` can be reassigned. A `val`

declaration can be used when the programmer wants the compiler to help ensure immutability, and a `var` declaration must be used if the programmer wishes to reassign the variable. The while loop is executed once per turn until the game is over. Depending on the game, multiple players may take actions each turn. Within the loop, the `game` variable is reassigned the new `GameState` after any moves for that turn have been made.

The first thing that happens within the loop is to retrieve the set of `PlayerPosition`s that will take actions this turn from the `rules.nextPlayers` method. Next, a `Map` is built from each moving `PlayerPosition` to the `PlayerAction` selected by the `Player` in that position. To produce the `Map`, a factory method of the companion object of the `Map` class is used to get an empty `Map`, then an inline function is passed to the `onTurn.map` method, which will call the function for each of the set of players whose turn it is to move. The function has the type `(PlayerPosition) => (PlayerPosition, PlayerAction)`, it takes a `PlayerPosition` and returns a 2-tuple (a.k.a. *pair*) of the `PlayerPosition` and the `PlayerAction` selected by the player in that position. The result of the call to `onTurn.map` is a `Set[(PlayerPosition, PlayerAction)]`, this set is added to the empty `Map` to produce the `choices` variable, needed by the `rules.executeChoices` method to produce the succeeding `GameState`. Scala allows method names to use symbols, the method that adds the set of pairs to the map is `Map.++(xs: TraversableOnce[(PlayerPosition, PlayerAction)])`, the method name is simply '`++`'. The return type of the `onTurn.map` call is a `Set` which is a subtype of `TraversableOnce` so all the pairs are added to the map.

Scala infers the type of the `seat` variable in the inline function to be a `PlayerPosition` since that is the type required as the argument for the function passed to the `onTurn.map` method. We had to use the name `seat` instead of the extreme shorthand of the '`_`' symbol since the argument was used multiple times within the function. The function

maps a `PlayerPosition` to the `PlayerAction` chosen by the player sitting in that position. In order to do this, the function first uses `rules.playerView` to retrieve the view of the `GameState` "visible" to the player sitting in the argument `seat`. Then the function uses `rules.choices` to produce the set of options for the player. The view of the `GameState` and the set of options are then passed to the `Player` so that the player can make their decision. An assertion is then used with the `rules.legal` method, if the player has made an illegal choice, then the `GameDriver` throws an exception, and the game ceases. If the move is legal, then it is returned[4] as the mapping from the `PlayerPosition` to the `PlayerAction` taken by the player, and once all of these mappings have been collected, they are passed to the `rules.executeChoices` method and the next `GameState` is produced.

[4]Recall that the final value encountered within a block of code (enclosed by '{}' symbols) is the value returned by that block.

2.4.3 Some simplifications

Code listing 7 The SimpleRules trait.

```
trait SimpleRules[PP <: PlayerPosition,
                  GS <: GameState with PlayerView
                                  with PlayerChoice
                                  with PlayerAction]
                 extends Rules[PP, GS, GS, GS, GS]
                 with    NonSimultaneous[PP, GS, GS, GS] {

  def winner(game: GS): PP

  def playerView(game: GS, player: PP) = game

  def legal(game: GS, choice: GS) = {
    choices(game).contains(choice)
  }

  def executeChoice(game: GS, choice: GS) = {
      require(legal(game, choice))
      choice
  }

  def winners(game: GS): Set[PP] = Set(winner(game))

}
```

While it is very nice to have a most general case to specialize for any specific case that may arise, for many of the specific cases this leads to a lot of clutter such as passing in the same type parameter up to four times. To simplify the process of constructing the specific case of a game, adapter traits were written for the simplest versions of games. The `SimpleRules` trait can be sub-classed to implement any game that has no hidden information, no chance, a reasonable branching factor, no simultaneous actions, and no cooperative play. The word "simple" may be a misnomer, the game Chess can be implemented as a subclass of `SimpleRules`, it covers a very large class

of games, however among the spectrum of games that the framework can be used for, the `SimpleRules` can only represent the simplest.

The advantage `SimpleRules` grants is that only two type parameters need to be specified, the `PlayerPosition` and the `GameState`, and method signatures are adapted to simpler versions that make more sense in the context of a simpler game. An example is that the winners method has been implemented in terms of a new abstract method, winner. That means that subclasses will not need to implement the winners method, but only the winner method which returns only a single winner, which is what most simple games have.

Code listing 8 The SimplePlayer trait.

```
trait SimplePlayer[GS <: GameState with PlayerView
                                    with PlayerChoice
                                    with PlayerAction]
                  extends Player[GS, GS, GS]
```

`SimplePlayer` is an simplification of `Player` analogous to `SimpleRules` and `Rules`. It reduces the number of type arguments to one for games that have no hidden information, no chance, and a reasonable branching factor. The `Player.apply` method is reduced to `def apply(game: GS, choices: Set[GS]): GS`.

Chapter 3

Strategy

3.1 Game theory

Game theory formulates a game as a tree[5] in which each node represents a state of the game, and a state in which the game has ended is called a terminal state. For all non-terminal states a player has a decision to make from among several choices. Each of these choices is represented as an edge which leads from the state, to the state which is a result of making that decision. The playing out of a game is represented in this tree as a path from the root to a terminal node. Note that duplicate physical configurations of a game (i.e. a Chess board setup) could be present in multiple nodes of the tree of a game. A node in the tree represents the entire history of the game that lead to that state, so any game in which different sequences of moves can lead to the same board setup will have duplicates. Also note that elements of game history such as whether an “en passant” capture is available, or whether a player could claim a draw due to repeated board positions are represented by each node in addition to the physical configuration of the game.

If we examine this model in light of the aspects of board games discussed earlier, some of them prove problematic and require adjustment. Hidden information poses

no problem, the game state is implicit in the node of the tree, as is the subset of information that a player is aware of, it doesn't affect the choices a player can make, only the players ability to evaluate those choices. Chance does pose a bit of a problem however, the branches of the tree represent the choices a player can make, and the endpoint of an edge represents the state that the decision led to. If it's up to chance to determine what state the game is in after the player makes their decision, how could this be represented? The way this problem is solved is by introducing a new pseudo-player to play the part of chance. When an event occurs that depends on chance, an artificial node is introduced, we can't determine the state of the game yet because it depends on chance. From this artificial state, it's the turn of the pseudo-player "chance", and the "choices" available to the chance player branch out from this node. The chance player makes their "decision" (randomly), and we arrive at a real node in the tree that represents the state of the game after the chance event took place. This is depicted in figure 3.1.

Innumerable choices leads to a kind of problem as well, if there are infinite choices from some node, then the tree is impossible to draw. However, the tree is impractical to draw in most cases even for small games, the tree of the game Tic-Tac-Toe would require 255,168 terminal nodes (ignoring the non-terminal nodes). If the trees are impractical to draw anyway, and only serve as a mental model, then let us think of a state with infinite choices as a node with infinite children. Simultaneous turns requires some adjustment as well. Rather than each outgoing edge representing a choice of a single player, we must enumerate all possible collective choices, and let each of these be represented by an edge. For example, if two players are making simultaneous decisions and each must decide between A and B, then the branches from that node could be labelled AA, AB, BA, and BB corresponding to all possible combinations of the players' decisions. Finally, cooperative play may affect the decision a player makes, but it doesn't affect the tree of choices. So the tree is general enough to serve

as a model for the most complex game within our scope.

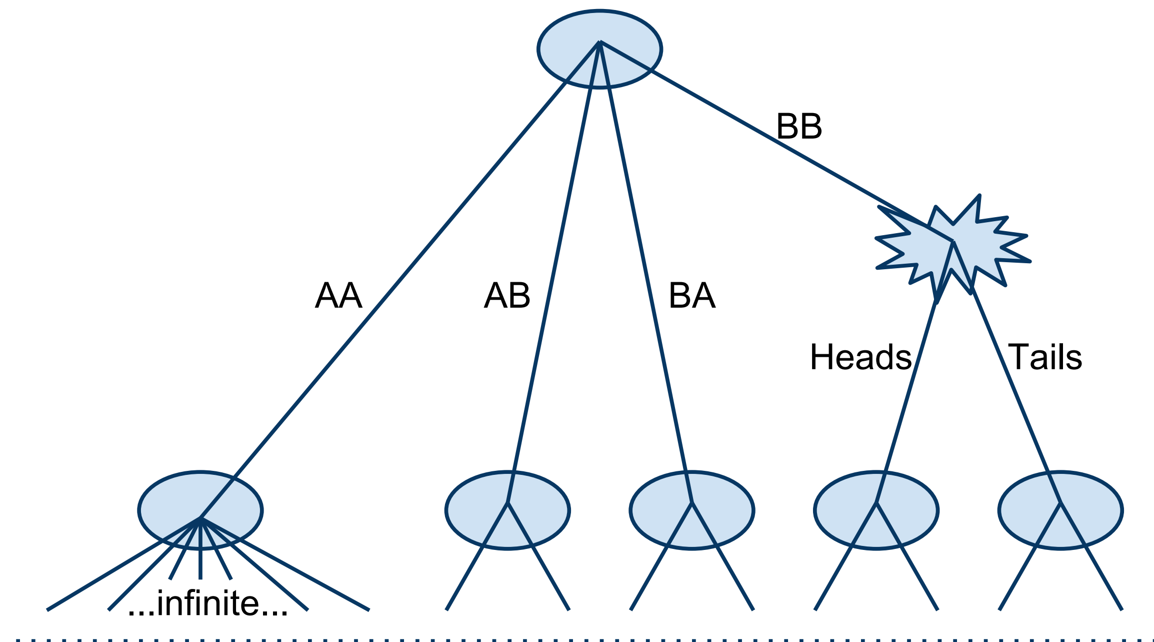

Figure 3.1: This diagram depicts collective choices being made from the root node, an artificial node where the decision between Heads and Tails is made by the pseudo-player chance, and a node that has an infinite number of choices.

English language usage of the word strategy is vague, typically it's used to describe a policy guiding and restricting certain actions, but there may be many alternate actions that are all consistent with the strategy. Game theory makes the definition much more specific[5], a pure strategy defines, for each point in the game tree described above, the single action that will be taken. If even a single decision is altered, then you've got a separate, distinct strategy. Note that each decision which a pure strategy defines, makes all nodes of any sibling branch unreachable. This means that in a practical sense, a strategy only needs to define which action to take for each node of the subgraph of the game tree that may actually be encountered.

The different subgraphs defined by a player's strategy when moving in different playing positions (i.e. playing as white versus playing as black in Chess) are essentially disjoint. Some nodes (game states) will be present in both subgraphs, but the decision

made at such a node will be made by different players in the different subgraphs. Consider the initial state of the game as the simplest example. In Chess, both a white strategy and a black strategy must include the root node (the initial game state). In a white strategy it is the player owning the strategy who has a decision to make, in a black strategy it is the opponent of the player owning the strategy who makes the decision. So the root node appears in both the black and white subgraphs, but the decision made at the root node is made by different players in the different subgraphs. In fact, given a set of strategies covering all playing positions, the nodes that their subgraphs have in common will define a path through the game tree, the game that will be played out among them as they make decisions. This means that a player's strategies when playing in different positions are distinct. Even though there are very likely important aspects of play that are the same for both players, with a strategy defined in this way, a strategy for black would not apply when a player was playing as white.

Game theory defines the outcomes of games in terms of a number called the utility[5], which is represents how good or bad the outcome of the game is for a player. Utility plays a role in economic games such as auctions where one bidding strategy might lead to higher utility than another. The games that we are concerned with only have winners and losers, which may be defined as utilities of 1 and -1 respectively. Another concept from game theory is that of an equilibrium[5]. An equilibrium is a selection of strategy by the players such that if one of the players changed their strategy, their utility would decrease, which means that there would be no incentive to switch. Some games have equilibria with pure strategies, but it has been proven that all games have at least one equilibrium for *mixed* strategies (this is the famous Nash equilibrium[6]). A *mixed* strategy combines several pure strategies into a single strategy. A mixed strategy is a probability distribution over the set of all pure strategies. When a new game is started one of the pure strategies

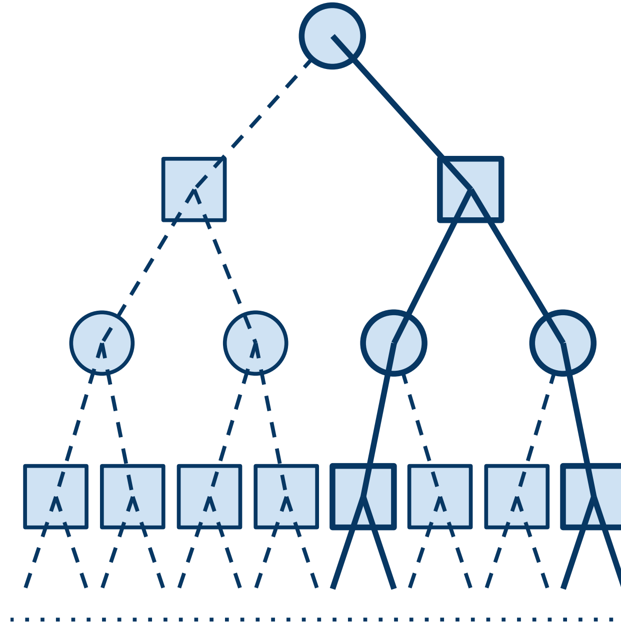

Figure 3.2: The solid lines depict the subgraph defined by a strategy for the circle player, each circle has only one solid outgoing edge. The nodes connected by dashed lines cannot be reached if the strategy is used. Since the strategy only specifies the decisions of the circle player, all choices for the square player must be considered.

is selected probabilistically. Mixed strategies can be thought of as a generalization of pure strategies, and vice versa, a pure strategy can be thought of as a special case of a mixed strategy in which one pure strategy has a selection probability of 1, and all others have a probability of 0. It seems unwise to disallow a mixed strategy as an option since if the opponent uses one they may have an advantage.

The concept of a `Player` maps to a game theoretic strategy. It should be noted that the term $D(c_n)$, the probability distribution over a players choices from a state from the formalization of the board games framework, does not precisely map to the concept of the mixed strategy. Game theory defines a mixed strategy as a probability distribution over the set of pure strategies[5], that definition is more coarsely grained. Once a pure strategy has been selected there are no distributions within the pure strategy. The distribution $D(c_n)$ provided by the function s_i is a probability distribution over a players choices from each state, this is very fine grained, there

are many distributions within a single strategy defined by s_i. The strategy function was defined this way because for games of reasonable size it is impractical (if not impossible) to generate all pure strategies in order to define a mixed strategy over them. It will become clear in section 3.5 how it is possible to define a distribution over the choices from every state. Note that a distribution over choices strategy is a generalization of pure strategies, just as mixed strategies are. A special case of a distribution over choices would be a distribution in which one of the choices had a probability of 1, and the rest had a probability of 0. A strategy where all states had distributions of this form would represent a pure strategy. It is my conjecture that a strategy made up of distributions over choices is also a generalization of a mixed strategy, that is, that a mixed strategy can be expressed as a special case of a distribution over choices strategy. In the framework, distribution over choices strategies were used so that something similar to mixed strategies (if not a generalization of them) could be involved in a practical way. This may have unforeseen consequences.

3.2 Current methods

Ideally we would have time to search the entire tree of a game and find the perfect strategy. For games that do permit complete search, an algorithm called mini-max[7], which minimizes maximum loss (of utility), is used to find the perfect strategy. It does a depth first search, and evaluates the utility of the terminal states of the tree (for our purposes evaluations are either 1 for a win, -1 for a loss, or 0 for a draw). These utilities percolate up the tree based on which players" turn it is in a given state. We can determine the value of any state for which we know the utility of all of it's branches, at the beginning of the search these will be states one step back from the end of the terminal states. If we get to make the decision, then we will select the branch that gives us the highest utility, and if our opponent makes the decision, they

will select the lowest. Once all nodes have been labeled, we know that from a given state we can pick any branch with the same utility as the current node, and safely achieve our maximum utility (which means we will win if it's possible).

To improve the performance of the mini-max algorithm, and possibly allow it to be effective for games with slightly larger trees, we can attempt to prune branches from the tree. The basic concept is that if we have search to the terminal nodes for some subtree, then we have a guarantee of a minimum and maximum utility for ourselves within that subtree. When continuing the search on that subtree, if we find a node at which our opponent can make a decision that would decrease our maximum utility, we know that we would not make the decision that put our opponent in that position, so we can prune part of the tree. Likewise if we find a node for which we could make a decision that would increase our minimum utility we know that our opponent would not make the decision that led us to that state, so again we can prune part of the tree. This is called alpha-beta pruning[8] after the parameters used (Greek letters) to keep track of the minimum and maximum utilities. Alpha-beta pruning can significantly improve the performance of mini-max without sacrificing completeness, the alpha-beta solution will be exactly as good as full blown mini-max, just faster. It is still examining most of the nodes of the tree.

Unfortunately, with reasonable sized games we don't have enough time to do a complete search, so other means have been devised. The first step is realizing that we won't be able to get true values for utilities since our search won't be able to descend all the way to the terminal nodes. To compensate for this, we must use a heuristic, which is an estimate of our utility at a given node. Next, we have to determine how to decide when to stop our descent into the tree since we know we cannot go all the way to the terminal nodes. The common method is called iterative deepening[8]. It selects a maximum depth, runs a depth first search (which terminates at that depth), then increases the depth and runs the search again to the new depth. This search

can use alpha-beta pruning just as mini-max did, and can also take advantage of information gained from the previous shallower searches.

If we could create a perfect heuristic, the values on the nodes would look just like they did as a result of the mini-max algorithm. If we had a heuristic that was slightly less than perfect, and got more accurate the closer it was to the end of the game, perhaps it would report 0 at the beginning of a game, and for good paths numbers increasing towards 1, and for bad paths numbers decreasing towards -1. This example is only meant as an illustration of a smooth heuristic (monotonic) in order to contrast it with some real life examples. In Chess, with heuristics based on material, in sections of the tree where pieces are captured, large swings in the values of the heuristic occur. If we only look into the tree as far as a capture we make, that branch looks favorable, but if we look a little farther and our opponent is immediately able to respond with a better capture of their own then the branch looks unfavorable. If we end up trading pieces, but make a slight gain in position, then the heuristic would go through drastic swings, but then calm down after the swings are over at a slightly higher value than before the swings. This concept is known as quiescence[8], and can be intentionally applied during the search. The value of the heuristic is trusted less if it is swinging wildly, and the search can be continued until the heuristic is quiescent which we assume will be a more reliable value.

Note that a method of the form described above represents a pure strategy, and can be adapted for use in the board games framework. The method above is designed to return the action with the highest estimated utility, if it were adapted to return a distribution in which the action with the highest estimated utility had probability 1 and all others 0 then it would fit into the framework. At each evaluation it would run a search in the game tree for it's allotted amount of time, and then return it's decision. This is an example of a function that maps game states to numbers (utilities in this case) being adapted into providing a distribution over the choices from a state,

as required by the board game framework. The role that this function would play in the board game framework is that of a game theoretic strategy, and therefore, a player.

3.3 A different approach

The heuristic that underlies the alpha-beta pruning tree search above is flawed, and these flaws are accepted for the sake of simplicity, and perhaps intuitiveness. The search is run in order to ameliorate the flaws in the heuristic. The heuristic bundled together with the search is essentially just a different (almost certainly superior) heuristic. Let's consider this bundled heuristic not as a search, but just as a function that returns a numerical evaluation of the choices available from a state. It is a slow function in the sense that the better estimate we want, the longer we must allow the function to run. Could this function be sped up? The search that is run has been studied in it's own domain very rigorously, and it's doubtful that this change in perspective would shed any new light on it, but what about the heuristic?

Could we find some other function to evaluate the choices from a state? Of course a function could simply be generated at random, but we are in search of a superior function. If we could find a less flawed heuristic, less searching would be necessary to overcome it's flaws. Extrapolating on the idea of a less flawed heuristic in the extreme takes us to some ultimate heuristic for which no search is required at all. In fact this would no longer be a heuristic, but a solution of the game (though proving it was so would be a different matter entirely). This seems a worthy goal indeed.

This approach is significantly different than the tree searching approach. The tree searching techniques look from the root "along" the tree toward the terminal states of the game. This approach is orthogonal to that other in the sense that it looks at the tree more like we would see it drawn here on paper in front of us. The search that

must be performed with this approach, in the pure strategy case, is a search for a subgraph of the game tree in which each node requiring a decision from the player has only one branch (the predetermined decision). This is a search for a subgraph rather than a repeated search for the next node of a path within a tree. In the distribution over choices strategy case (different from standard mixed strategies) we are searching for an assignment of a probability to each branch of the game tree such that the probabilities branching from each node sum to 1. Once we have our function, to use it's guidance as a pure strategy, we simply evaluate each child of the current game state, and select the one with the highest utility as the move to make. To use the function for a distribution over choices strategy, we can normalize the utilities for all of the children into a probability distribution, and select a move accordingly.

The function needed for the heuristic must take complex data as input, the state of a game, and returning a number to be interpreted as the utility of that state. Each function whose returned values lead to different decisions represents a different strategy, we need to be able to search among the different possible functions for the better strategies (and hopefully the best). We propose now that an artificial neural network in it's capacity as a universal function approximator[18] may be used for this function. We propose that genetic algorithms may be used to evolve a population of these "players" to a high degree of skill.

3.4 The simple neural network implementation

An artificial neuron is a model of a biological neuron. The main parts of biological neurons are the cell body, dendrites, and the axon. The dendrites are a forest of short, thin, branching protrusions from the cell body, they are the sensory apparatus for the cell, they receive input from other cells. The axon is a distinct long thin protrusion (up to a meter in humans) which may also branch. Dendrites meet the

axon terminals of other neurons at small gaps called synapses. The axon's purpose is to carry pulses from the cell body, and transmit them to the dendrites of other neurons.

When a neuron fires, it sends a pulse out of it's axon either in the form of neurotransmitter chemical, or even as an electrical voltage. Interestingly, this pulse is binary, either on or off, a digital signal in the natural world. Dendrites that meet a firing axon sense the pulse and pass it along to the cell body. The size of the gap at the synapse (and other mechanisms) can diminish the strength of the signal passed on to the cell body. When a neuron's cell body has received enough stimulation through it's dendrites it fires a pulse of it's own through its axon.

Some neurons may fire very easily and often, and some rarely. The factors affecting ease of firing are the threshold value, and synapses strength. If a neuron has a high threshold, it will require more total stimulation before firing it's own pulse. If most of the input synapses for a neuron are weak, then more pulses across those synapses will be required before the neuron reaches it's threshold. A particular neuron may have strong synapses with all of it's outgoing neighbors so that when it fires, it sets off a storm of activity, causing even the rarely firing neurons to fire by causing all of their incoming neighbors to fire. Alternatively a particular neuron firing may cause only a very specific chain of neurons to fire. In fact a particular neuron may cause a chain of neurons to fire that ends with itself, indirectly providing stimulus for it to fire again.

The typical abstraction used in an artificial neural network makes each artificial neuron a node in a network, which is a directed graph with weighted edges. Edges represent the axon-synapse-dendrite link between two neurons, and the weights are used as multipliers, representing the effect that the gap (and other mechanisms) have on the pulse being transmitted. Commonly firing rates are used rather than simulating the actual firing of individual neurons. This means that rather than summing

a neuron's weighted input values and outputting a 1 or 0 based on a threshold, it sums the weighted input *rates*, and passes this value to the transfer function which determines at what *rate* the neuron will fire under the given stimulus. The unit of the firing rates is unimportant to the abstraction, it's only important that they all are in the same units, we can imagine that they are all in Hz for example.

In order to evaluate the network, there must be external input, and one or more neurons in the network must be designated as output neurons. The external input is in the form of source nodes in the network, with no incoming edges. The output values of the source nodes are not calculated, but set directly when the network is to be evaluated. They provide input to other neurons in the standard way. If a neuron is a member of a cycle in the network, it may contribute to causing itself to fire again, networks with cycles are called recurrent networks. In recurrent networks, once the input values are set, firing rates may fluctuate for a while, but then come to rest in a stable state, it is then that the output values of the network can be read. In networks with no cycles, called feed-forward networks, no time is needed for stabilization, the entire network can be evaluated once, and the output values read.

Evaluating the network in this way gives us a mapping from an input tuple to an output tuple, and for different input tuples, we can get different output tuples. The network can be thought of as a function, in fact the network *is* a function. Consider a network with input values i_1 and i_2, and nodes a, and b where b is the only output node (depicted in Fig. 3.3). For this example let the weight from b to a, $w_{b,a} = 0$ (there is no edge from b to a, this is a network with no cycles). Let the transfer functions of a and b be f_a and f_b respectively. The output value of a is $f_a\big((w_{i_1,a} \times i_1) + (w_{i_2,a} \times i_2)\big)$, and the output value of b (and the network) is $f_b\Bigg((w_{i_1,b} \times i_1) + (w_{i_2,b} \times i_2) + \Big(w_{a,b} \times f_a\big((w_{i_1,a} \times i_1) + (w_{i_2,a} \times i_2)\big)\Big)\Bigg)$. If there were cycles in the network, it would be more difficult to find a closed form, but it could still be evaluated iteratively until the output values stabilized, though theoretically

the values may oscillate and not converge.

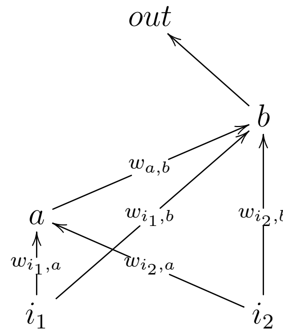

Figure 3.3: This diagram shows the nodes and weights of a simple network whose output at b is just a function of it's inputs at i1 and i2.

The network can be thought of as remembering the mappings between different values. This makes more intuitive sense if you think of it as having been trained for a purpose rather than just mapping random values. So where are these mappings stored? The only factors that go into determining the mappings are the transfer functions and the weights between nodes. The weights establish topology as well, a zeroed weight "removes" an edge. When training neural networks, typically the transfer functions are kept constant, and the weights are allowed to vary. A real number is a much simpler value to vary than a function, a real number can only go up or down. It's important to note that if the transfer functions are all made linear, then the function of the network will be linear, since the output of each node will be a linear combination of it's inputs. If non-linear network behavior is desired, then a non-linear transfer function must be selected, a sigmoid function (which is non-linear) is commonly used for it's ease of differentiation.

Feed-forward networks (those with no cycles) can be organized into layers, where a layer is a subset of nodes of the network, and each node in the subset only has edges leading to nodes in higher layers (*see Fig. 3.4*). We know from Cybenko[18] that a feed forward network with just an input layer, output layer, and one additional "hidden" layer in between can approximate any function. Since we intend to use genetic

algorithms to optimize these networks (their weights), rather than predetermining the topology of the network, we should give the genetic algorithm as much room as possible to work within. We will use a "complete" feed-forward network, meaning that we will include all possible edges without inducing a cycle. This means that a network of n non-input nodes will have n layers, each with 1 node. The node in layer 1 will provide input to the other $n-1$ nodes, and the node in layer 2 will provide input to the $n-2$ nodes in the $n-2$ layers above it etc. This means that the network will contain $\frac{n^2-n}{2}$ edges. If the genetic algorithm determines that it is optimal to zero out the weight of an edge, essentially removing the edge from the network, it can. It may be that with more layers, it's easier for the genetic algorithm to approximate the function we need for our strategy.

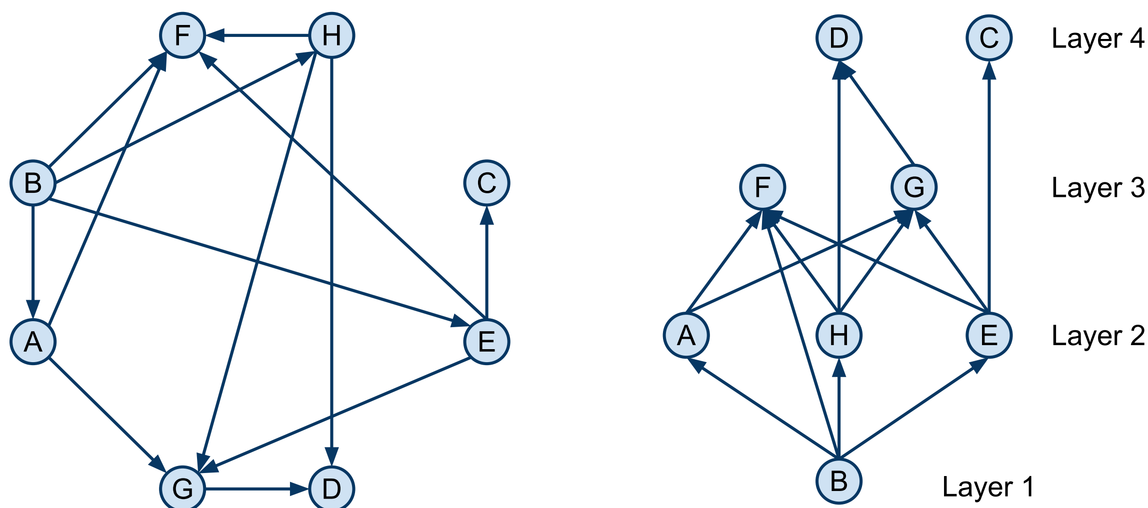

Figure 3.4: This diagram shows a feed-forward network, laid out first circularly, then in layers. Any missing edge can be thought of as having a weight of zero (i.e. BC or BD).

To use a neural network as a function mapping game states to utilities, for use as a strategy, we must write a "sensor" which translates a game state into a series of numbers to plug into the input nodes. To avoid biasing the strategy or the search, we will use a binary representation of the game state[1] which will allow the genetic

[1]In the example of Tic-Tac-Toe, a possible sensor would translate a game state into a twenty bit

algorithm to guide the network to use any more complex features of the state that are important. Rather than trying for a model that allows the genetic algorithm to determine the number of nodes necessary in the network, we can simply run with a small number of nodes, and increase the number of nodes until the quality of the player ceases to increase.

3.5 The search space

We've discussed the perspective of a strategy as a subgraph of a game tree, specifically that for pure strategies, at a node where a player has choices, that player's strategy must specify the choice that will be made. For a distribution over choices strategy, when a player has a choice, that player's strategy must specify a probability distribution over those choices. In both cases, all possible opponent responses must be included in the tree that represents the strategy.

Let's say that the root node of the game tree of Tic-Tac-Toe (the empty board) is at depth zero, and that the depth of a node is equivalent to the number of moves that have been made up to that point in the game. The root of the game tree has nine branches, since the first player can play in any of the nine open boxes. This gives us a total of nine nodes at depth one. From each of these nine game states, there are eight branches, since the second player can play in any of the eight boxes that the first player left empty. This gives us a total of $(9 \times 8) = 72$ nodes at depth two. Progressing in this manner to a full board would form a tree with $\prod_{i=0}^{d-1}(9-i)$ nodes at depth d, leading to $9! = 362{,}880$ leaves (at depth nine). Some games will end as early as depth 5, so some of the branches will not reach the full depth of nine. Therefore 9! is an overestimate of the number of leaves, the actual number of leaves

string by setting the first bit to whether or not it's X's turn to move, the second to whether it's O's turn, then according to a predetermined ordering of the nine boxes, the next nine corresponding to a box containing an X, and the final nine corresponding to a box containing an O. There are pairs of bits per box, of which, at most one will be set, corresponding to the box being empty, containing an X, or containing an O. Both cannot be set because a box cannot contain both.

in the tree of Tic-Tac-Toe is 255,168. Some of the nodes in the tree may be identical, but every path in the tree that starts at the root is unique to the end node of the path.

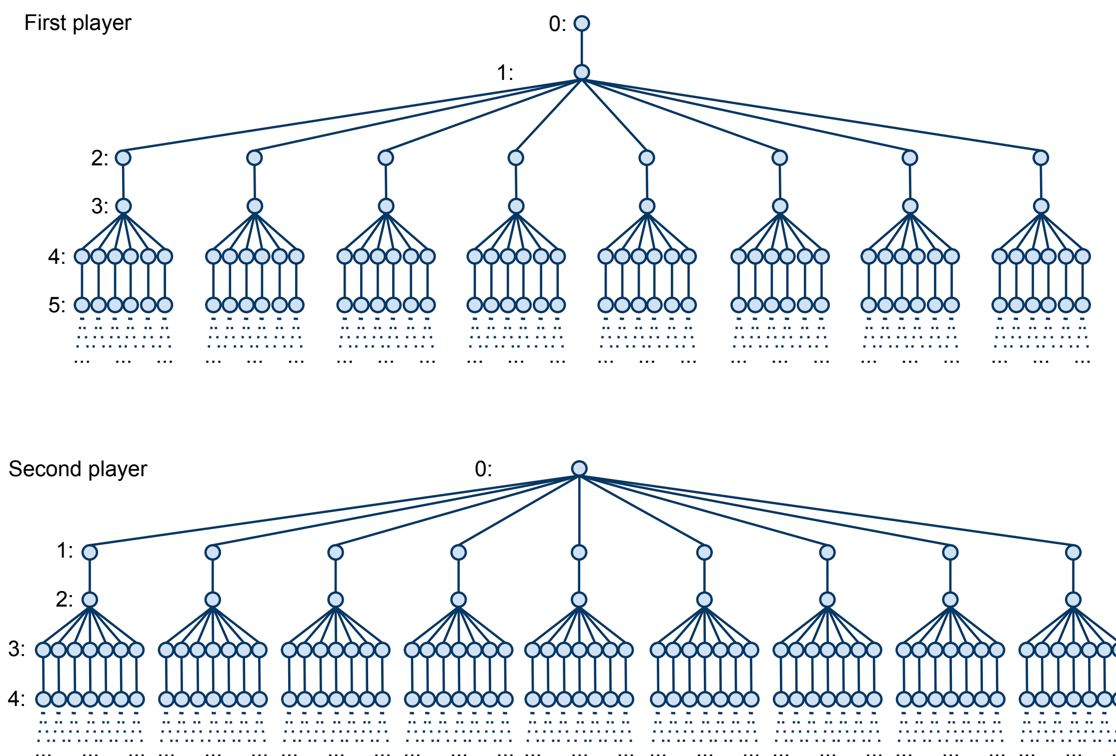

Figure 3.5: This diagram shows the tree structure of a subgraph strategy for Tic-Tac-Toe for the first player to a depth of five, and for the second player to a depth of four. The first player has included a single initial move, then all eight second player responses, then one first player response per second player response, etc. The second player strategy has included all nine initial first player moves, then one second player response for each, then all seven first player responses, etc.

Now let us consider a subgraph[2] of the game tree that represents a pure strategy for the first player. At the root node (the empty board) the first player has nine choices, and their strategy must specify which was decided upon. So the root of the subgraph will have a single branch representing the choice specified by the strategy.

[2]Note that any connected subgraph of a tree is also a tree. Typically the word subtree is used refer to the entire tree below some node other than the root, since this is not our meaning we use the word subgraph here. Since the subgraphs we refer to are also trees, we may refer to the "root", a "branch", or "leaves" of a subgraph, or the "depth" of a node in a subgraph.

The second move is up to the second player, and of the eight options available, the first player cannot know which the second player will select, so all eight must be included. Therefore the single node at depth 1 in the tree will have eight branches. Each of the eight nodes at depth 2 has seven branches in the game tree, but the first player's strategy must specify one of these seven, which means that each of the eight nodes at depth 2 has only a single branch. Proceeding in this manner, we can see that in the subgraph for a strategy for the first player, every node at an even numbered depth will have a single branch representing the first players response to a move by the second player, meaning that each odd depth will have the same number of nodes as the even depth above it. The subgraph will have $\prod_{i=0}^{\lfloor d/2 \rfloor -1} 8-2i$ nodes at depth d (384 leaves at depth 9), once again ignoring the fact that many games will end early. In similar fashion we can calculate an overestimate on the number of leaves in a subgraph strategy for the second player, which has $\prod_{i=0}^{\lfloor \frac{d+1}{2} \rfloor -1} 9-2i$ nodes at depth d (945 nodes at depth 9). These numbers give us some inkling of how complex the task of defining a strategy may be, as well as the proportion of the game tree that a strategy must cover.

Now that we have established what it is that we're searching for, we can consider the question of how large the search space is. How many subgraphs of the game tree of Tic-Tac-Toe are there of the structure described above? Since a pure strategy completely defines every decision that a player will make, merely changing a single decision results in a distinct strategy. If we count up the number of ways that the set of all decisions can be made, we will have the number of possible strategies, and the size of the search space. Since we know the structure of a strategy, and know each place that a decision is required, we can simply multiply the different ways that each decision could be made together. We know which depths a player makes decisions at, and we know how many choices they have at each node of a given depth. If we raise the number of choices to the power of the number of nodes, we will know how

many ways the decisions at that depth can be made. If we then multiply the numbers of ways the decisions at the different depths can be made, we will have the total number of strategies. Using the formulas derived above, for the first player we have $\prod_{d\in\{0,2,4,6,8\}}(9-d)^{\Pi_{i=0}^{\lfloor d/2\rfloor-1} 8-2i} = 9^1 \times 7^8 \times 5^{8\times6} \times 3^{8\times6\times4} \times 1^{8\times6\times4\times2} = 7.46 \times 10^{132}$. For the second player we have $\prod_{d\in\{1,3,5,7\}}(9-d)^{\Pi_{i=0}^{\lfloor \frac{d+1}{2}\rfloor-1} 9-2i} = 8^9 \times 6^{9\times7} \times 4^{9\times7\times5} \times 2^{9\times7\times5\times3} = 1.87 \times 10^{531}$. To form a complete player with a strategy for playing first, and a strategy for playing second, we have to again multiply these values together, the final overestimate of possible player strategies is $7.46 \times 10^{132} \times 1.87 \times 10^{531} = 1.39 \times 10^{664}$. This number only applies to pure strategies, distribution over choices strategies would clearly be a continuous (infinite) search space.

Tic-Tac-Toe is a small enough game that the entire game tree can be traversed, having a maximum depth of 9 and only 255,168 leaves. Most other games more interesting to humans are exponentially more complex, far to large to traverse, and therefore to represent explicitly. For this reason, we have an implicit definition of the game tree, the rules of the game. The rules provide a concise definition of what state transitions are legal. This is brought up to point out the transformation this form of strategy definition makes to the search space. It was noted earlier that some nodes in the game tree may be identical, though paths from the root are unique. By identical, it is meant that there is no difference between the game states represented by two identical nodes, and since the rules can only operate on game states, this implies that the subtrees of identical nodes are identical. Since the states and subtrees are identical, and child states are produced implicitly, we can consider that identical subtrees are the same subtree rather than duplicate copies. This means that two parent states may lead to the same child state, which means that the game tree is not a tree. For cycle-free games, the game tree reduces to a feed-forward network, for games with cycles, the game tree reduces only to a general directed graph. Tic-Tac-Toe is cycle-free because at each turn a box is filled, and there is no way to empty

out a box. This compression of the game tree is due to the redundancy within the tree, and since a strategy is defined on the game tree, the size of a strategy, and total number of strategies (the size of the search space) is also compressed.

Further simplification of the search space can be recognized through observing that some moves are more important to get right than others. A strategy that gets the important moves right will win more often than a strategy that gets only less important moves right. For example, if a player has a move that will take the game into a branch of the tree (or graph) where the game always ends in victory, that is a more important move than one internal to that branch, once in that branch random moves would suffice for victory. The set of strategies that are identical except for the moves made within that branch are, for practical purposes identical strategies, they have the same unavoidable result. Sets of strategies like this can be thought of as plateaus in the search space where many very similar strategies are equivalent.

3.6 How the search might proceed

Only for solved games (such as Tic-Tac-Toe) is there an objective measure of the quality of a strategy. For unsolved games, a strategy can be shown to be flawed by showing that there is some other strategy that is superior, but a strategy cannot be shown to have no flaws[3]. So the quality of a strategy is measured relative to other strategies, and this measurement is inexact. Say we have a set of strategies S where $\{A, B\} \subset S$. Strategy A wins against every other strategy in the set except for B, and strategy B loses to every other strategy in the set except for A. Most ranking systems would say that for larger sets S, A is the best strategy in the set, and that strategy B was the worst, even though the worst beats the best. There is a system for computing estimated relative skill levels called TrueSkill[4] that, based on a set of win loss records, even accounting for individuals participating on teams, will assign

[3]Showing that a strategy has no flaws is equivalent to proving a solution to the game.

a skill level to each of the set of players. Then, given a set of skill level assignments can predict a victor with a probability.

Our goal is to produce a skillful player given only the rules of a game, meaning that we aren't given an example player to compare our answer to. Without solving the game, competition is our only means of measuring quality. It seems that under these circumstances, we must iteratively construct candidate players, have them compete, and estimate their skill based on the results of the competition. Then we can use information about better players to construct yet more skillful candidate players. Genetic algorithms seem well suited to this task as they maintain a population of candidate solutions which can be made to play each other in a tournament in order to obtain win/loss records for rating. The better rated individuals will get a higher probability of reproduction, and their traits will be passed on to future generations. Since the win loss record is obtained from competition within a population, the estimated skill will only apply to that population. We hope that we have several strategies that are good for different reasons (due to different traits), and that when they reproduce, good pieces of several individuals will recombine into an individual that is superior to any of the ancestors. There is also a chance that an individual is copied directly into the next generation without any change. If this happens we expect that it's rating in the new generation will be lower than it's rating in the prior generation since we expect that the level of competition has stepped up slightly.

One way that the search might proceed is by progressively learning the openings of the game through trial and error. Openings of games are studied because for most games, moves toward the beginning are more important. One example of this is that if you've lost by move five, it's irrelevant what you might have done on move seven. If we try to characterize a game in which the initial moves are not as important as later ones, we find that we may as well move randomly for the first several moves as this won't put us at any disadvantage, and in fact there is no advantage to be had by expending

effort to carefully consider the early moves. If there are important moves, after a sequence of unimportant moves, then we can consider that the opening of the game is merely delayed rather than non-existent, though it is hard to envision such a game. At some point though, we expect to find a decision that is worth carefully considering, as different choices will result in different advantages (likelihood of victory) amongst the players. If there are never any moves that give different advantages to the players, then whatever strategy is employed will not affect the outcome of the game.

We start with a population of random strategies, or more precisely, a population of random neural networks that we will use to implicitly define strategies. Since each strategy will have a random evaluation of each first move, the first move selected by each should be random, and every move thereafter should also be random. It seems reasonable to expect that a strategy that has randomly selected a good first move should end up with a higher rating than a strategy that selects a weaker first move. Since the remainder of the game after the first move is made will be random, and a good first move should narrow the game into a branch of the game tree that slightly favors the moving player, it seems like a random player with a better first move should win more games. If a small set of initial moves starts appearing in larger percentages of the population, then the same phenomena may occur for the second move. If the search proceeds only in this manner then we have is a probabilistic tree exploration as opposed to an exhaustive one.

The above has examined how a search may extend good sequences of moves from the root of the game tree, it's also possible that the search may extend good sequences of moves from the leaves of the tree backwards toward the root. In Tic-Tac-Toe, a player that consistently recognizes and selects a state with three of their marks in a row (a victory state) when available, will end the game earlier, and with better consequences than a player that sometimes decides on some other alternative. This kind of development should lead to higher ratings and therefore higher levels of reproduction

and the propagation of this trait. A similar kind of recognition of the potential for three marks in a row for the opponent could be recognized, and the choice that blocks their win could be recognized and selected. If these patterns were mastered, then a similar phenomena may occur for moves slightly earlier in the game, such as forks which are moves that threaten to win along two lines. Only one of these lines can be blocked, which means that the player making the fork will win on their next move. In the same progression, blocking a fork would also be a valuable move.

The hope is that rather than simply extending sequences of moves from the root or from the leaves, that elements of strategies will be drawn from across branches, that will apply to all branches. The depends on the power of genetic algorithms to identify high performance traits and propagate them throughout a population. Learning small patterns that are important in many branches, and at many levels of the tree, and composing the small patterns into larger patterns, even overlapping patterns that influence decisions when they are recognized among choices. These kinds of concepts could could be used when opposing strategies lead to never before seen branches in the game tree (which will be very common for reasonably sized games) and still achieve good results.

Chapter 4

The Search

4.1 Genetic algorithms for search

Genetic algorithms are a search heuristic inspired by the process of evolution in the natural world[9]. Genetic algorithms simulate a population of individuals evolving to adapt to their environment generation by generation. As with natural selection, individuals more adapted to their environment are more likely to have the opportunity to reproduce, while less adapted individuals are less likely to survive. The idea is that individuals have traits, elements of their make-up that determine how well suited they are to their environment, such as large teeth in wolves. When individuals reproduce, their traits are mingled in their offspring, and if strong individuals reproduce the combination of traits in the children may be stronger than that of either parent.

The fact that offspring inherit traits from their parents has been known for millennia, and used extensively in agriculture to breed desirable traits. The set of physical traits that make up an individual are collectively called the individual's phenotype. A relatively new discovery, is that the instructions for growth and development of organisms are in DNA (deoxyribonucleic acid) and are a sequence of four substructures (or bases) abbreviated G, C, A and T. A particular sequence of G's, C's, A's and T's

in a particular location along the series of instructions can correspond to a particular trait, such as blue eyes. That particular location along the instructions is called a gene, and the particular sequence of instructions that results in blue eyes is called an allele for the eye color gene. There are alternate alleles for the eye color gene, such as a sequence of instructions coding for brown eyes. The set of all alleles that make up the instructions for a particular individual are called that individual's genotype. The terminology can get a little confusing with the overlap of word roots between gene and genotype. A phenotype is the set of traits in a developed individual, a gene is a location in the instructions where an allele is specified, and the genotype is the set of alleles that make up the instructions for building an individual. The canonical representation of an individual (a genotype) in a genetic algorithm is a string of binary digits. The set of values in a binary genotype $\{0, 1\}$ is a small finite set, a simpler analog of nature's $\{G, C, A, T\}$.

The reproduction of individuals is modeled in genetic operators which recombine individuals, forming other individuals. The canonical genetic operators, which are inspired by biological processes, are crossover and mutation. Crossover lines up the genotypes (sometimes also called chromosomes) next to each other, randomly selects a point along their length, and produces a new genotype consisting of the first portion of one, and the second portion of the other genotype. Mutation scans the entire length of a genotype, and with a low probability switches the binary digit in each position. In canonical genetic algorithms, after two parents are selected, they are crossed over, and the resulting genotype is then subjected to mutation, both genetic operations are used in sequence.

The natural selection of individuals is facilitated by a function which maps an individual to a "fitness" value. This fitness function is the simulation's representation of the environment. Since fitness is determined based on the phenotype, if the phenotype is not the same as the genotype, then the genotype must be transformed

into the phenotype before fitness can be evaluated. An individual with a higher fitness value gets a higher probability of being selected to bear offspring. In canonical genetic algorithms, the probability of selection for reproduction is proportional to the fitness value, so an individual with a fitness value of 100 would be twice as likely to be selected as an individual with a fitness value of 50.

The final, and most loosely defined aspect of genetic algorithms is the higher level plan called the reproductive plan. It specifies things like when a chromosome is selected for reproduction, and when a chromosome is selected for deletion. An example of a reproductive plan would be: [11]

- Form an intermediate population by selecting individuals to move on to the next generation with probability proportional to their fitness. Some individuals will not make it to the next population, some will receive multiple copies.
- Randomly select individuals from the intermediate population to reproduce. Each member of the intermediate population is equally likely to join the pool of parents.
- Form couples from the parent pool randomly, using each member of the pool only once.
- Perform the crossover twice to produce two children for each pair of parents and replace the instances of the parents with the children in the intermediate population.
- The final step in constructing the next population is to execute the mutation operator on each member of the intermediate population, including on chromosomes that are not new children and weren't selected as parents.

The theory behind genetic algorithms depends on traits (corresponding to alleles), which may be present or absent, contributing to the fitness of an individual. Such traits are specified by subsets of the chromosome called schema. A schema specifies a value for some positions in the chromosome, and uses a wildcard for other positions

(i.e. if '*' is the wildcard then '**10*', '*0*1*', and '1111*' would be schemata). Any chromosome that has the values in the positions specified by the schema matches the schema (i.e. the schema '*0*1*' is matched by both '00010' and '10111')[1]. If you were to produce all chromosomes that match a given schema, their fitness values would form the fitness distribution of that schema. Rather than producing all chromosomes that match a schema, we consider that all chromosomes in a population that match the schema represent a sample of the schema's fitness distribution.

There are two properties of schemata that permit an analysis of how genetic algorithms work, these are the *order* and *defining length* of the schema[11]. The *order* of the schema is the number of positions that it specifies, and the *defining length* is the difference between the first and last specified positions (i.e. the order of '*0**1*1**' is 3, and it's defining length is 7 - 2 = 5). The analysis involves the probabilities of schemata being disrupted by the genetic operators, and tracking the expected number of instances of a schema (the number of chromosomes that match a schema) from generation to generation.

If we take the set of chromosomes from a population that match a schema, to be a sample from the fitness distribution of that schema, we can use the mean fitness of those matching chromosomes as an estimate of the fitness of the schema. If we eliminate (for the moment) the genetic operations, and only include fitness proportional selection, we can show that schemata whose estimated fitness remained a constant proportion above the average fitness of the population will get an exponential increase in the number of matching chromosomes as generations proceed. The only counteracting forces are the disruptions caused by crossover and mutation. The larger the order of the schema (the more specified positions), the more likely it is to be disrupted by mutation, though mutation is very low probability. The longer the

[1] Note that in this model all chromosomes and schema are the same length. It's not so much a matter of a schema *not matching* a chromosome of a different length, it's more that they are *incompatible*.

defining length of the schema, the more likely it is to be disrupted by a crossover operation, when the crossover point is somewhere along the schema. These genetic operations must be included as they are the exploratory force of the algorithm, with selection alone, no new candidate solutions would be found. The balance among these factors makes schema which are short, low order and high fitness the driving force behind the search. These schema are likely to give a boost in fitness to offspring who inherit them, and are more likely to be realized in an offspring than longer, high order schema which may be disrupted.

4.2 The genetic algorithms framework

The implementation of the genetic algorithms framework was made to be extensible. Components defining pieces of behavior must be supplied to the framework for it to function. Several implementations of these components are provided, and new ones may be easily written. The basic algorithm only specifies an iterative nature in which the population repeatedly "evolves" into a new population, at each iteration a termination condition is checked to see if evolution should cease. The termination condition is one of the components that must be provided. A termination condition which terminates the algorithm after a specified number of generations is provided, but a terminator based on some measure of convergence, or lack of improvement over time could be implemented.

The evolution of a population into a new population corresponds to the reproductive plan, and must be provided with several components in order to function. An effort was made to make the evolution as general as possible, but there are such a multitude of ways of implementing genetic algorithms that there may be no generalizable portion. The evolution itself will likely need to become a component in the future. The framework's model of evolution must be supplied with several values

and components. All of these values and components may be specified as functions which can return different values when called at different times. The additional level of indirection provided by using functions to supply these elements allows the algorithm to vary dynamically from generation to generation. For example, the evolution requires that a number of children be supplied. This value can be the result of a function which can return different values when called at different times, allowing the number of children produced to change from generation to generation. The values and components that must be provided are:

- The number of children to be produced this generation.
- A function that returns a set of genetic operators that will be used for producing children.
- A method of selecting parents from the population.
- The size of the next population.
- A flag that can give children special status which always allows them to survive to the next generation.
- A method of selecting survivors for the next generation.

A genetic operator used within the framework must specify the number of parents it requires and the number of children it produces. Given the number of children to be produced, and a probability distribution over operations to be used, the framework can determine which operations to use, and how many parents are required to execute those operations. Once the number of parents needed is known the method for selecting parents can be used, and the operations can be executed. Once the children have been produced, the next generation must be produced, of the given size. If all children are to survive, then they are added to the next population, then using the survivor selection method on the previous population, the remainder of the next population is filled in.

There is a utility class provided with the framework which implements stochastic universal sampling[13], which minimizes error in expected value of multiple random selections. When sampling a distribution, say randomly selecting two elements from a distribution of five equally likely elements, there is a chance that the sample will not fairly represent the underlying distribution. For example there is a probability that the same element is selected twice, which is less representative than if two different elements were selected. Another case may be selecting twenty-five elements from the same set of five elements. The expected result would be to end up with each element five times, though this is unlikely when selecting the elements according to probability one at a time. Stochastic universal sampling minimizes this error by selecting the elements from the distribution all at once, using a single random value. When selecting twenty-five from five above, stochastic universal sampling will always produce the expected result of five copies of each element. This utility is available for use in all three of the selection methods that must be provided to the evolution, operator selection, parent selection and survivor selection.

For the canonical binary chromosome representation, genetic operators provided by the framework include mutation, one-point and two-point crossover, as well as the standard operator which performs both one-point crossover and mutation atomically. For the selectors, the framework provides implementations of the canonical fitness proportional selection, the more practical rank proportional fitness, maximum fitness selection (for selecting the most fit n), and random uniform selection. Fitness must obviously be provided by client code, but a wrapper is provided that will cache these values in case they are expensive to evaluate. If the fitness function more naturally operates on real numbers than binary, a class is provided that interprets sections of binary strings as gray codes[2] and translates them to integers, and in turn to real

[2] A Gray code specifies an ordering on n-bit binary strings. That is, for the 2^n binary strings of length n, a Gray code specifies which string follows each string, in fact Gray codes form a cycle. Using this ordering, a correspondence with the integers can be established. Gray codes are often used in genetic algorithms because they have the property that two adjacent values differ only by

numbers, given a minimum value, a maximum value, and a decimal precision (number of decimal places). This mapping will also report the number of binary digits required to cover a range and precision of values.

4.3 Tic-Tac-Toe as a test case

Once most of the infrastructure was in place, the board games framework, the simple neural network, and the binary genetic algorithms framework, Tic-Tac-Toe was chosen as the first game implemented. This was prior to implementing any rating system. Tic-Tac-Toe allows us to use an independent measure of fitness, rather than a measurement relative to the rest of a population of players. Since Tic-Tac-Toe is solved, and it is known that with perfect play that all games end in draws, we know that if a player never loses then that player is in some sense optimal. There are shades of grey among "optimal" players by this definition, there could be differentiation based on the number of times they win. However, minimizing the number of losses seems the least stringent measurement, leading to the largest number of "optimal" players, making such players easier for the system to find.

Tic-Tac-Toe being a small game, we can measure the number of losses for a player by playing them through all possible games, that is, fully exploring the subgraph of the game tree defined by the player's strategy. If we count up the number of losses at terminal nodes in the strategy tree, we have our measure of a players quality. In bigger games such as Chess, a similar measurement cannot be made. It should be easier to find an optimal player using an independent measure of fitness since all of the dynamics of the population are absent. Since we have the option with this initial test case, it was decided to search with independent fitness as a step along the way.

The role of the board games framework is to play a set of competitors through a

a single bit. This means that when a binary string is interpreted as an integer, the mutation of a single bit is more likely to produce only a small change rather than a large jump.

game to it's conclusion. It can be used to run a tournament to generate a win/loss record for each player so that player can be ranked among the population. The Tic-Tac-Toe specific code that plugs into the board games framework was written even though we were not yet using relative fitness within the population, but independent fitness, the number of losses. Parts of the board game framework were used when building the piece of the fitness evaluator that, given a player, would play exhaustively and count that player's losses. The only Tic-Tac-Toe specific code necessary is a class for the state of the game, a rules class to manage state transitions and administrative issues such as termination, and a sensor that would enable evolving players to "see" game states, and enact their strategy. The genetic algorithms framework operates in terms of binary chromosomes, and the board games framework operates in terms of players that, when presented with a game state, respond with an action to take. Recall the discussion of genotypes and phenotypes. The genotype is the binary chromosome which is manipulated by genetic operations, and the phenotype is the player that plugs into the board games framework whose fitness can be evaluated.

The `SimpleMaxPlayer` plugs into the board games framework, is defined by a function that maps a game state to a numerical value. The `SimpleMaxPlayer` evaluates all choices, and returns as it's decision the choice with the highest numerical value. We have chosen to use a neural network for our function, and that is what is represented by each of our binary chromosomes. Since we have chosen to use a "complete feed forward" neural network, once the number of neurons is determined all of our neural networks have the same topological structure, including the same number of weights. Simply by altering the weights in the neural network, we can change the function that the network models. Given the number of inputs, and the total number of neurons in the network, our simple neural network implementation can calculate the number of weights needed (one for each connection between neurons). It can also accept a list of the appropriate number of values and distribute

those values idempotently as weights in the network, meaning that an identical list of values will produce an identical network. We can use the facilities of the genetic algorithm framework to take a binary string in sections, and translate each section into a real number. This translation of a binary string into a real valued string gives us what we need to construct a neural network. The remaining concern is that we get a string of the appropriate length. Once the mapping from binary string to real valued string is defined, the classes that perform the mapping will also tell us the length that the binary string must be (the length of our chromosome).

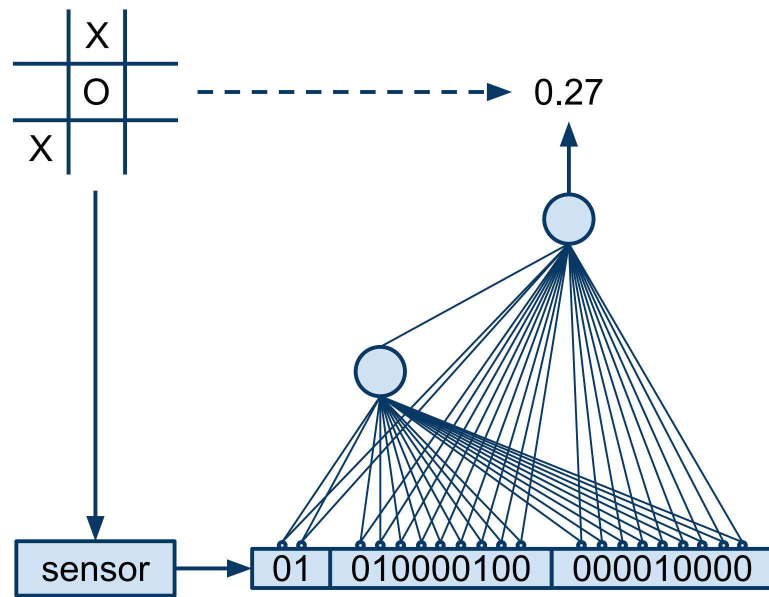

Figure 4.1: This diagram shows a Tic-Tac-Toe state being mapped to a numerical value. First the sensor breaks down the state into a binary description, then this binary string is fed into a complete feed forward neural network (of two nodes), which outputs a number. The number is interpreted by the system as the player's evaluation of the state. The sensor and neural network are wrapped in a SimpleMaxPlayer which, when given a set of states, returns the one that evaluates to the highest value.

The Tic-Tac-Toe sensor maps a game state to a 20-bit string. The first bit answers the question "is it X's turn to move?", and the second bit answers the question "is it O's turn to move". There is apparent redundancy here in the sense that one bit is the negation of the other, it can only be one player's turn, exactly one of the two

bits will be set. However, for a terminal state, one in which the game has ended, neither bit will be set, and in addition, in the spirit of being general, games with 3 or more players and games with simultaneous turns cannot represent whose turn it is with a single bit. The remaining 18 bits of our 20-bit string are filled in using an arbitrary sequence for the nine boxes on the board. The first 9 answer the question "is there an X in this box?", and the second 9 answer "is there an O in this box?". Once again there seems to be some redundancy, but in this case, an empty box must be represented as well as one with either mark in it. If each bit is to correspond to a single box, meaning that each box will have a dedicated set of bits then the number of bits per box must be at least 2, to represent the 3 possible states (and the fourth will never occur, both X and O in the same box).

Now we are able to take our binary chromosome, and turn it into a real valued string, turn that into a neural network, attach a game state sensor, wrap it all in a `SimpleMaxPlayer`, and we have a transformation from a genotype into a phenotype, a player capable of playing Tic-Tac-Toe within the board games framework. For the moment we will not use the full functionality of the board games framework though, we will use portions of it to exhaustively test a player and count the number of times they lose.

While all the components are tucked away in their various frameworks, there are still many parameters that can be used to adjust the search. One parameter in particular, the size of the neural network, has an exponential effect on the size of the search space, since adding a single node adds a connection with a weight between it and every other node in the network. The idea was to find an optimal Tic-Tac-Toe player by minimizing the number of losses, starting with 1 neuron, and stepping up by 1 until an optimal player was found. Once an optimal player was found using an independent fitness measure, we would have a value for the number of neurons that is known to be capable of producing an optimal player. It may be that an

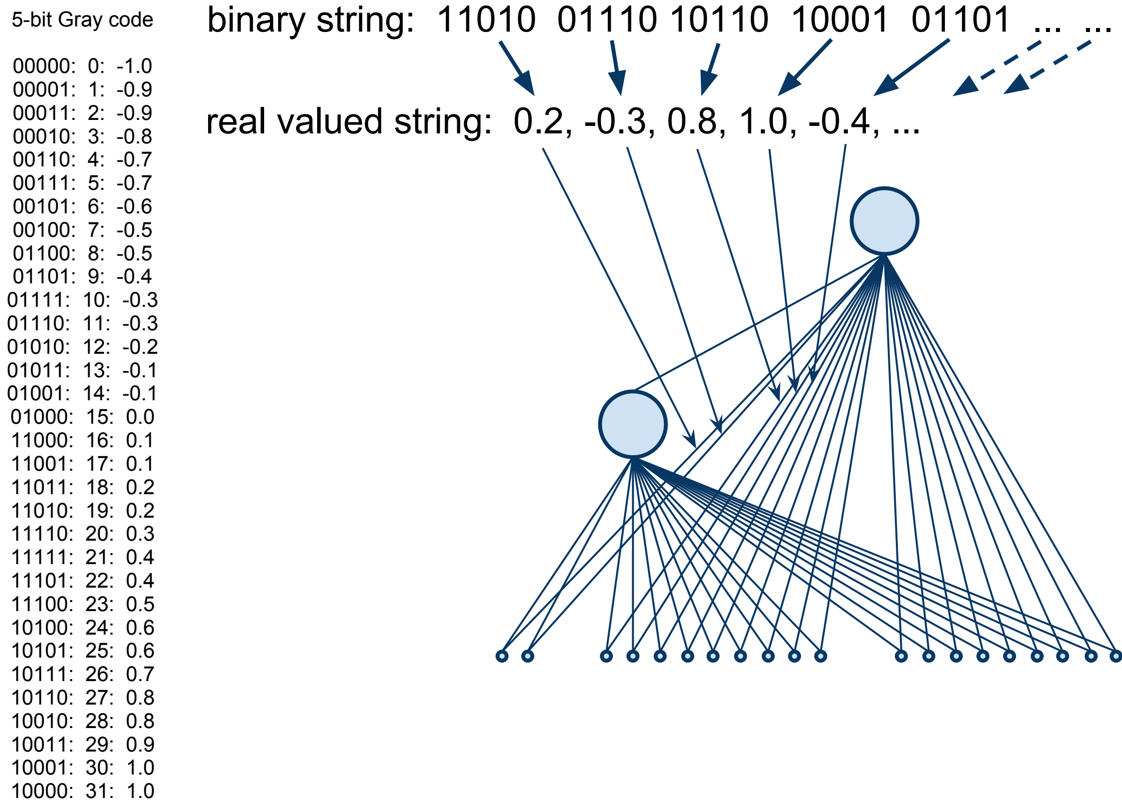

Figure 4.2: Allowing our weights to range from -1.0 to 1.0 with a precision of 1 decimal place requires 21 distinct values. A 4-bit Gray code supplies only 16, so a 5-bit code, which supplies 32, must be used. The binary string is sliced into substrings of length 5, then the Gray code is applied to obtain the real valued string. The weights on the edges in our neural network are then taken directly, in sequence, from the real valued string. The lower node in the pictured network has 20 inputs, and the higher has 21, so a binary chromosome that could fully define the weights of this network would need to be 5 * (20 + 21) = 205 binary digits long.

optimal player could be produced with fewer neurons, but the desire was to quickly get a value that was known to be feasible, so that testing the system with a relative measure of fitness could commence. Of course, the optimality of the player produced with relative fitness would still be verified. An assumption was made that if there was hope for the system to produce competent player of a larger game like Chess, it would have to be able to easily produce an optimal player of a small game like Tic-Tac-Toe.

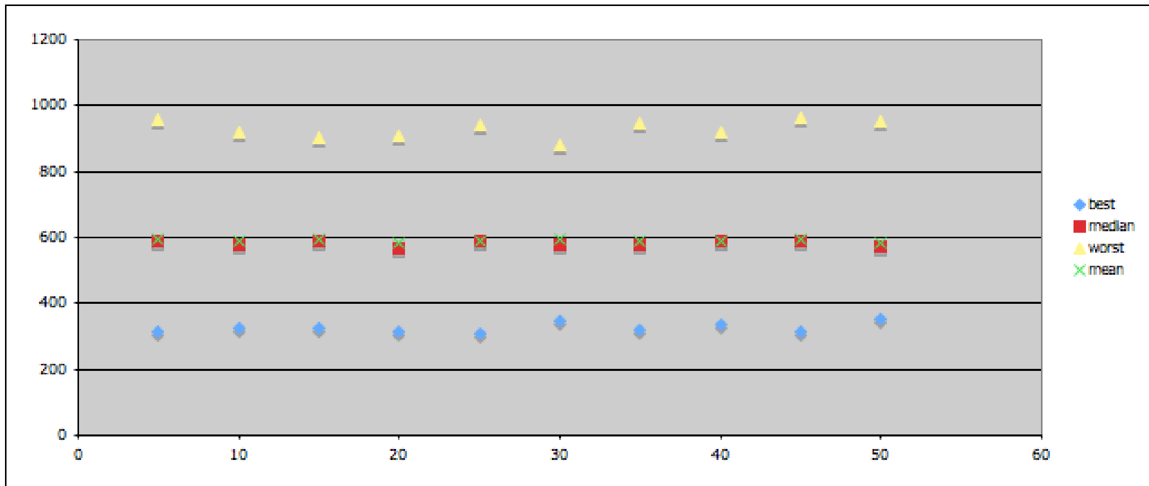

Figure 4.3: A plot of the data collected to analyze samples of random binary chromosome players with different numbers of neurons. The horizontal axis is the number of neurons, the vertical axis is the number of losses. For each value for the number of neurons, a sample of 500 random chromosomes was taken. It can be seen that none of the statistics varied by much, the bottom row of points represents the best of each sample, the top represents the worst, the squares in the center the median, and the x's in the center the mean.

No specialized components have been written for the genetic algorithm framework. The instantiations of the standard framework components, corresponding to the reproductive plan used is characterized in the following description. The initial population will be constructed randomly, and the size of the population will remain constant from generation to generation. At each generation half the population will be replaced by new children. Parents will be selected with probability proportionate to their rank in the population, and they will be recombined using the standard operator which performs one-point crossover and then mutation on the offspring. The remaining half of the population will be filled in by the most fit of the previous population. Even with all these decisions made, there are several numeric parameters that remain tweakable, the most notable being the number of neurons, the size of the population, and the number of generations the algorithm should run. Additional important numeric parameters include those defining the range and precision of the

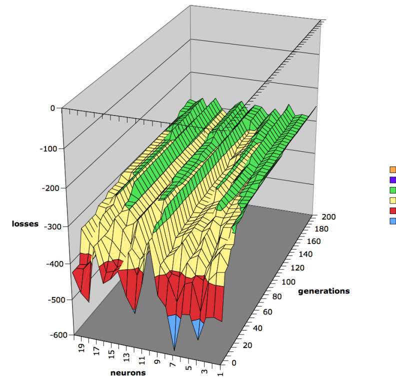

Figure 4.4: A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 200, and population size remained at 5.

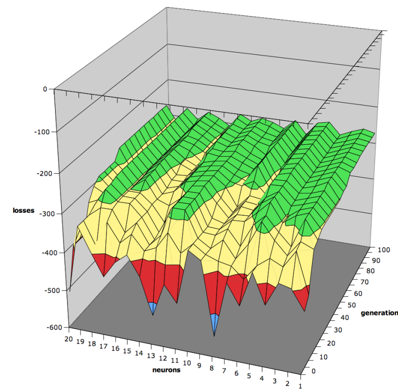

Figure 4.5: A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 10.

weights in the neural network as translated from the binary chromosome, and the probability of mutation in the standard operator. The selected range for weights is from -1.0 up to 1.0, with a precision of 1 decimal place, and the probability of mutation is 0.03.

It was the thought that ad hoc adjustment of the remaining parameters, number of neurons, size of population, and number of generations, would yield optimal players as Tic-Tac-Toe is such a small game, but this did not prove to be the case. After many unsuccessful attempts, a more thorough attempt to understanding how the

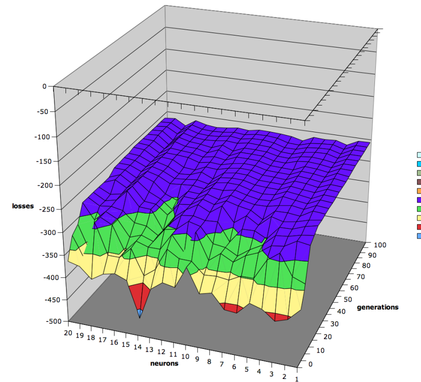

Figure 4.6: A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 40.

parameters affect the performance of the algorithm was made. As a baseline, statistics were taken on samples of random chromosomes with different numbers of neurons to see if there was any kind of structural bias, none was found. A plot of the data collected appears below, each sample of 500 random chromosomes corresponds to column of data points.

Next a fine grained look was taken at the number of neurons and the number of generations, since 4D graphs are difficult to make. Three plots appear in figures 4.4, 4.5, and 4.6, the first for population size 5, the next for 10, and the third for 40. Each of these plots shows the progression of the genetic algorithm for different numbers of neurons ranging from 1 to 20. Every 5 generations a data point was taken, the best of the population, since if the algorithm terminated at that point, the best would be the solution used. The population size 5 graph is still a little ragged at the 100th generation, so that one proceeds to 200, but it seems obvious that all three have leveled out. Having observed that regardless of the number of neurons, that the algorithm ceases to improve from around 250 losses by the 200th generation, a plot was made of neurons versus population size, at a constant number of generations of 200. The graph is missing a large portion of the far quadrant, it took several days to

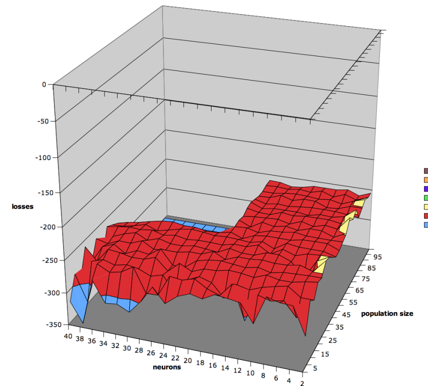

Figure 4.7: A plot of the number of losses for networks constructed from binary chromosomes with numbers of neurons ranging from 2 to 40, as population increased from 5 to 100, and the number of generations was always 200. The missing quadrant was to computationally intensive to compute.

produce the data, and each point attempted in that quadrant did not finish in less than a day. It seems obvious from the data present that nothing much would change in that quadrant. It seems clear that this attempt has failed, and that something else must be tried.

4.4 Real valued genetic algorithms

Since the each weight in the neural network is ultimately determined by a specific contiguous substring of positions in the chromosome, it's hard to imagine that many useful schema can form that don't fall on the same boundaries as these substrings. Let's focus on a single weight from the network, and the corresponding substring of the chromosome. If a crossover occurs, and the crossover point is somewhere in the middle of that substring, what can we say about the substring that will be inherited by the child? We know that part of the substring will come from one parent, and the remainder from the other parent. The only way that the child's substring will match one of the parents is if both parents had identical substrings. When a substring is split, it's context is lost, and it seems likely that recombining two split substrings into

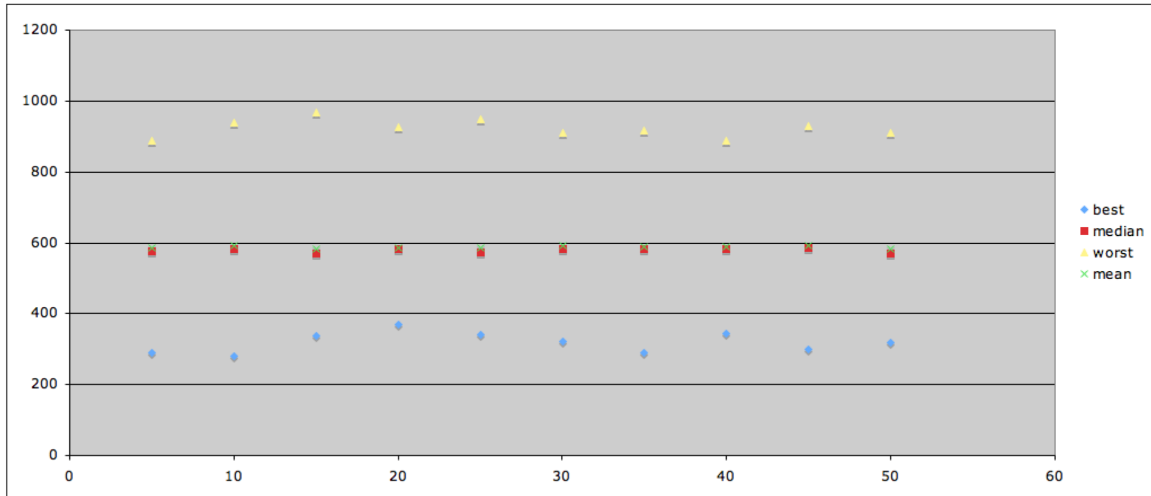

Figure 4.8: A plot of the data collected to analyze samples of random real chromosome players with different numbers of neurons. The horizontal axis is the number of neurons, the vertical axis is the number of losses. For each value for the number of neurons, a sample of 500 random chromosomes was taken. It can be seen that none of the statistics varied by much, the bottom row of points represents the best of each sample, the top represents the worst, the squares in the center the median, and the x's in the center the mean.

a new substring will produce an essentially random weight in the child. In addition, the intention of a mutation operator is to ensure that within a finite number of steps any point in the search space can be reached[12], it is intended to produce only small changes. Even with the use of Gray codes, which increase the likelihood of a mutation causing only small change in a weight, a change could still be large and disruptive. After the failure with binary chromosomes, it was surmised that it may be due to disruptions like these.

It was determined that an alternate representation should be tried, real valued chromosomes. Another possible deficiencies in the binary chromosome attempt that this should ameliorate are the cost of additional precision, and the potential disruption of crossover and mutation. To gain a single decimal place of precision in a binary representation takes at least three, and sometimes four additional bits per neural network weight. The change from the range of -1.0 to 1.0 with 1 digit precision to

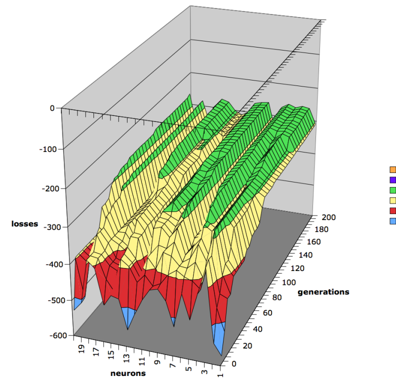

Figure 4.9: A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 200, and population size remained at 5.

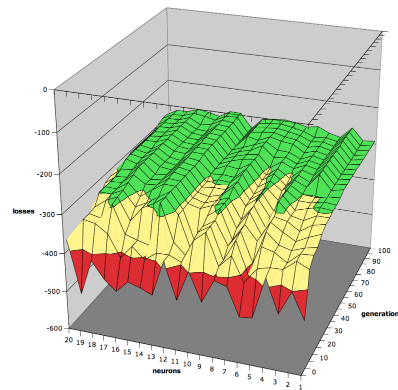

Figure 4.10: A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 10.

the range -10.00 to 10.00 with 2 digits of precision is a jump from 5 bits to to 11 bits for each of the large number of weights required to populate a complete feed forward neural network. If more precision in the weights is needed to optimize a neural network, then a substring of the chromosome corresponding to a weight in the network will need to be made longer and longer. This seems counterproductive if genetic algorithms depend on short schema. With real valued chromosomes we get infinite precision "for free". Using real valued chromosomes may solve these problems.

One nice thing about real valued chromosomes is that, with Gray codes eliminated,

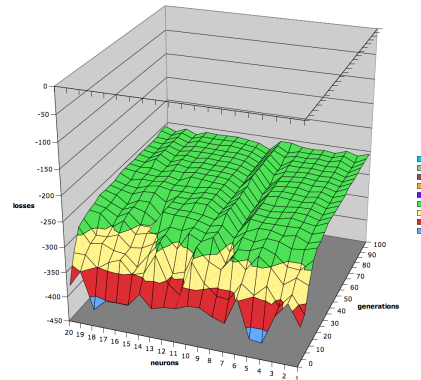

Figure 4.11: A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 1 to 20, as generations increased from 0 to 100, and population size remained at 40.

the translation step is greatly simplified. Since we have infinite precision available (full precision of the datatype used), we model each position in the chromosome as a real value from 0.0 to 1.0 so that simple general purpose operators can manipulate them. To transform to the phenotypic value, we can multiply by the range, and add the minimum value. A new set of operations is necessary with real chromosomes, but other than that the genetic algorithms framework functions unchanged. The new operators implemented were creep mutation, two-point crossover, and LX[14] recombination (a blending operator). Creep mutation selects positions to mutate with low probability, as with binary mutation, it randomly adds or subtracts a small configured value to each selected position. Two-point crossover is similar to one-point crossover, but two crossing points are selected, and the portions of the chromosomes that lie between the crossing points are swapped. With two-point crossover traits from the parents pass directly to offspring as with one-point crossover. With real valued chromosomes there is another option, positions from two parents can be "blended". One method of blending would be, for each position, for the offspring to receive the average of the values of the parents. A simple average can have detrimental affects on the population though, so LX crossover was used. LX crossover selects a

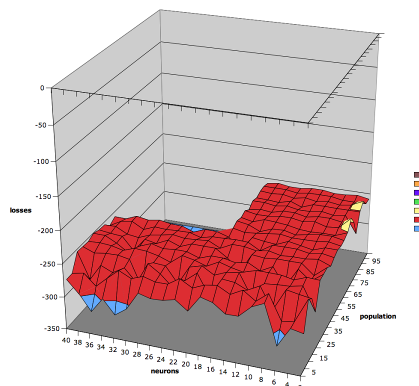

Figure 4.12: A plot of the number of losses for networks constructed from real chromosomes with numbers of neurons ranging from 2 to 40, as population increased from 5 to 100, and the number of generations was always 200. The missing quadrant was to computationally intensive to compute.

random number between the parent values. Each of these operations was given equal likelihood when recombining two parents.

As can be seen, the results are eerily similar. It may be that training a neural network to play perfect Tic-Tac-Toe is simply much more difficult that at first thought, or there could be something fundamentally wrong with the approach using genetic algorithms. Either way, if changes to the parameters are not enough, and more drastic changes to the reproductive plan are called for, it seems wise to eliminate at least one variable from the equation. If by whatever means necessary we can construct a neural network that can play perfect Tic-Tac-Toe, then we can come back to genetic algorithms with that knowledge.

4.5 Backpropagation in neural networks

Backpropagation of error is a traditional way of training neural networks when the function it is to model is known. If we construct a perfect player of Tic-Tac-Toe through brute force, then we can form a function mapping game states to real numbers that will enact the perfect player's strategy. If we collect all possible choices that the

player could ever be presented with and then isolate the optimal choices the player decides to make, we can map the optimal choices to the value 1.0, and all other choices to the value 0.0. This function, when plugged into a `SimpleMaxPlayer` would play identically to the perfect player. Once we have these mappings, we can train a neural network using backpropagation to model this function. This is not a player produced from the rules alone, but rather a player mimicking a perfect player which was produced through brute force. Our purpose here though, is to determine how many artificial neurons are required to model a perfect player.

Backpropagation seeks to minimize the sum of the squared error at the output nodes of the network. Since we have target output value for a given input, we can measure the error in the network. It makes sense that we could adjust the weights of the connections coming into the output node to reduce the error, and the backpropagation algorithm does just that, but it goes farther. Data flows through the network from the input nodes along the weighted edges into other nodes. A node receiving data sums all of it's inputs and processes the sum through it's activation function to produce it's output value. This process continues until values are determined for the output nodes of the network. Backpropagation calculates the derivatives with respect to the sum of squared error not only for the output nodes, but for the output of every node, then backward through the activation function for the input to that node, then to the weights on the incoming connections to that node. Once the derivatives of the weights are known, each weight is decreased by an amount proportional to it's derivative, this proportion is called the "learning rate". The derivative of a weight with respect to the sum of squared error tells us which direction the weight needs to be adjusted in order to decrease the error. Weights with negative derivatives should be increased and weights with positive derivatives should be decreased in order to decrease the error. More than just identifying the direction of the shift, the derivatives tell us which weights will affect the error most, so we shift those weights more,

proportionate to their derivatives.

Once implemented the backpropagation algorithm was put to the test learning first the unary logical functions, IDENTITY and NOT, then the binary functions, AND, OR, XOR, and IMPLIES. Networks of 3 nodes were able to learn all of these within 100 training cycles of the backpropagation algorithm. A more complicated function was tested next, a period from the sine function which, in contrast with the logical functions, incorporated real values as well as many data points. The sine wave was learned within 50,000 training cycles by a network with only 3 nodes.

Tic-Tac-Toe of course is much more complex than a sine wave, and in order to make it as simple as possible for the network to learn, some steps were taken. The first was to eliminate symmetries. The 4 possible rotations of a Tic-Tac-Toe board have no effect on the meaning of the board, or the outcome of the game. Reflection of the board, similarly, has no effect, leaving us able to replace up to 8 boards with a single representative orientation. Brute force was used on the game with symmetries removed, and some simple heuristics were used to reduce the number of boards encountered a bit more. Taking a winning move when one is available, for example, can prune a few unnecessary boards from a branch. When all was said and done, a player playing with the constructed strategy will only ever encounter 319 boards, 72 of which should be rejected, and 247 of which should be accepted. It's interesting to note that while backpropagation reduces the error monotonically, the number of losses by the Tic-Tac-Toe player occasionally increases. If the network were able to map all boards to just the correct side of 0.5 (toward 0.0, or toward 1.0), the player would play optimally. However, even if backpropagation were given an optimal player to start with, it may immediately make a few of the mappings exact which reduces the error by a great deal, and as a sacrifice to accomplish this, move several boards to just the wrong side of 0.5, causing the player to play terribly.

Though a perfect player was found with 25 nodes after about 21,000 training

cycles, the result was transitory, within 200 training cycles the error in the network decreased from 8.53 to 8.29 and the player had gained 3 losses. The result can only be repeated very rarely. This lack of precise correspondence between decrease in error and decrease in losses is problematic. Even with the simplifications made to the game of Tic-Tac-Toe it was difficult for backpropagation to learn to the function of an optimal player. Much more work is obviously needed if the system is to produce optimal Tic-Tac-Toe players as a matter of course.

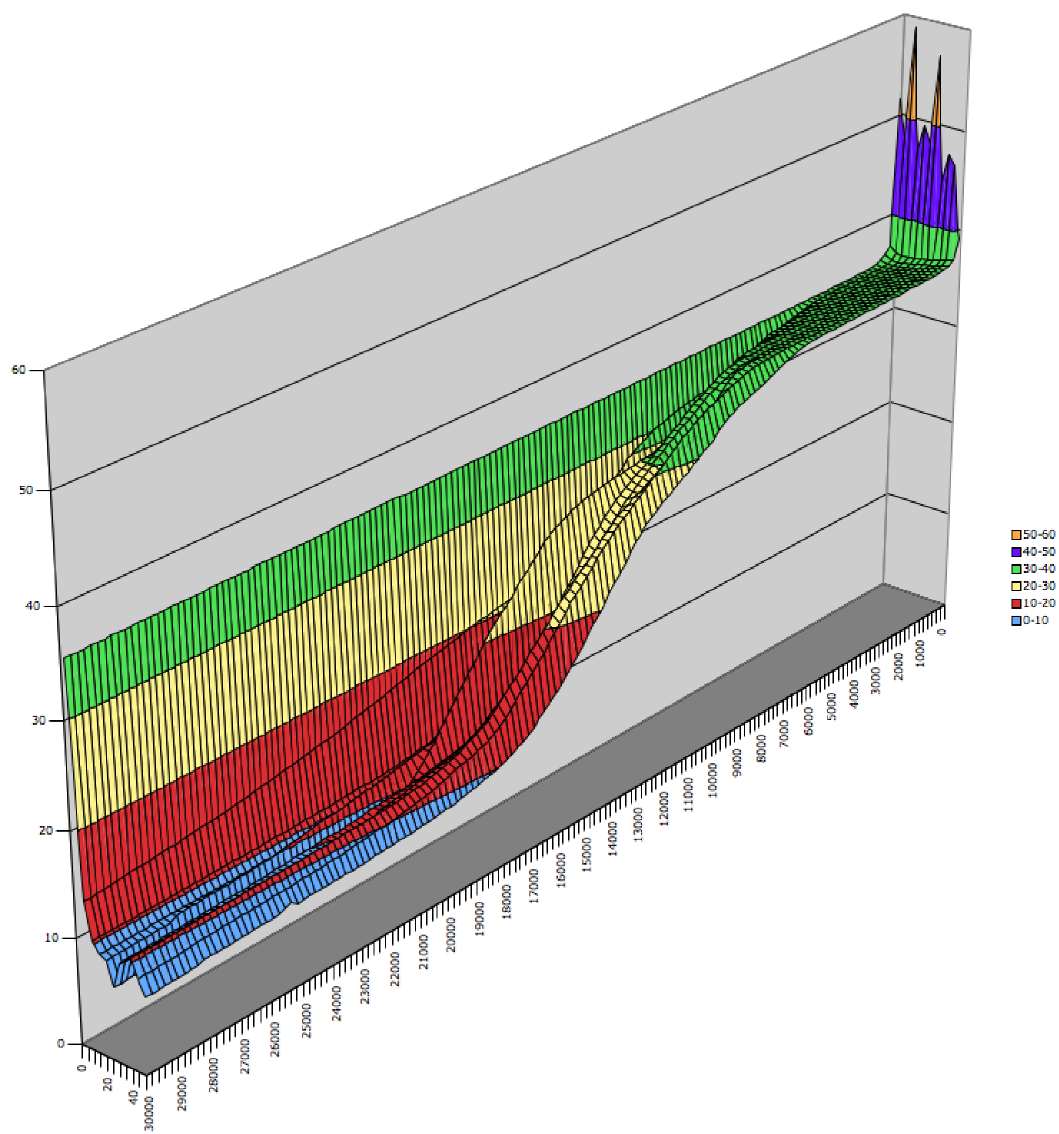

Figure 4.13: This is a plot showing the error reduction by backpropagation for sample networks with from 5 to 40 nodes, with a learning rate of 0.005, through 30,000 training cycles.

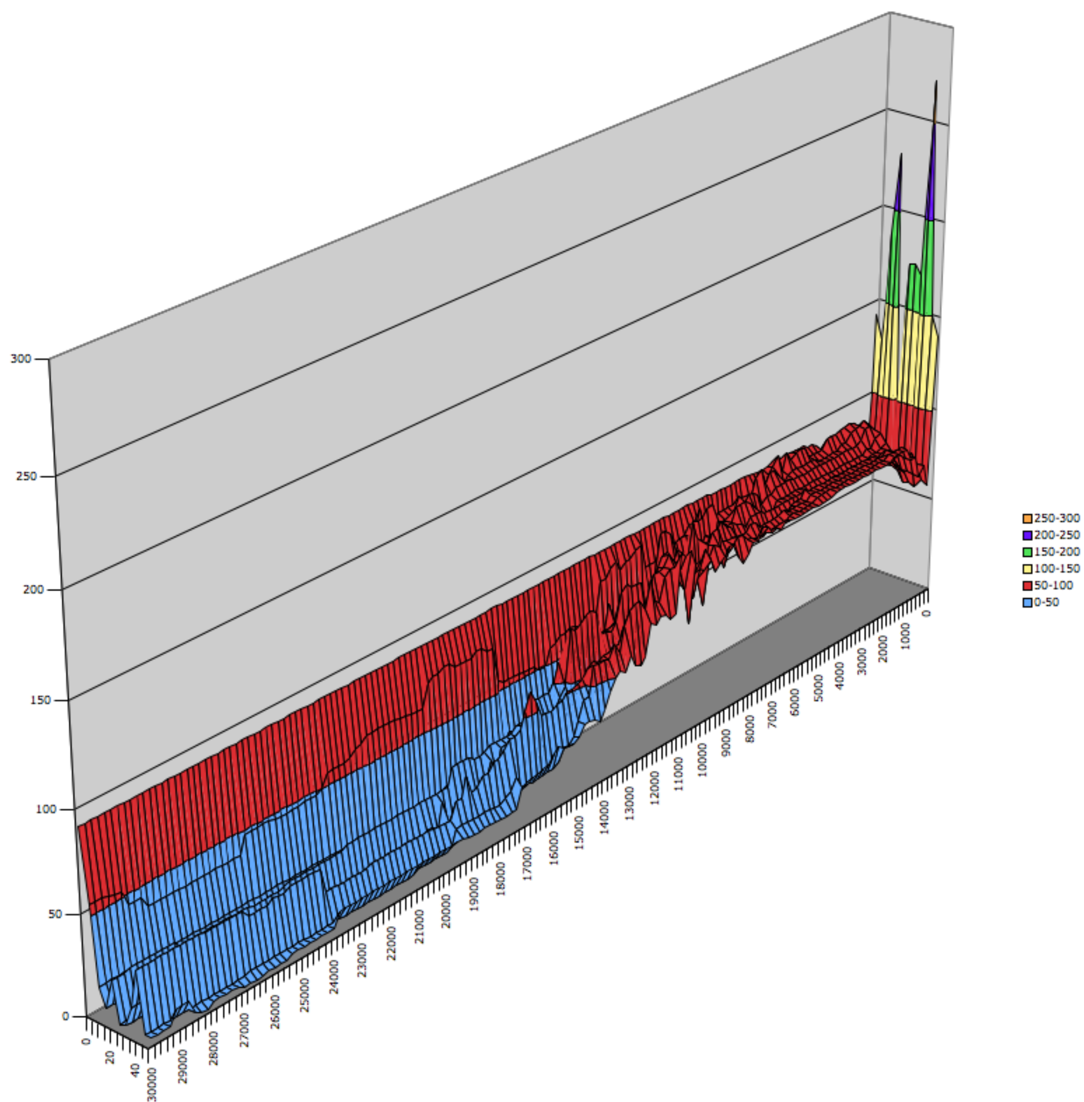

Figure 4.14: This is a plot showing the number of losses for sample networks with from 5 to 40 nodes as the error was reduced by backpropagation through 30,000 training cycles.

Chapter 5

Next steps

One possible way forward would be to question the assumption that if the system is to produce good players of complex games like Chess that it must easily produce optimal players of simple games like Tic-Tac-Toe. Due to the unexpected difficulty in producing optimal Tic-Tac-Toe players no other games were attempted. This is doubtful though. Another option would be to abandon the abandonment of tree search. It's almost certain that the hybrid of an evaluation function evolved with the system, used as the heuristic in an alph-beta tree search would outperform the evaluation function alone.

There are many avenues that may be pursued to improve both the neural network and genetic algorithm frameworks used in this work. The frameworks are not state of the art. It was thought that the problem could be attacked with a more basic toolkit, but this is apparently not the case. The frameworks have been written with the plan for them to become state of the art in mind though, they are as flexible and extensible as possible, and new functionality will be added in the future.

Specific to the neural network, it is common for a neural network implementation to include a bias node[8], an additional input node which is always set to 1.0. This allows backpropagation, or whatever weight optimizing mechanism (perhaps genetic

algorithms) to add constants to the functions that they represent. The net stimulus into a node is the sum of weights multiplied by the input values, including a particular weight multiplied by an input value that is always 1.0, the bias. This allows that weight to be learned as a constant value in the sum which has an effect on the output value of the node.

Rather than simply mapping of optimal boards to 1.0 and rejected boards to 0.0 and trying to minimize the error, a better method of classification could be used. Using cross entropy[21] instead of sum of squared error is supposed to keep backpropagation from doing so much work to reduce error by getting some values very close to the 1.0 or the 0.0, and instead get all values categorized properly. Also, instead of having a global learning rate for all weights in the network, making the learning rates specific to each weight can improve performance[20].

Another, constructive approach to optimizing neural networks is called cascade correlation[19]. Using cascade correlation, the network is trained up to a certain point, then a node is trained to the error function, and added to the network, then the network is retrained together. This goes on until the error is low enough, repeatedly adding nodes trained to the error, to try to eliminate the error. Aside from networks with changing topologies such as this, experiments could be done with simple, single hidden layer feed forward networks instead of the complete feed forward networks.

There are other options besides genetic algorithms for training neural networks, some of these, such as Q learning could be explored. Q learning allows you to delay reinforcement, so that a series of evaluations can be performed, and the result of the series can be used to train the network. This could apply directly to making all the decisions to play a game, and training the network based on whether it won or lost.

On the genetic algorithms front, a thorough analysis of the mutation probabilities could be done, as well as the range and precision values used, though it seems unlikely that either of those would be the key to the problems experienced. One hopeful

candidate would be to use inversion[9] which is supposed to allow chromosomes to reorganize themselves in order to build better schema. With the static layout, since short, low order schema are the driving force of the search a lot may depend on the static layout selected. It would be nice if the algorithm could find an intelligent layout on it's own.

There are a plethora of genetic operators to choose from, for example, uniform crossover[16], which is antithetical to the schema theorem has had success. It is essentially n-point crossover where n is the length of the chromosome. Using uniform crossover each position along the chromosome can be probabilistically swapped independently from any other position. Many other operators could also be explored. Even larger changes to the reproductive plan could be made, one possibility would be using some measure of the diversity of the population, and having the probability of mutation depend on this measure. As diversity went down, the probability of mutation could go up, to produce more diversity.

There is a genetic algorith out there called CMA-ES[15] which stands for Covariance Matrix Adaptation - Evolutionary Strategy. It is the current champion of a suite of function optimization functions, and is supposed to be good for very difficult problems. There is also another kind of genetic algorithm that could be tried called a PMBGA, a Probabilistic Model Building Genetic Algorithm. Instead of using genetic operations, it is purely statistical. It samples the population and builds a probabilistic model from the results. Then to produce the next generation, it samples the probabilistic model it has built. As the generations go by, it not only builds models of the values in certain positions but the relationships between values in different positions. It seemed very interesting.

Another alternative to attempting to optimize a neural network, could be to use some other kind of function. Genetic programming is a genetic algorithm that runs on tree structures. These tree structures can be interpreted as formulae that can be

evaluated, such as parse trees for mathematical expressions.

Finally, toward the very end of this work, some related work was found, called GGP, General Game Playing[22]. That work includes a domain specific language called GDL, Game Description Language that can describe a large class of games. Recently GDL-II[23] was created which added hidden information and chance to GDL. Several automated game players have been created to play games written in GDL, it's certain there is a lot to be learned from them. It would be great to write a parser that could implement a game written in GDL within the board game framework, and interact with that community.

Bibliography

[1] Odersky, Spoon and Venners, *Programming in Scala*. Artima, 2008.

[2] Shannon, C., *Programming a Computer for Playing Chess*. Philisophical Magazine Ser. 7, Vol 41, No. 314 1950.

[3] Glickman M., *A Comprehensive Guide to Chess Ratings* American Chess Journal 3 1995.

[4] Herbrich R., Minka T., and Graepel T., *TrueSkill(TM): A Bayesian Skill Rating System* Advances in Neural Information Processing Systems 20 2007.

[5] Shoham and Leyton-Brown, *Multiagent Systems Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[6] Nash J., *Equilibrium Points in n-Person Games* Proceedings of the National Academy of Sciences of the United States of America, Vol. 36 No. 1. 1950

[7] von Neumann J., *Zur Theorie de Gesellshaftsspiele* Mathematische Annalen Vol. 100, pp 295-320 1928.

[8] Russel and Norvig, *Artificial Intelligence A Modern Approach*. Prentice Hall, Second Edition, 2003.

[9] Holland, J. H., *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[10] Whitley D., *A Genetic Algorithm Tutorial* Statistics and Computing (4) 1994.

[11] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Verlag, Symbolic Computation Series, Third, Revised and Extended Edition 1996.

[12] Radcliffe, N. J., *Genetic Neural Networks on MIMD Computers*. Ph.D. Thesis (Edinburgh University), 1990.

[13] Baker, James E., *Reducing Bias and Inefficiency in the Selection Algorithm* Proceedings of the 2nd International Conference on Genetic Algorithms and their Application Hillsdale, New Jersey: L. Erlbaum Associates 1987.

[14] Herrera F., Lozano M., and Sanchez A.M., *A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study* International Journal of Intelligent Systems 18 2003.

[15] Hansen N., Muller S. D., Koumoutsakos P., *Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)* Evolutionary Computation Vol. 11 No. 1 2003.

[16] Syswerda G., *Uniform Crossover in Genetic Algorithms* Proceedings of the 3rd International Conference on Genetic Algorithms 1989.

[17] Werbos P., *Backpropagation Through Time: What It Does and How to Do It* Proceedings of the IEEE, Vol 78, No. 10 1990.

[18] Cybenko G., *Approximation by superpositions of a sigmoidal function* Mathematics of Control, Signals, and Systems (MCSS), Vol. 2, No. 4 1989.

[19] Fahlman S. E., Lebiere C., *The Cascade-Correlation Learning Architecture* Advances in Neural Information Processing Systems 2 1990.

[20] Schraudolph N., *Online Local Gain Adaptation for Mutli-Layer Perceptrons* Tech. Rep. IDSIA-09-98, Istituto Dalle Molle di Studi sullIntelligenza Articiale 1998.

[21] Bishop C. M., *Neural Networks for Pattern Recognition* Oxford University Press 1995.

[22] Genesereth M., Love N., Pell B., *General Game Playing: Overview of the AAAI Competition* AI Magazine Vol 26, No 2 2005.

[23] Thielscher M., *GDL-II* Springer-Verlag 2010.