Mastering Quoridor

by

Lisa Glendenning

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Bachelor of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2005

Acknowledgments

I would first like to acknowledge and thank Dr. Darko Stefanovic, my research mentor, for his guidance and encouragement throughout the process of developing this research project. I would also like to thank Dr. Chris Moore for his suggestion of Quoridor as a subject of study, and for sharing his ideas on the subject, including the use of random walks. Finally, I would also like to express my gratitude to Dr. Terran Lane for suggesting a machine learning algorithm, and to Joseph Farfel for testing my agent.

Mastering Quoridor

by

Lisa Glendenning

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Bachelor of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2005

Mastering Quoridor

by

Lisa Glendenning

Computer Science, University of New Mexico, 2005

Abstract

In this paper, the board game Quoridor is analyzed, and the development of a program able to play Quoridor is described. Since there is no existing body of expert knowledge on the game of Quoridor, the set of useful features of the game has not yet been defined. Therefore, a number of features are proposed in this paper. A genetic algorithm was used to optimize the weights of a linear weighted evaluation function using these features. The effectiveness of this genetic algorithm as a learning algorithm is evaluated. The resulting Quoridor artificial player is able to play adequately, but not well enough to beat a human player.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The notion that computers might play games has existed at least as long as computers. In the nineteenth century, Charles Babbage considered programming his Analytical Engine to play chess, and later thought about building a machine to play tic-tac-toe [17]. Games provide a useful domain to study machine learning and other artificial intelligence techniques, while providing a structured problem space in which optimization algorithms can be applied to search for solutions.

While computers are acknowledged to be world champions in games like chess and Othello, there are still many complex games for which artificial players have not yet been able to reach the level of play of human expert players, or even that of amateurs. One such game is Quoridor. Quoridor is a relatively new game (1997) that, to the author's knowledge, has not yet been extensively analyzed. One attempt to develop an artificial player for Quoridor is documented in [15].

Quoridor is a board game of 9 rows and 9 columns for 2 or 4 players. Each player has a pawn that begins at one side of the board, and the objective of the game is to move your pawn to the opposite side of the board. The twist on this simple goal is that each player has a certain number of fences that they can place anywhere on the board to hinder the other

players progress. This makes the game interesting and difficult to play, because there are many possible moves and judging who is winning is not trivial.

To develop a computer program to play the game of Quoridor, I used the same basic approach as many game-playing algorithms. My artificial player uses a minimax search algorithm on the game tree and a linear weighted evaluation function on the leaves. Since the field is lacking expert knowledge on tactics and strategies for this particular game, I analyzed the game and proposed some features for use in the evaluation function, most of which quantify paths from the pawn to the winning side of the board.

Instead of statically assigning weights to these features based on my limited playing experience, I decided to employ a learning algorithm to tune the weights. I tried two different algorithms; the first algorithm, based on single-layer neural network learning, was quickly discovered to be ineffectual. The second algorithm, a member of the genetic algorithm family, was more successful, but does not seem to be an optimal algorithm for the task either. The resulting Quoridor player is able to play the game, but does not pose a challenge for a human player. This project hopefully lays the groundwork for more advanced research on developing algorithms for playing Quoridor.

Chapter 2

Background

2.1 Quoridor

Quoridor ^{TM}is a relatively new board game (1997) developed by Gigamic, and was named "Games Magazine's Game of the Year" for 1998. Quoridor has not been extensively analyzed, unlike classic artificial-intelligence games such as Chess and Go. Quoridor has deceptively simple rules from which arise surprisingly complex tactics. Over the course of the game, players create a maze that they must be the first to navigate through to win.

Quoridor has a 9×9 board, and can be played with two or four players. This project focuses only on the two-player game. Each player has a pawn, black or white, that begins the game in the middle of opposing edges of the board (see Figure 2.1). A player wins when his/her pawn has reached any square of the opposite edge of the board from which it started.

A draw determines who is allowed the first move. On a player's turn, the player can either move his/her pawn one square in a cardinal direction, or place one fence. Each player begins the game with ten fences. These fences can be placed anywhere on the

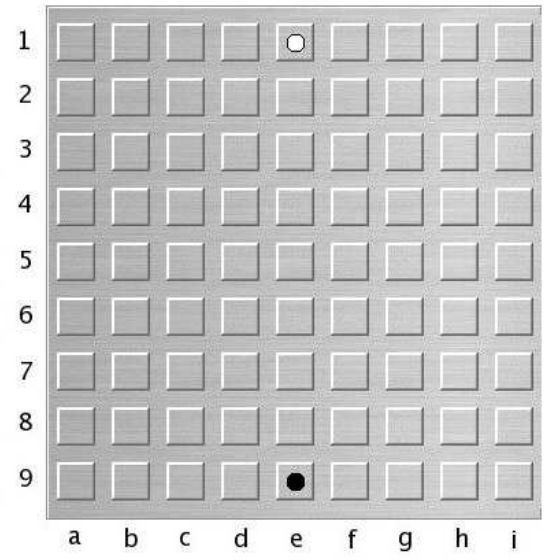


Figure 2.1: Quoridor board.

board, with certain restrictions, to hinder the opponent or help yourself. A pawn cannot move through a fence. Fences must be placed between two sets of two squares horizontally or vertically (see Figure 2.4) and cannot intersect an existing fence. Also, a fence cannot be placed that completely blocks a player's access to his/her goal.

If the two pawns are adjacent to each other with no fence between them, the player whose turn it is can jump his/her pawn over the opponent's pawn. This jump must be in a straight line, unless a fence is behind the other pawn, in which case the jump must be diagonal (see Figure 2.2). In the four person game, it is forbidden to jump more than one pawn. One situation that is not covered by the official rules is illustrated in Figure 2.3 and concerns whether jumping is allowed when pawns are adjacent and one pawn is on the edge of the board. In my implementation of the game, I allow the edge of the board to be treated similarly to fences for jumping.

For the purposes of describing the play of a Quoridor game, I have developed a move notation based on chess algebraic notation (see `http://www.uschess.org`). In this

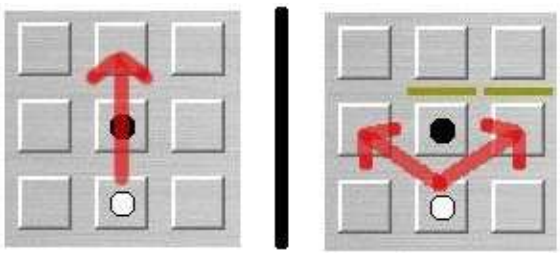


Figure 2.2: Allowed pawn jumps.

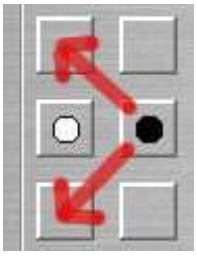


Figure 2.3: The white pawn is on the edge of the board. In my implementation, the black pawn is allowed diagonal jumps in this situation.

notation, each square on the board is uniquely identified by a letter-number combination. The columns are labeled `a` through `i` from left to right, and the rows are labeled `1` through `9` from top to bottom (see Figure 2.1). The black pawn begins at square `e9`, and the white pawn begins at square `e1`. A pawn move is simply identified by the square that the pawn is moved to. A fence move is identified by the fence's "northwest square", which is the square with the smallest number and letter closest to `a` out of the four squares that the fence is adjacent to (see Figure 2.4). This square identification is followed by `h` or `v` to identify whether the fence is placed horizontally or vertically. In addition, a sequence of moves should be identified by the turn number, where a turn is ended after both players have moved, first black then white. For example, `1.e8 e2 2.e7 e3...`. The game begins with turn 1. A single move should also be identified by whether it was black's move or white's move, by adding a `b` or `w` after the turn number and before the dot, e.g., `1b.e8`. Using the convention that black always goes first, a game can be rebuilt from a

given sequence of moves.

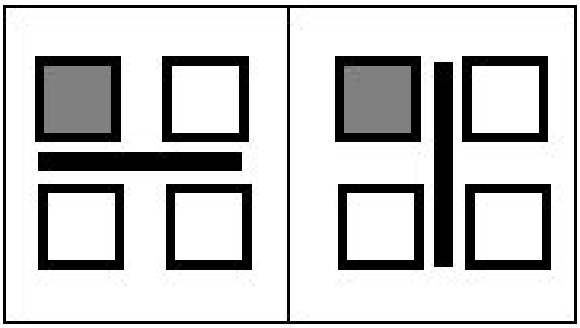


Figure 2.4: Northwest squares of horizontal (left) and vertical fences are shaded.

The rest of this section is devoted to the illustration of the play of a typical Quoridor game. The game begins with the moves `1.e8 e2 2.e7 e3 3.e6 e4`, during which both players charge straight for the goal (Figure 2.5.a).

Now, the black player must make a decision. If either pawn continues moving forward at this point, the opponent will jump and be at an advantage. The black player decides to use a combination of several tactics that I have discovered to be useful. One such tactic is 'creating a box', which simply means enclosing both pawns in the same space and controlling the exit to be advantageous to you but not to the opponent. Also, if you force both pawns to take the same path, that limits the damage that the opponent can cause you, since both pawns are affected by any fences placed in the common path. Another useful tactic involves making the space that your pawn occupies small, so that the longest possible path your pawn could take is smaller as well. Also, although you cannot cut the opponent off completely from his/her goal, you can control the route by which the opponent travels by fencing off part of the board so that the opponent is forced along one path. There is also often a tradeoff between playing defensively, i.e., placing fences in your path to limit the damage that the opponent can do to you, and playing offensively, i.e., making the opponent's path longer. Since fences are a limited resource, they should be used effectively.

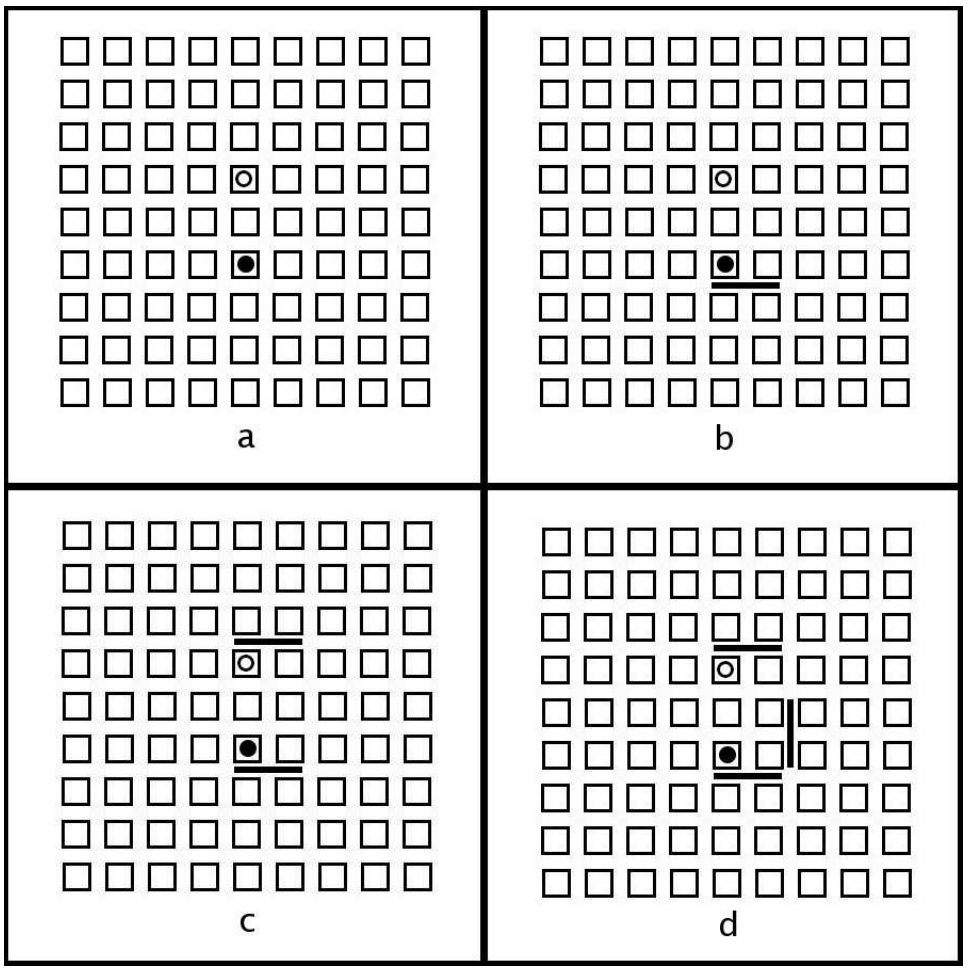


Figure 2.5: Quoridor game: (a) 3w.e4 (b) 4b.e6h (c) 4w.e3h (d) 5b.f5v.

The black player takes the offensive in moves `4.e6h e3h 5.f5v e5h 6.c6h f4 7b.f3v`, shown in Figure 2.5.b-d and Figure 2.6.e-h.

The white player realizes what is happening, but it is too late. The black player is about to create an enclosed area and cut off one side of the board from the white player. The white player tries to control his/her path by cutting off the right side of the board, thus ensuring that the left side will stay open in move `7w.g6h` (Figure 2.7.i).

The black player beats this effort by finishing cutting off the left side in move `8b.a6h`

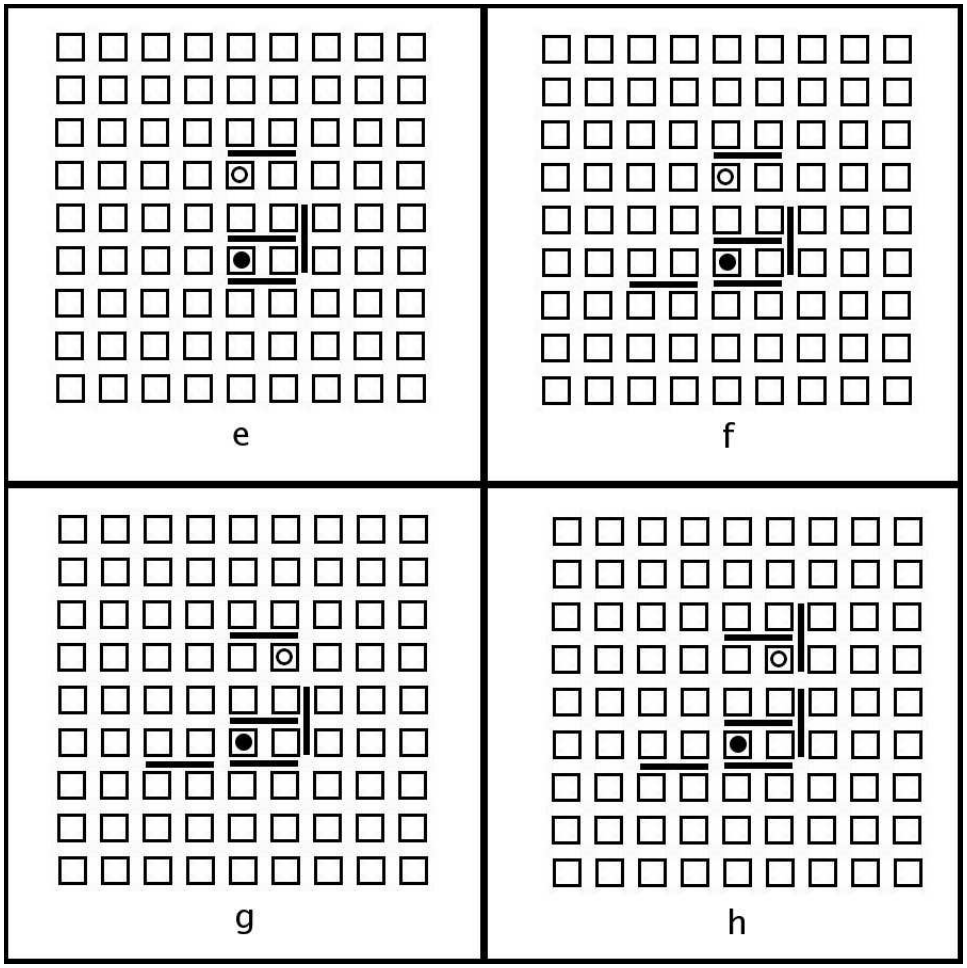


Figure 2.6: Quoridor game: (e) 5w.e5h (f) 6b.c6h (g) 6w.f4 (h) 7b.f3v.

(Figure 2.7.j). The white player starts moving towards the goal again in move `8w.e4` (Figure 2.7.k) and the black player continues creating a box in move `9b.c5v` (Figure 2.7.l).

In moves `9w.d4h` and `10b.d2v`, the white and black players place fences that hurt the opponent but not themselves, then both pawns move, to `10w.d4` and `11b.d6` (Figure 2.8).

The white player continues moving forward and places an offensive fence in moves

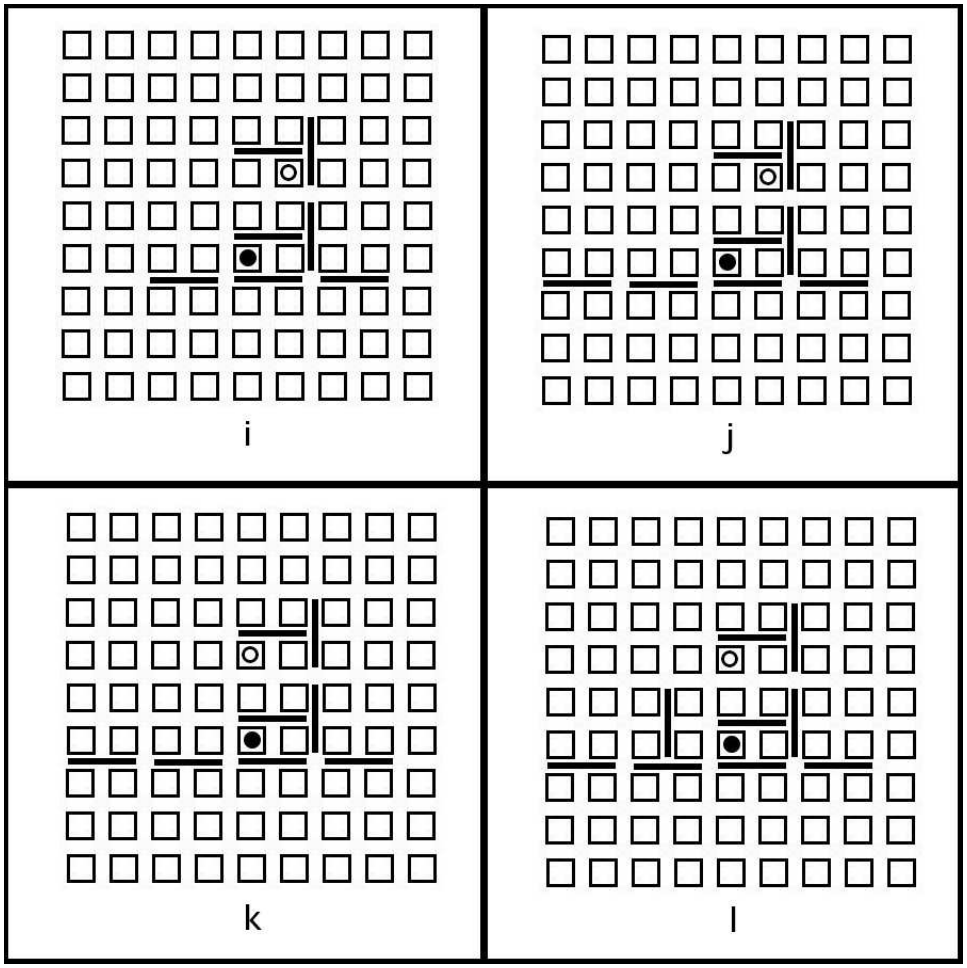


Figure 2.7: Quoridor game: (i) 7w.g6h (j) 8b.a6h (k) 8w.e4 (l) 9b.c5v.

`11w.d3` and `12w.c3h`, while the black player moves forward and places a defensive fence to prevent the white player from lengthening the spiral in moves `12b.d5` and `13b.4h` (Figure 2.9).

The white player moves to `13w.d2` and the black player takes the opportunity to further frustrate the white player in move `14b.c1h`. The white player retaliates by placing another offensive fence in move `14w.a2h`. At this point, the black player has one fence left, and realizes that there is no way the white player can do any more damage, so the black player plays his/her last fence offensively in move `15b.h7h` (Figure 2.10). At this

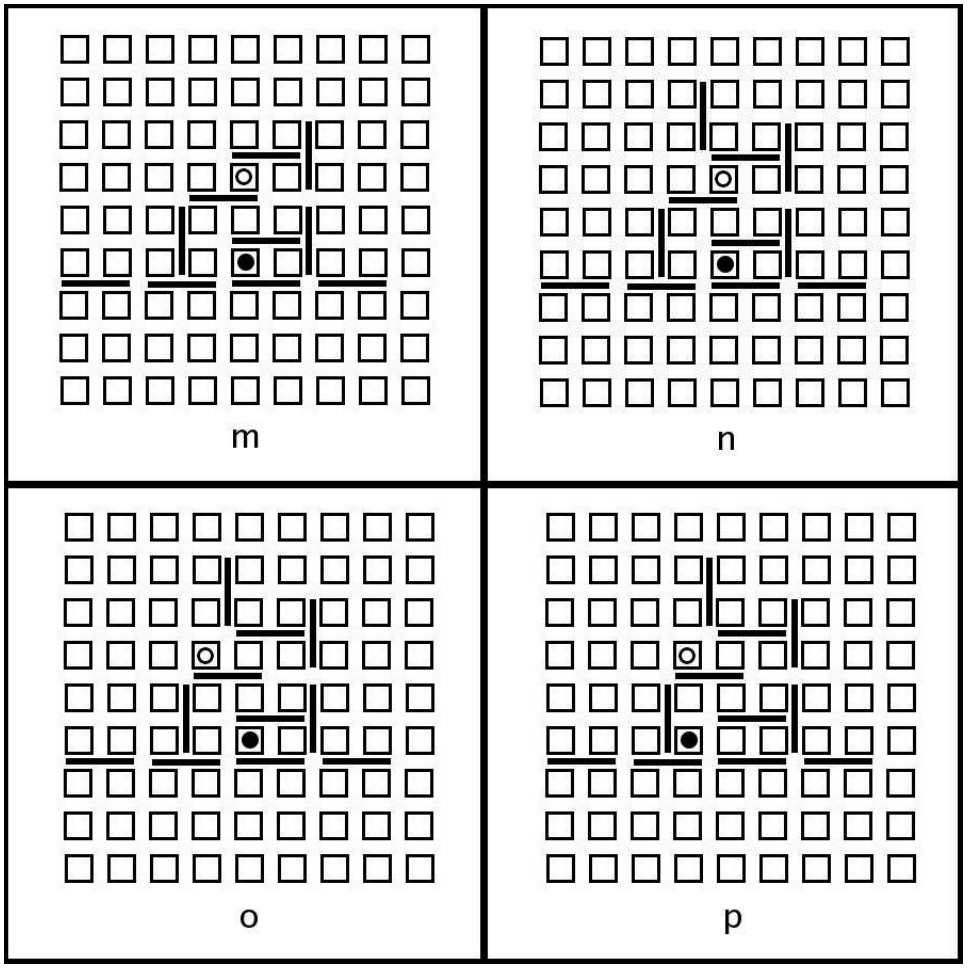


Figure 2.8: Quoridor game: (m) 9w.d4h (n) 10b.d2v (o) 10w.d4 (p) 11b.d6.

point, the game is effectively over. The white player still has four fences remaining, but there is no available placement on the board that would lengthen the black player's path. There is also no point in placing defensive fences since the black player is out of fences. At this point in a Quoridor game, it is a matter of counting the turns until each pawn reaches its goal. The white player surrenders – the black player is twelve squares away from winning, while the white player is twenty squares away from winning. The complete list of moves is given in Table 2.1.

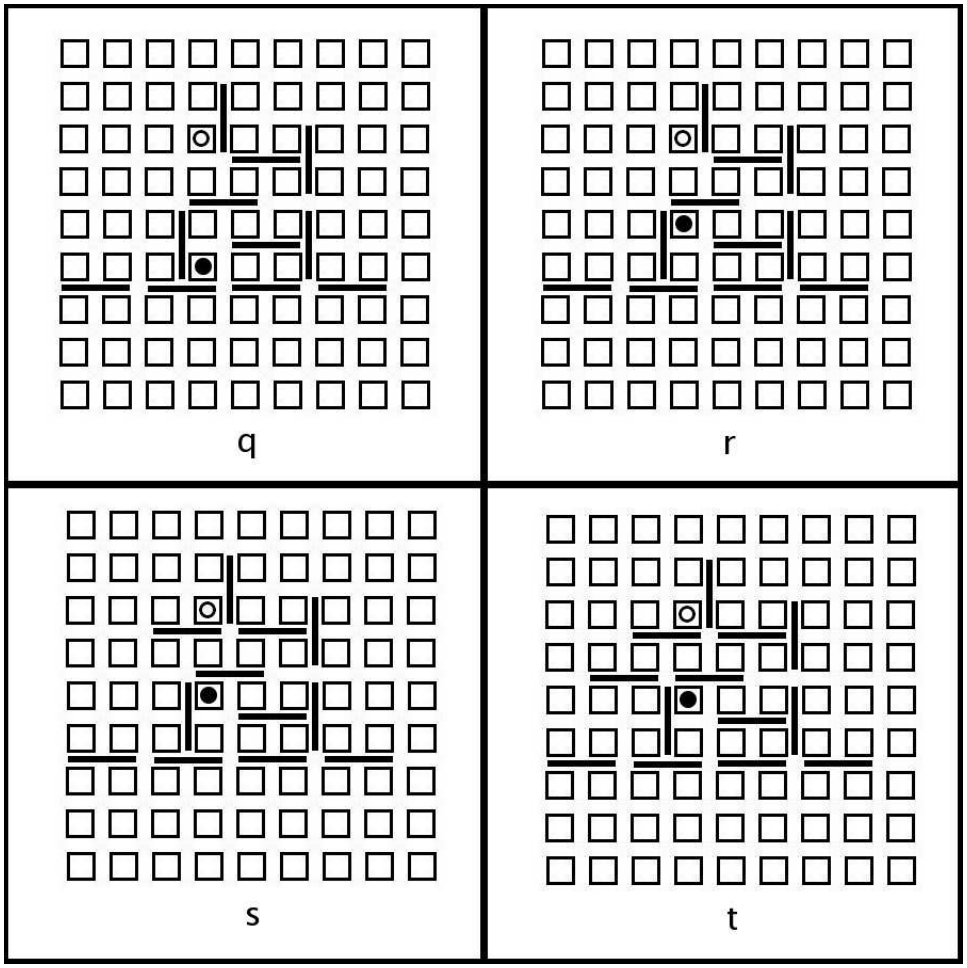


Figure 2.9: Quoridor game: (q) 11w.d3 (r) 12b.d5 (s) 12w.c3h (t) 13b.b4h.

Table 2.1: Move sequence for example Quoridor game.

1.e8 e2	6.c6h f4	11.d6 d3
2.e7 e3	7.f3v g6h	12.d5 c3h
3.e6 e4	8.a6h e4	13.4h d2
4.e6h e3h	9.c5v d4h	14.c1h a2h
5.f5v e5h	10.d2v d4	15b.h7h

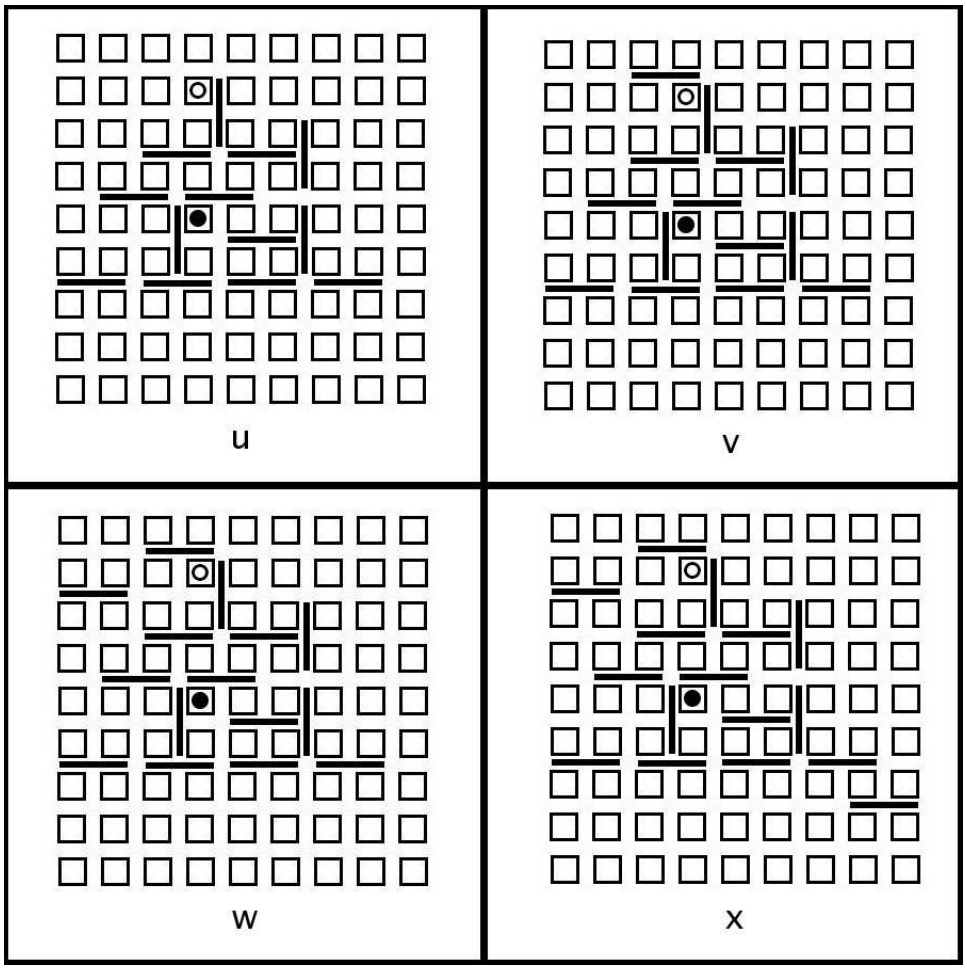


Figure 2.10: Quoridor game: (u) 13w.d2 (v) 14b.c1h (w) 14w.a2h (x) 15b.h7h.

2.2 Computers and Games

Our fascination with games has ancient roots in human history. For example, the game of Go was developed three to four millenia ago in China [20]. Another game, Awari, is believed to have originated from Ethiopia about 3500 years ago [5]. Some of the ancient strategic board games that we know today are believed to have military roots. Instead of having armies go to war, some of the planning and other strategic elements essential for conducting a battle could be practiced on a board [2].

Game playing was one of the first tasks undertaken in artificial intelligence. Indeed, the game of chess has sometimes been referred to as the *Drosophila* of artificial intelligence and cognitive science research – an ideal controllable environment that serves as a test bed for ideas about the nature of intelligence and computational schemes for intelligent systems. Since the beginning of the twentieth century, mathematicians have tried to model classical games to create expert artificial players, but only recently have computers been able to best humans in certain games. Many decades of research focusing on the creation of grandmaster standard computer chess programs culminated in the defeat of Garry Kasparov, the World Chess Champion, by IBM's purpose-built chess computer, Deep Blue, in 1997. Deep Blue, and, since, Deeper Blue, still mainly rely on specialized hardware and brute force methods to gain an advantage over the opponent by examining further into the game tree [13]. Deep Blue managed to search 126 million positions per second on average. Chinook, developed by Jonathan Schaeffer and colleagues, was declared the world champion in checkers in 1994. In 1997, the Logistello program defeated the human world champion in Othello [18].

Traditional artificial intelligence techniques for games rely on exhaustive search of the game tree, evaluation functions based on expert knowledge, large "opening book" and endgame databases, and massive computing power. There are limits, however, to how far this brute-force approach will take artificial players, and researchers are running

into these walls in more complex games, such as those that involve chance and imperfect information. Research in games in artificial intelligence has begun to focus on alternative methods, including adaptive learning techniques.

Adaptive learning was being used for checkers as far back as the 1950's, with Samuel's seminal work in checkers. Gary Tesauro combined Samuel's reinforcement learning method with neural network techniques to develop TD-GAMMON, which is ranked among the top three backgammon players in the world [18]. Another modern example is Anaconda, a checkers player developed by Chellapilla and Fogel. Anaconda uses an artificial neural network (ANN), with 5046 weights, which are evolved via an evolutionary strategy. The inputs to the ANN are the current board state, presented in a variety of spatial forms. The output from the ANN is a value which is used in a mini-max search. During the training period the program is given no information other than a value which indicates how it performed in the last five games. It does not know which of those games it won or lost, nor does it know if it is better to achieve a higher or a lower score. Co-evolution (playing games against itself) was used to develop Anaconda, which has achieved expert levels of play [6].

Evolutionary algorithms (see Section 3.2), such as the co-evolution used to develop Anaconda, are also an area of research in games. Davis et al. [6] used an evolutionary strategy to optimize the weights of a simple evaluation function for Awari, and achieved what they considered a "reasonable level of play." A later Awari player, called Ayo, was developed by Daoud et al. [5]. They used a more sophisticated evaluation function than Davis et al., while also employing a genetic algorithm to evolve the weights. Ayo outperformed the previous player with a shallower game-tree search, showing the advantage in this case of a more accurate evaluation function over a deeper search. Kendall et al. [13] propose an approach for the tuning of evaluation function parameters based on evolutionary algorithms and applies it to chess. In a different application of genetic algorithms, Hong et. al. [21] propose a genetic-algorithm based approach to enhancing the speed and

accuracy of game tree searches.

2.3 Representations of Games

Mathematical game theory views any multiagent environment as a game provided that the impact of each agent on the others is "significant", regardless of whether the agents are cooperative or competitive. In game theory, Quoridor, like chess and checkers, can be described as a deterministic, sequential, two-player, zero-sum game of perfect information with a restricted outcome (win, lose, and draw) – also known as a *combinatorial game* [6].

Deterministic games have no element of chance. Backgammon is an example of a game of *chance*, because play involves rolling dice.

In *sequential* games, players take turns instead of making a move simultaneously. The Prisoner's Dilemma is an example of a *simultaneous* game. In the classical Prisoner's Dilemma, two suspects are arrested and interrogated in separate rooms. Both suspects are offered the chance of walking free if they confess and incriminate the other suspect. Each suspect has two options: cooperate (stay silent) or defect (confess to the crime), and both suspects make their decisions simultaneously, i.e., without knowing the other suspect's decision. If both suspects cooperate, they are punished minimally. If one suspect defects and the other cooperates, the suspect who cooperates gets a maximum sentence while the suspect who defects is not punished. If both suspects defect, they are both punished moderately.

In *zero-sum* games, the utility values at the end of the game are always equal in magnitude and opposite in sign. For example, if one player wins a game of chess $(+1)$, the other player necessarily loses (-1). In *non-zero-sum* games, the players' goals are not necessarily conflicting. The Prisoner's Dilemma is a non-zero-sum game. An example of a corresponding payoff matrix for the Prisoner's Dilemma is illustrated in Table 2.2, where

a higher payoff is a better outcome. Notice that the possible outcomes do not add up to zero.

Table 2.2: The possible payoffs in each outcome of the Prisoner's Dilemma are shown in italics for P1 and in bold for P2.

	P1 Cooperates	*P1* Defects
P2 Cooperates	**-2**, *-2*	**-5**, *0*
P2 Defects	**0**, *-5*	**-4**, *-4*

In games of *perfect information*, the current state of the game is fully accessible to both players. Poker is an example of a game of *imperfect information*, because some information about the state of the game, i.e., the hands of other players, is hidden from the players.

In combinatorial game theory, which is the branch of game theory concerned with combinatorial games, Quoridor can be characterized as *partizan*, in contrast to an *impartial* game, in which the allowable moves depend only on the position and not on which of the two players is currently moving, and where the payoffs are symmetric. Nim is an example of an impartial game. In Nim, players take turns removing objects (counters, pebbles, coins, pieces of paper) from heaps (piles, rows, boxes), but only from one heap at a time. In the *normal* convention, the player who removes the last object wins; in the *misere* convention the player to move last loses [12].

A combinatorial game has a precise definition in combinatorial game theory. The two players are usually called Left (L) and Right (R). Every game has some number of *positions*, each of which is described by the set of allowed *moves* for each player. Each move (of Left or Right) leads to a new position, which is called a (left or right) *option* of the previous position. Each of these options can be thought of as another game in its own right, described by the sets of allowed moves for both players [19].

From a mathematical point of view, all that matters are the sets of left and right options

that can be reached from any given position – we can imagine the game represented by a rooted tree with vertices representing positions and with oriented edges labeled L or R according to the player whose moves they reflect. The root represents the initial position, and the edges from any position lead to another subtree, the root of which represents the position just reached [19].

A game can be identified by its initial position and completely described by the sets of left and right options, each of which is another game. This leads to the following recursive definition of a game: Let L and R be two sets of games. Then the ordered pair $G := (L, R)$ is a game. Note that the sets L and R of options may well be infinite or empty [19].

Traditional combinatorial game theory also assumes that the player that makes the last move wins, and that no position may be repeated. A *loopy* game, however, does not impose the no-repetition assumption, and thus requires a more complex theory. Quoridor is an example of a loopy game. A loopy game is played on a directed pseudograph (a directed graph that is allowed to have multiple edges from one vertex to another, or edges from a vertex to itself). Each vertex of this graph corresponds to a position of the game. The edges are partitioned into two sets, Left's and Right's. Each player plays along his own edges only, changing the position from the vertex at the start of the edge to that at the end. Left and Right play alternately; if either is ever left without a move, he loses. To specify a loopy game completely, one needs to specify a start-vertex as well as the graph and its edge-partition. One also needs to specify for each legal infinite sequence of moves whether it is won by Left, won by Right, or is drawn [16].

2.4 Game-Playing Approach

2.4.1 Game Trees

In artificial intelligence, games like Quoridor are often treated as *adversarial search problems*. To define this problem, we will consider a game with two players, called MAX and MIN, who take turns moving until the game is over. At the end of the game, points are awarded to the winner and penalties are given to the loser. Such a game can be formally defined as a kind of search problem with the following components [18]:

- The *initial state*, which includes the board position and identifies the player to move.
- A *successor function*, which returns a list of $(move, state)$ pairs, each indicating a legal move and the resulting state.
- A *terminal test*, which determines when the game is over. States where the game has ended are called *terminal states*.
- A *utility function* (also called an objective function or payoff function), which gives a numeric value for the terminal states. In zero-sum games, these values are equal in magnitude but opposite in sign.

The initial state and the legal moves for each side define the *game tree* for the game. Figure 2.11 shows part of the game tree for tic-tac-toe. The optimal solution to this search problem is a sequence of moves leading to a *goal state* – a terminal state that is a win. MAX must find a contingent *strategy*, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, and so on. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent [18].

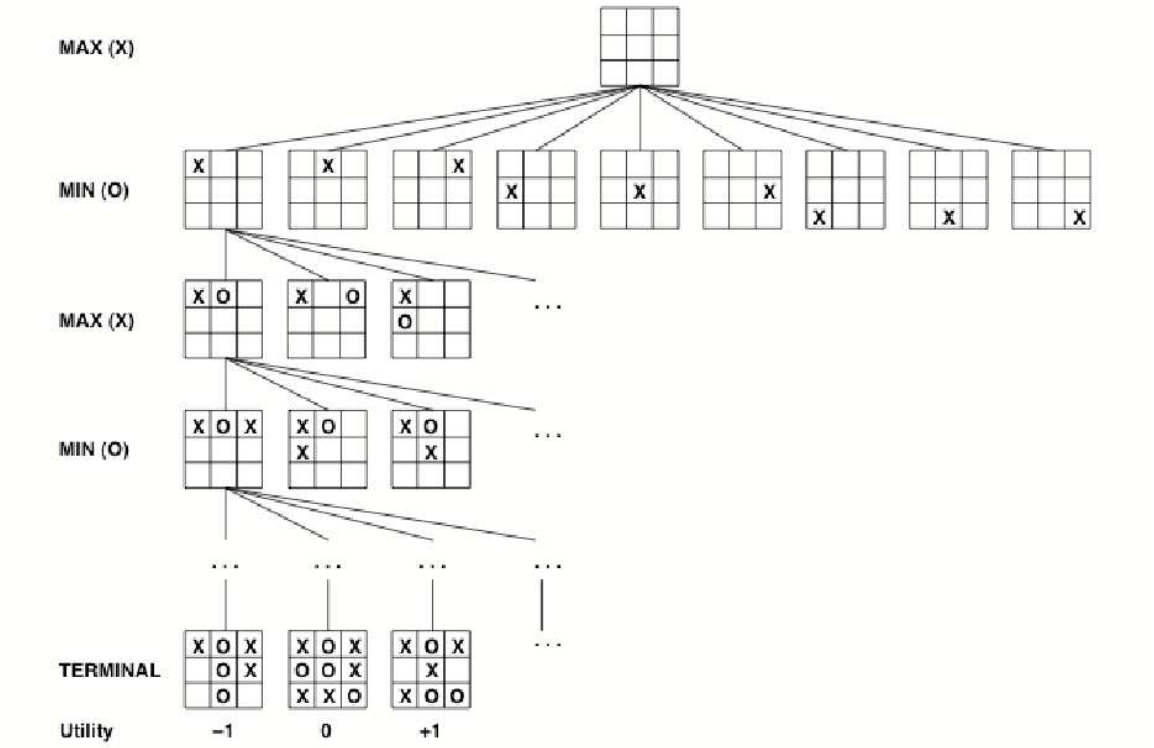


Figure 2.11: A partial search tree for the game of tic-tac-toe. The top node is the initial state, from which MAX has nine possible moves. Play alternates between MAX and MIN until eventually we reach terminal states, which can be assigned utilities for MAX [18].

To describe this optimal strategy, we will consider the trivial game shown in Figure 2.12. The possible moves for MAX at the root are labeled A_1, A_2, and A_3. The possible replies to A_1 for MIN are A_{11}, A_{12}, and A_{13}, and so on. This particular game ends after one move each by MAX and MIN. This tree is therefore one *move* deep, consisting of two *half-moves*, each of which is called a *ply* [18].

Given a game tree, the optimal strategy can be determined by examining the *minimax value* of each node, which is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there until the end of the game. The minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX

will prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value. We can therefore define the minimax value of a node n, MINIMAX-VALUE(n), as follows:

$$\begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node} \end{cases}$$

This definition has been applied to the game tree in Figure 2.12. In this game tree, we can now identify the *minimax decision* at the root: move A_1 is optimal for MAX. The definition of optimal play for MAX assumes that MIN will also play optimally, thereby maximizing the *worst-case* outcome for MAX. If MIN does not play optimally, then MAX will do even better [18].

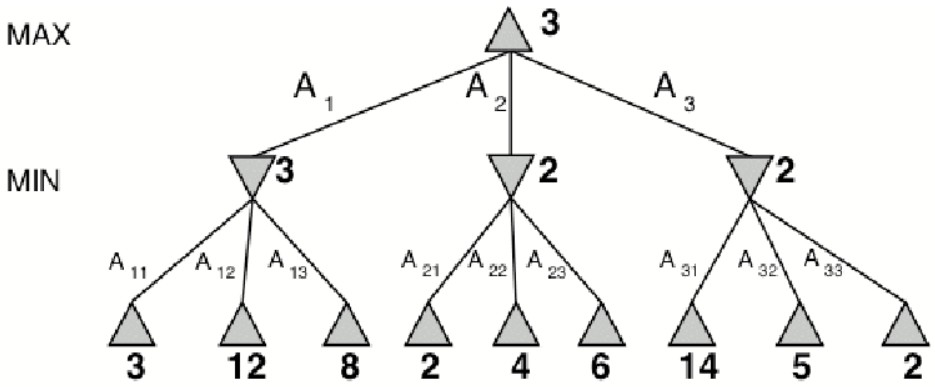


Figure 2.12: A two-ply game tree. The $\triangle$ nodes are MAX nodes (meaning it is MAX's turn to move), and the $\triangledown$ nodes are MIN nodes. The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is A_1, and MIN's best reply is A_{11} [18].

The *minimax algorithm* (Figure 2.13) computes the minimax decision from the current state. It uses a simple recursive computation of each successor state which proceeds all the way down to the leaves of the tree, and then the minimax values are *backed up* through the tree as the recursion unwinds. The minimax algorithm performs a complete depth-first

exploration of the game tree. If the maximum depth of the tree is m, and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all successors at once, or $O(m)$ for an algorithm that generates successors one at a time [18].

```
function MINIMAX-DECISION(state) returns a move
    v ← MAX-VALUE(state)
    return the move in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Figure 2.13: Minimax algorithm.

2.4.2 Minimax Enhancements

For simplicity, we can modify the game tree values slightly and use only maximization operations. The trick is negate the returned values from the recursion. In order to do this, the utility value of a state is determined from the viewpoint of the player whose turn it is at that state, not, as is done in the minimax algorithm, always from MAX's viewpoint. This is called the *Negamax algorithm* [14] (Figure 2.14).

```
function NEGAMAX-DECISION(state) returns a move
    v ← NEGAMAX-VALUE(state)
    return the move in SUCCESSORS(state) with value v

function NEGAMAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, −NEGAMAX-VALUE(s))
    return v
```

Figure 2.14: Negamax algorithm.

The problem with minimax search is that the number of states it has to examine is exponential in the number of moves. By realizing that it is possible to compute the correct minimax decision without looking at every node in the game tree, we can effectively cut the exponent in half. This technique is called *alpha-beta pruning*. When applied to a standard minimax tree, it returns the same decision as minimax would, but prunes away branches that cannot influence the final decision. Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node v somewhere in the tree such that MAX has a choice of moving to that node. If MAX has a better choice α either at the parent node of v or at any choice point further up, then v will never be reached in actual play (Figure 2.15) [18].

Alpha-beta pruning gets its name from two parameters that describe bounds on the backed-up values that appear anywhere along the depth-first path in the tree. α is the value of the best (highest-value) choice we have found so far at any choice point along the path for MAX. β is the value of the best (lowest-value) choice we have found so far at any choice point along the path for MIN. Alpha-beta search (Figure 2.16) updates the values of

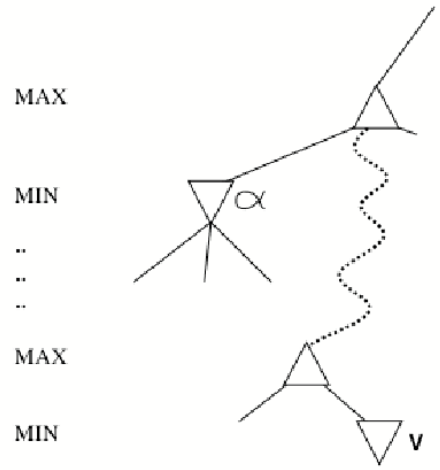


Figure 2.15: The general principle of alpha-beta pruning: If α is better than *v* for MAX, *v* will never be reached in play [18].

α and β as it goes along and prunes the remaining brances at a node as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively [18].

The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. If the successors that are likely to be best are examined first, then alpha-beta needs to examine only $O(b^{m/2})$ nodes, instead of $O(b^m)$ for minimax. If successors are examined in random order, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate *b* [18].

Even with alpha-beta pruning, for any complex game, the game tree is orders of magnitude too large to be examined fully in a practical amount of time. One solution is to examine the tree to a fixed depth, and then apply a heuristic *evaluation function*, EVAL, to the leaves of this search. These values can then be backed up the tree as utility values are [18]. Since game-playing programs are often given a certain time limit in which to make a decision as to which move to make, one method of maximizing the depth of the search within a fixed time limit is *iterative deepening*. The idea of iterative deepening

```
function ALPHA-BETA-NEGAMAX(state) returns a move
    v ← NEGAMAX-VALUE(state, −∞, ∞)
    return the move in SUCCESSORS(state) with value v

function NEGAMAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        child ← −NEGAMAX-VALUE(s, −β, −α)
        if (child > v) then v ← child
        if (v > α) then α ← v
        if (α >= β) then return α
    return v
```

Figure 2.16: Alpha-beta negamax algorithm.

(Figure 2.17) is to search the game tree to a fixed depth d of one ply, then repeatedly extend the search by one ply until time runs out. Only the minimax decision from the most recent completed iteration is used. Although iterative deepening seems to waste time by re-examining the upper levels of the tree repeatedly on each iteration, due to the exponential nature of game tree search, the overhead cost of the preliminary $d-1$ iterations is only a constant fraction of the d-ply search. Iterative deepening can also improve the effectiveness of alpha-beta pruning by using the results of previous iterations to improve the move ordering of the next iteration [14].

2.4.3 Evaluation Functions

An evaluation function returns an estimate of the expected utility of the game from a given position. How is a "good" evaluation function defined? First, the evaluation function should not take too long to calculate. Second, for non-terminal states, the evaluation func-

```
function ITERATIVE-ALPHA-BETA-NEGAMAX(state) returns a move
    d ← 1
    while time is not up do
        m ← ALPHA-BETA-NEGAMAX(state, d)
        d = d + 1
    return m

function ALPHA-BETA-NEGAMAX(state, d) returns a move
    v ← NEGAMAX-VALUE(state, d, −∞, ∞)
    return the move in SUCCESSORS(state) with value v

function NEGAMAX-VALUE(state, d, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    if (d == 0) then return EVAL(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        child ← −NEGAMAX-VALUE(s, d − 1, −β, −α)
        if (child > v) then v ← child
        if (v > α) then α ← v
        if (α >= β) then return α
    return v
```

Figure 2.17: Iterative-deepening alpha-beta negamax algorithm.

tion should be strongly correlated with the actual chances of winning [18]. When there is a time limit for the search, a tradeoff has often been found between the accuracy, i.e., computational time, of the evaluation function and the depth examined by the search. In other words, if the algorithm can quickly evaluate a single node, then the algorithm will be able to examine many nodes, but if the algorithm takes a long time to evaluate a single node, not as many nodes can be examined.

Most evaluation functions work by calculating various *features* of the state – for example, the number of pawns possessed by each side in a game of chess. The features, taken together, define various *categories* or *equivalence classes* of states: the states in each cate-

gory have the same values for all the features. Most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value. For example, introductory chess books give an approximate *material value* for each piece. Other features in chess are "good pawn structure" and "king safety", which are assigned a numerical value. These feature values can then be individually weighted and summed; this kind of evaluation function is called a *weighted linear function* and can be expressed as

$$\mathsf{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

where each w_i is a weight and each f_i is a feature of the state [18]. If the weights are viewed as a row vector $\mathbf{w}$ and the features as a column vector $\mathbf{f}$, then we can equivalently define EVAL as

$$\mathsf{EVAL}(s) = \mathbf{w} \cdot \mathbf{f}$$

Weighted linear functions assume that the contribution of each feature is independent of the values of the other features. To avoid this assumption, current programs for chess and other games use *nonlinear* combinations of features [18].

The features and weights used in chess-playing programs come from centuries of human chess-playing experience. In games where this kind of experience is not available, the weights of the evaluation function can be estimated by machine learning techniques.

Chapter 3

Method

3.1 Learning

Inductive learning takes place as the agent observes its interactions with the world and its own decision-making process. A learning agent can be thought of as containing a *performance element* that decides what actions to take and a *learning element* that modifies the performance element so that it makes better decisions [18].

For the application of learning weights for an evaluation function (see Section 2.4), the performance element is the algorithm that, given a state, decides which move to make. But what should the learning element be?

We would expect the learning process to produce the following results: If a feature is not found to be a useful evaluator of the board state, one would expect that, over a long enough period of learning, the weight of that feature would tend to zero. One would also predict that the weights of the features that positively correlated with winning the game would become positive, while the weights of the features that turned out to be an indicator of losing the game would become negative. One would also predict that the weights of

the features that are the most important and reliable indicators will have a larger weight relative to the weights of the features that are less important.

Dr. Terran Lane proposed that for feature weight learning I should use an algorithm based on a *single-layer neural network*, or *perceptron network* with a single output unit (Figure 3.1). The output unit y is a weighted sum of the input units $\mathbf{x}$:

$$y = \sum_{j=0}^{n} W_j x_j$$

There is a simple learning algorithm that will fit a threshold perceptron to any linearly separable traning set. The idea behind the perceptron-learning algorithm is to adjust the weights of the network to minimize some measure of the error on the training set. Thus, learning is formulated as an optimization search in *weight space* [18].

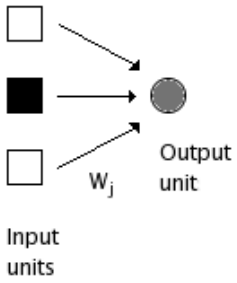


Figure 3.1: A single-layer neural network (perceptron network) with three inputs and one output.

The proposed learning algorithm based on perceptron-learning works as follows: First, create a population of agents with randomly generated weights. Also, create a population of agents that will be kept static for the purpose of evaluating the learning progression. Have the agents play each other in a round robin fashion, and perform a learning operation on each agent at the end of a game. To perform the learning operation, the agent stores the *feature vectors* for each state encountered during the course of the game. A feature vector contains the values of the features for a particular state before the *weight vector* is applied

to obtain a heuristic value for the expected utility value of the state. When the game is over, we can calculate the following value $\mathbf{f}_\varepsilon$:

$$\mathbf{f}_\varepsilon = \sum_{j=1}^{n} \gamma^j \mathbf{f}_{n-j}$$

where n is the number of feature vectors, $\mathbf{f}_i$ is the ith feature vector of the game, and γ is the *learning rate*, and is in the range $[0,1]$. We then modify the weight vector $\mathbf{w}$ of the agent in the following way:

$$\mathbf{w} = \begin{cases} \mathbf{w} + \mathbf{f}_\varepsilon & \text{if the agent won the game} \\ \mathbf{w} - \mathbf{f}_\varepsilon & \text{if the agent lost the game} \end{cases}$$

The idea of this algorithm is to modify the weight of a feature to reflect the distribution of that feature throughout the game, where the values of that feature at the end of the game are weighted more heavily (by the learning rate) to reflect the idea that the states near the end of the game more directly influence whether the game is won or lost. If a feature has a high value near the end of a game that is won, the weight for that feature will be moved in the positive direction. Similarly, if a feature has a high value near the end of a game that is lost, the weight for that feature will be moved in the negative direction.

When I experimented with this algorithm, I discovered that it did not work well for this particular application. I implemented an agent that used two features in its evaluation function, SPP and SPO (see Section 3.4). Common sense dictates that the weight of SPP should be positive, and the weight of SPO negative. After a few rounds of learning, however, I discovered that both weights were strongly tending negative, leading to an agent that was completely ineffective at moving. The reason for this was that the value of SPP was always between 0 and 1, so whenever the agent lost (which was often) the weight would always be pushed in the negative direction. It was not possible for the weight to be pushed in the correct (positive) direction when the agent lost. This algorithm might be effective with features that range from -1 to 1, but I decided to abandon this approach and

explore using a genetic algorithm as a weight learning function. I chose to use a genetic algorithm because the function of a genetic algorithm is to optimize numeric values.

3.2 Genetic Algorithms

Evolutionary algorithms are a family of computational models inspired by the biological processes of evolution and natural selection. For an in-depth discussion of how evolution has produced problem-solving adaptations in nature see [7]. *Genetic algorithms* (GAs) are a subset of evolutionary algorithms. A GA is any population-based model that uses selection and recombination operators to generate new sample points in a search space [22]. These algorithms can be described as function optimizers, but this does not mean that they yield globally optimal solutions [23]. GAs have been successfully applied to the fields of optimization, machine learning, neural networks, and fuzzy logic controllers, among others [21].

An implementation of a genetic algorithm begins with a *population* of typically random *chromosomes*, which can be thought of as candidate solutions for some problem of interest. Each chromosome is then evaluated to obtain its individual *fitness* in solving a particular problem. Then *selection*, *recombination*, and *mutation* are applied to the population to create the *next population*. These operations are similar to, but not identical to, their biological counterparts. The process of going from the current population to the next population constitutes one generation in the process of a genetic algorithm [23]. An overview of a sample genetic algorithm is shown in Figure 3.2.

Given that variation exists within a species, individuals within a population will differ in their ability to cope with a given environment and successfully reproduce. This varying reproductive success of individuals based on their different genetic constitutions is *natural selection*. The concept of natural selection is often simplified to “survival of the fittest.”

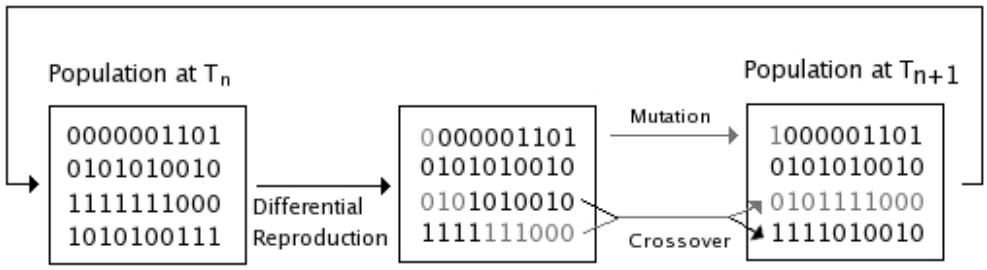


Figure 3.2: Sample genetic algorithm overview [11]. A population of four chromosomes is shown at time *n*. During differential reproduction, in which chromsomes are chosen to reproduce based on their individual fitness, the first chromsome is selected once, the second is selected twice, the third is selected once, and the fourth is selected zero times. After selection, the mutation operator is applied to the first chromsome, and the crossover operator is applied to the third and fourth chromsomes. The resulting population is shown in the box labeled T_{n+1}.

In genetic algorithms, the fitness of individuals is determined by applying a *fitness evaluation function* to the chromosomes. The fitness evaluation function provides a measure of performance with respect to a particular set of parameters. The fitness level of an individual is then used to determine the probability that it will produce offspring for the next generation.

There are a number of ways to do selection. One method involves mapping the population onto a roulette wheel, where each individual is represented by a space that proportionally corresponds to its fitness (Figure 3.3). By repeatedly spinning the roulette wheel, individuals are chosen using *stochastic sampling with replacement* to choose the pool of chromosomes that will combine to form the next population. This method increases the probability that the chromosomes of the current population with the highest fitness will contribute to the next population [22].

A pair of chromosomes is selected from the eligible pool with some probability corresponding to their fitness, and then recombination is applied to the pair to produce one

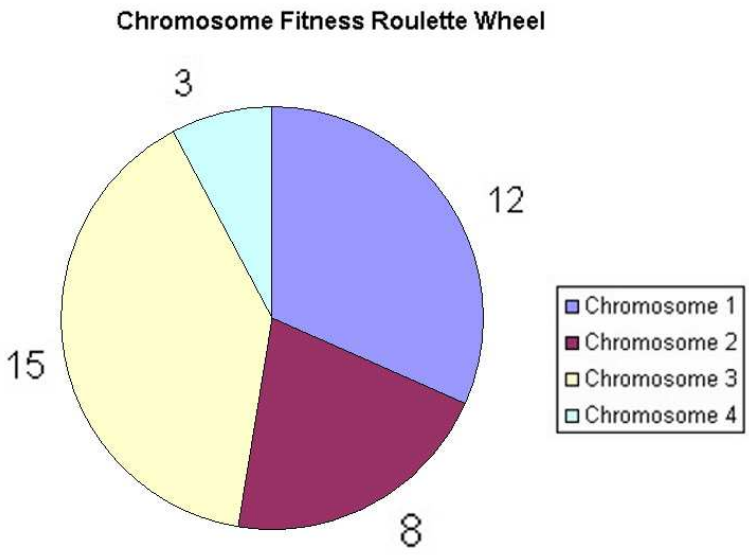


Figure 3.3: Mapping a population onto a roulette wheel based on fitness.

or more offspring chromosomes. A chromosome can be thought of simply as encoded information. A classic chromosome is a string of bits, for example <`10101110010`>. Chromosomes are not limited to bits, however. Another example of a chromosome is <`"I'm a gene", 80, xyxxxyy`>. Mimicking the structure of biological chromosomes, digital chromosomes can be arbitrarily divided into blocks of information called *loci*. In the first example, each bit can be viewed as a locus, and in the second example the loci are separated by commas. Continuing with the biological analogy, each atomic subunit of a locus is a *base*.

Recombination involves combining the parent chromosomes in some way. One method of recombination involves *crossover points* (Figure 3.4), at which parent chromsomes swap loci.

After recombination, a mutation operator can be applied to the offspring chromosomes. With classic chromosomes, this means that each bit has a certain low probability of being

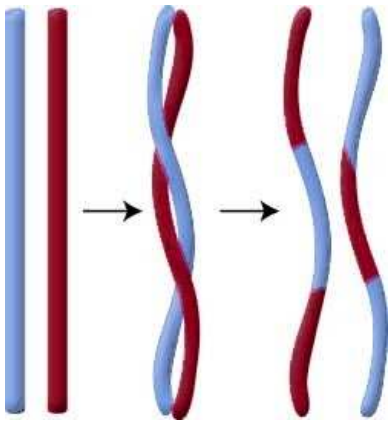


Figure 3.4: Recombination of two parent chromosomes with two crossover points to produce two offspring.

flipped. More generally, this means that each locus has some probability of being modified in some random way. Typically, the mutation rate is less than 1% [23]. Higher mutation rates make the genetic search less directed and more random.

A number of theories have been proposed that attempt to rigorously analyze the performance of evolutionary algorithms [10]. The general argument about genetic algorithm performance has three components [11]:

- Independent sampling is provided by large populations that are initialized randomly.
- High-fitness individuals are preserved through selection, and this biases the sampling process towards regions of high fitness.
- Crossover combines partial solutions, called "building blocks," from different chromosomes onto the same chromosome, thus exploiting the parallelism provided by maintaining a population of candidate solutions.

In an analysis of GAs in comparison with other algorithms, Baum et al. find that a variation of a GA is the strongest known approach to solving the *Additive Search Problem* (ASP),

which can be described as follows: Let $X \equiv \{1, 2, \ldots, L\}^N$. There is an oracle that returns the number of components common to a query $x \in X$ and a target vector $t \in X$. The objective is to find t with as few queries as possible. For simplicity, it is assumed that $N > L$ [1].

Perhaps the most common application of GAs is multiparameter function optimization. Many problems can be formulated as a search for an optimal value, where the value is a complicated function of some input parameters. For problems that don't require the exact optimum, GAs are often an appropriate method for finding "good" values, where "good" can be near optimal or even a slight improvement over the previously best known value. The strength of the GA lies in its ability to manipulate many parameters simultaneously[11]. Deb et al. demonstrate the power of a GA variant in tackling real-parameter optimization problems. The performance of this GA is investigated on three commonly used test problems and is compared with a number of evolutionary and optimization algorithms [8].

3.3 Features of Quoridor

In order to use a heuristic evaluation function, *features* of the game must be identified (see Section 2.4). Since Quoridor is a virtually unknown game in the game-playing literature, there was no reservoir of expert knowledge to draw upon, as was done in other game-playing programs. Therefore, I proposed a number of features from observation of play.

3.3.1 Path-finding Algorithms

One obvious feature of Quoridor is related to the goal of the game – moving your pawn to the other side of the board. One indicator of how well a player is doing is how many squares away that player's pawn is from that player's goal. A player's *goal* is defined as

the row or column of squares on the opposite side of the board from where that player's pawn began the game. The black player's goal is the squares `a1, b1, c1, d1, e1, f1, g1, h1, i1`, and the white player's goal is the squares `a9, b9, c9, d9, e9, f9, g9, h9, i9`. Because optimal path-finding algorithms can be expensive, I considered a variety of pathfinding-related features that vary in accuracy and performance. As discussed in Section 2.4, there is often a tradeoff between the accuracy of the evaluation function and the depth of the search.

The simplest path-related feature I used was a boolean value representing whether the pawn is on the *goal side* of the board (see Figure 3.5). This value is a 1 if the player's pawn is on a square in the goal side, otherwise it is a 0.

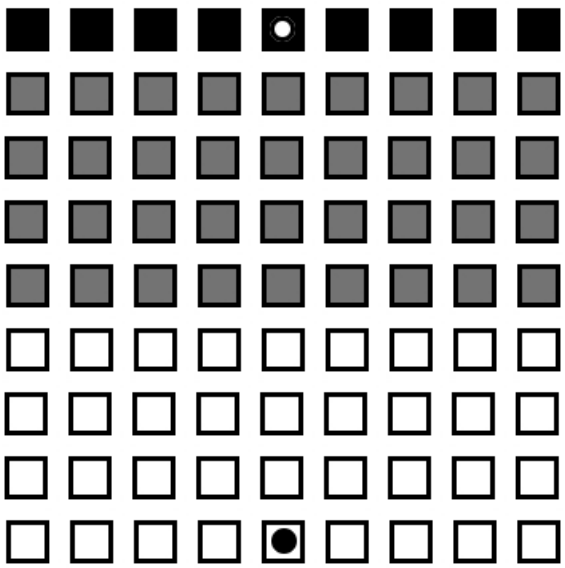


Figure 3.5: The *goal* of the the black pawn is shaded black, and the black player's *goal side* of the board is shaded gray.

The next-simplest feature I used was the *Manhattan Distance*, which is the distance between two points measured along axes at right angles. In Quoridor, the Manhattan Distance can also be thought of as the shortest path from the pawn to the goal ignoring all obstacles (see Figure 3.6).

At this point I ran into a problem with using path lengths in the evaluation function. To facilitate the learning of the weights (see Section 3.1), I decided to follow the convention that a positive feature is "good" for the player who is doing the evaluation, while a negative feature is "bad". With path lengths, however, the smaller the number, the shorter the path, and the better the feature for the player. For the Manhattan Distance, I addressed this problem by subtracting the actual Manhattan Distance from the maximum possible Manhattan Distance value. This turns the feature into a measure of progression along a path instead of a measure of the distance remaining until the end of the path. I also decided to make my features similar and bounded in their ranges by normalizing them to the range $[0, 1]$. For the Manhattan Distance, I simply divided the progression value by the maximum Manhattan Distance. In the situation shown in Figure 3.6, the final Manhattan Distance feature value would be $\frac{9-5}{9} = 0.556$.

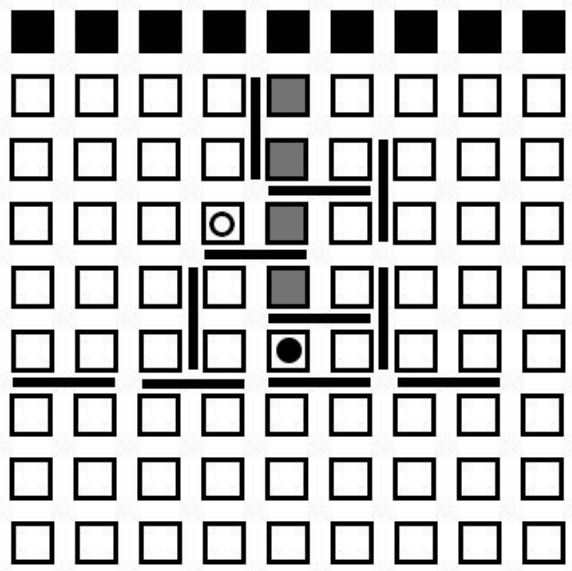


Figure 3.6: The squares that are part of the Manhattan distance for the black player are shaded gray. The Manhattan distance for the black player is 5.

Obviously, the Manhattan Distance, while a very speedy measurement, is not a very accurate measurement of the length of the shortest path from a pawn to its goal. The next

algorithm I used was *Dijkstra's algorithm*, a single-source shortest path graph algorithm. Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. For a detailed description of Dijkstra's algorithm, see [4].

Wait, Quoridor is a board game, not a graph! Well, actually, a Quoridor board can easily be represented by a graph. A Quoridor graph is built from a Quoridor board state from the perspective of one of the players by the algorithm in Figure 3.7. An example of a Quoridor graph is shown in Figure 3.8.

1	Create a vertex for each square
2	Create an undirected edge between each square and each of the square's cardinal neighbors
3	For each fence, remove the edges that the fence intersects
4	Remove the vertex that the opponent pawn occupies and the edges of that vertex, and create edges in the neighborhood of that vertex representing all of the possible legal jumps

Figure 3.7: Algorithm for turning a Quoridor board state into a Quoridor graph.

The *cardinal neighbors* of a square are those squares $+$ or $-$ a letter or number. For example, the cardinal neighbors of square `e5` are `d5, f5, e4, e6`. If a square is on an edge, it will have less than four cardinal neighbors. The only complicated part of the algorithm is the last step, which takes jumps into account for path creation.

Notice in Figure 3.8 that a Quoridor graph is not necessarily a completely connected graph.

The problem of finding the shortest path from the pawn to the goal is now transformed

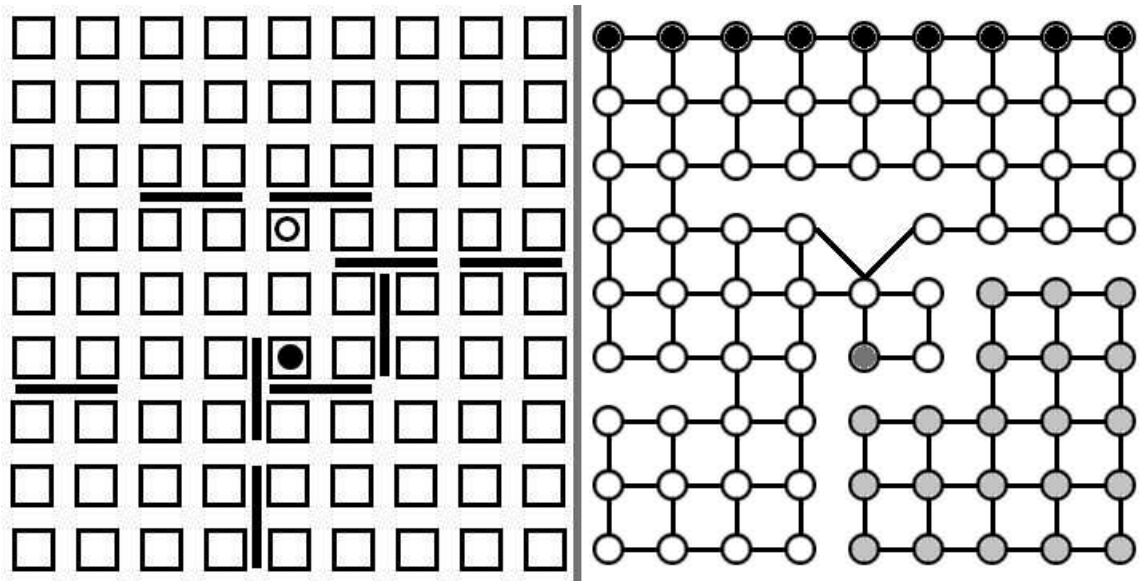


Figure 3.8: Sample Quoridor board state and corresponding Quoridor graph from the perspective of the black player. The vertex occupied by the black pawn is shaded a medium gray, and the goal is shaded black. Notice that the vertices shaded a light gray are not connected to the rest of the graph. Also notice that the vertex containing the white pawn is not present, and that the edges in that neighborhood reflect the possible jumps.

into the problem of finding the shortest path from the vertex representing the square occupied from the pawn to one of the vertices representing the goal.

Dijkstra's algorithm is ideal for use on a Quoridor graph, because it has optimal performance on sparse graphs (graphs with a small number of edges relative to the number of vertices) [4]. A Quoridor graph is indeed very sparse, considering that there are 81 vertices (9 rows of squares $\times$ 9 columns of squares) and that each vertex is connected to at most 5 other vertices (4 possible cardinal neighbors, 5 possible jump neighbors as seen in Figure 3.8).

Now to find the shortest path to the goal, simply run Dijkstra's algorithm using the vertex representing the square occupied by the pawn as the source, and choose the shortest of the resulting paths to each reachable vertex representing a goal square. Like the

Manhattan Distance, this value must be manipulated. This feature is trickier, as there is no obvious "maximum" value. I chose the maximum value to be 81, which is the number of squares on the board and thus the longest possible path. A better value, however, would be 73, since a pawn will never traverse more than one goal square, so the longest possible path should only include one of the goal squares. The value of the *shortest path* feature is $\frac{81-\text{length of the shortest path to goal}}{81}$. Since most of the time the shortest path to the goal will be a lot smaller than 81, there will not be a lot of variance in this feature. There is probably a better way to represent the shortest path feature.

Another useful model to apply to a Quoridor board state is to view it as a *Dirichlet problem*, which can be defined as follows: Let $S = D \cup B$ be a finite set of lattice points such that D is the set of *interior points* and B is the set of *boundary points*. A function f defined on S is *harmonic* if, for points (a,b) in D, it has the averaging property

$$f(a,b) = \frac{f(a+1,b) + f(a-1,b) + f(a,b+1) + f(a,b-1)}{4}$$

The problem of finding a harmonic function given its boundary values is called the Dirichlet problem [9].

By viewing each goal square as a boundary point with the value 1, and adding an invisible set of boundary points with the value 0 "behind" the row or column where the pawn starts the game, a Quoridor board state can be turned into a potential field. The pawn wants to move through the potential field towards the highest valued points (the goal). The problem is finding the values of the interior points (the board) so that the pawn knows which path to take. This can be done by finding a harmonic function. The value of this feature is then the value of the harmonic function at the point occupied by the pawn. This feature is already normalized, and a higher value means that the pawn is closer to the goal.

Finding the exact solution to a Dirichlet problem in two dimensions is not always a simple matter. One method for generating approximate solutions is the *method of relaxations* (Figure 3.9). The idea of this method is to iteratively average the interior points.

While the resulting function will not be perfectly harmonic, it will be more nearly harmonic than the previous iteration. For an example of applying the method of relaxations, see Figure 3.10 [9].

1	For each point $d \in D$, set the value of d to be the average of its neighbors.
2	Repeat step 1 until satisfied.

Figure 3.9: Method of relaxations.

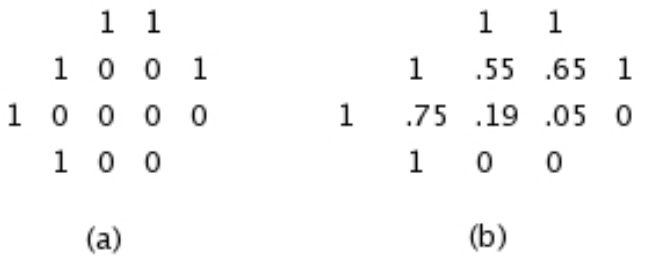


Figure 3.10: (a) All interior points are set to 0 and the boundary points are fixed to 1 and 0. (b) After one iteration of relaxation, moving from left to right and down to up replacing each value by the average of its neighbors.

An exact solution to a Dirichlet problem can be found by the method of *Markov chains*. A *finite Markov chain* is a special type of chance process that may be described informally as follows: we have a set $S = \{s_1, s_2, \ldots, s_r\}$ of *states* and a chance process that moves around through these states. When the process is in state s_i, it moves with probability P_{ij} to the state s_j. The transition probabilities P_{ij} are represented by an $r \times r$ matrix $\mathbf{P}$ called the *transition matrix*. To specify the chance process completely we must give, in addition to the transition matrix, a method for starting the process, i.e., the state in which the process starts [9].

After n steps, the probability that the process is in each of the possible states is provided by the matrix $\mathbf{P}$ raised to the nth power, or $\mathbf{P}^n$, in which entries $\mathbf{P}^n_{ij}$ represent the probability that the chain, started in state s_i, will, after n steps, be in state s_j. After a large number of steps, the probability of being in a state is independent of the starting state [9].

States that, once entered, cannot be left, are called *traps* or *absorbing states*. A Markov chain is called *absorbing* if it has at least one absorbing state and if, from any state, it is possible to reach at least one absorbing state. The states of an absorbing chain that are not traps are called *non-absorbing*. When an absorbing Markov chain is started in a non-absorbing state, it will eventually end up in an absorbing state. For non-absorbing state s_i and absorbing state s_j, we denote by B_{ij} the probability that the chain starting in s_i will end up in s_j. We denote by $\mathbf{B}$ the matrix with entries B_{ij} [9].

Assume now that $\mathbf{P}$ is an absorbing Markov chain and that there are u absorbing states and v non-absorbing states. We reorder the states so that the absorbing states come first and the non-absorbing states come last. Then our transition matrix has the canonical form:

$$\mathbf{P} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{R} & \mathbf{Q} \end{pmatrix}$$

Here $\mathbf{I}$ is a $u \times u$ identity matrix, and $\mathbf{0}$ is a $u \times v$ matrix with all entries 0 [9].

The matrix $\mathbf{N} = (\mathbf{I} - \mathbf{Q})^{-1}$ is called the *fundamental matrix* for the absorbing chain $\mathbf{P}$. The entry N_{ij} is the expected number of times that the chain will be in state s_j before absorption when it is started in s_i. $\mathbf{B} = \mathbf{NR}$ is the absorption probabilities matrix. For an absorbing chain $\mathbf{P}$, $\mathbf{P}^n$ will approach a matrix $\mathbf{P}^\infty$ of the form [9]:

$$\mathbf{P}^\infty = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{B} & \mathbf{Q} \end{pmatrix}$$

Now, going back to the original definition of a Dirichlet problem, let f be a function

with domain the state space of a Markov chain **P** such that for i in D

$$f(i) = \sum_j P_{ij} f(j)$$

Then f is a harmonic function for **P**. If we represent f as a column vector **f**, f is harmonic if and only if

$$\mathbf{Pf} = \mathbf{f}.$$

This implies that

$$\mathbf{P}^n\mathbf{f} = \mathbf{f}.$$

The vector **f** can be written as

$$\mathbf{f} = \begin{pmatrix} \mathbf{f}_B \\ \mathbf{f}_D \end{pmatrix},$$

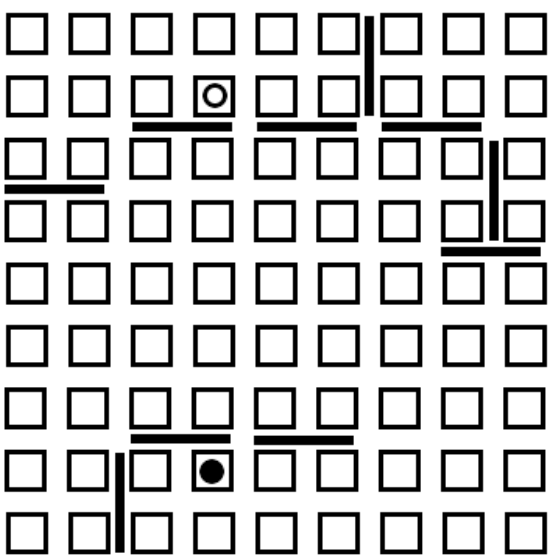
where $\mathbf{f}_B$ represents the values of f of the boundary points and $\mathbf{f}_D$ the values of the interior points. Then we have

$$\begin{pmatrix} \mathbf{f}_B \\ \mathbf{f}_D \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{B} & \mathbf{Q} \end{pmatrix} \begin{pmatrix} \mathbf{f}_B \\ \mathbf{f}_D \end{pmatrix}$$

and $\mathbf{f}_D = \mathbf{B}\mathbf{f}_B = (\mathbf{I} - \mathbf{Q})^{-1}\mathbf{R}\mathbf{f}_B$ [9].

To apply the method of Markov chains to a Quoridor board state, first consider the subset of squares that is reachable by the player. In other words, we are only going to use the connected component of the graph that includes the vertex the pawn is currently occupying. The goal squares become absorbing states, and the rest of the squares become non-absorbing states. We also need a set of absorbing states which are conceptually

"behind" the row or column that the pawn begins that game on. The absorbing states representing the goal have the value 1 in $\mathbf{f}_B$, and the other absorbing states have the value 0. The probability of moving from non-absorbing state s_i to any state s_j is 0 if the two states are not neighbors, otherwise it is $\frac{1}{degree(s_i)}$. The actual feature value is the value of $\mathbf{f}_D$ at the state representing the square occupied by the pawn. An example Quoridor board state and corresponding f values are shown in Figure 3.11.



(a)

1.0	1.0	1.0	1.0	1.0	1.0			
.91	.90	.96	.99	.99	1.0			
.82	.73	.48	.36	.30	.27	.25	.24	
.26	.29	.34	.31	.28	.25	.23	.24	
.24	.25	.27	.27	.25	.22	.20	.16	.15
.21	.22	.23	.23	.22	.20	.17	.14	.13
.17	.18	.21	.22	.20	.18	.13	.11	.10
.11	.12	.01	.01	.02	.03	.07	.07	.07
.06	.06	.01	.01	.01	.02	.03	.03	.03
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

(b)

Figure 3.11: (a) Sample Quoridor board state. (b) Corresponding harmonic function from the perspective of the black player with fences added for ease of comparison. Notice that only the connected component is assigned values, and also notice the added absorbing states with value 0. The value of this feature for the black player would be 0.01.

3.3.2 Other Features

Another feature I proposed was the distance between the pawns. This feature arose from the "creating a box" strategy, in which it is essential that both pawns be close together so

that they can be enclosed in the same labyrinth. Another situation where it is advantageous for the pawns to be close together is when one pawn is very close to its goal and the other pawn has not managed to advance very far. Of course, it is also easy to envision situations where a board state is bad for you if the pawns are close together. So it is not very likely that the *pawn distance* feature is a universal indicator of whether a player is winning or losing. This feature was also normalized by the formula $\frac{81-\text{pawn distance}}{81}$.

The number of fences a player still has in reserve is another obvious feature of Quoridor. Like the pawn distance, however, there are winning situations where the player has a lot of fences remaining, and there are winning situations where the player doesn't have any fences remaining. So this is also not necessarily a useful feature. The *number of fences* feature can be normalized by dividing by 10, the number of fences each player begins the game with.

The chess playing machine Deep Blue uses over 8000 features in its evaluation function, many of them describing highly specific patterns of pieces [18]. The level of expert knowledge in the game of Quoridor today is not nearly as advanced. The set of features that I came up with are the fairly obvious ones. No doubt, many more features could be proposed, for example, features that describe how strategically placed the fences are.

3.4 Implementation

To develop a Quoridor artificial player, I had to develop a search algorithm, evaluation function, and learning algorithm. For the search algorithm, I relied upon a standard iterative-deepening alpha-beta negamax search algorithm (see Section 2.4) with a few modifications. The first modification was to short-circuit the search if a 1-ply search revealed a winning terminal state, because there is no point in searching further in the tree if the agent is one move away from winning.

Another modification was the use of a *principal variation*, which is a list of (*state*, *move*, *value*) triplets representing one path through the tree. The function NEGAMAX-VALUE was modified to return a principal variation instead of just the value. The principal variation returned represents the optimal path from *state* to the leaf node of the search. Using a principal variation served several functions. The first was to prevent loops in the game tree, e.g., returning to a previous state. With a principal variation, a state can check to see if any of its child states are identical to any of its parent states. If so, the state can prune those children. The second function was for move ordering. On iteration i of the iterative-deepening algorithm, if a given state is in the principal variation of iteration $i-1$, the first move we examine is the one in that triplet. By examining the move previously found to be the optimal move for this state first, then the alpha-beta pruning is more likely to be effective.

The last modification was the ability to define a cutoff point for the search if it was determined that the search would be unable to complete the next iterative deepening iteration in a reasonable amount of time. For example, for the time limits I imposed during the learning process, I found that the search could not possibly complete the iteration at 3-ply if the branching factor was near its maximum. Since the result of the last (uncompleted) iteration is disregarded, any time spent past the 2-ply iteration was effectively wasted. Therefore, to save time, I imposed a cutoff when the last iteration had examined a certain number of leaf nodes (approximately the number of leaf nodes of a 2-ply tree, 133^2).

For the evaluation function, I chose 10 features. Table 3.1 lists the features, their descriptions, and whether I hypothesized that the weights would turn out to be positive, negative, or zero.

I chose to use all of the features discussed in Section 3.1, except for the method of relaxation. The reason for excluding the method of relaxation is that I found that calculating the feature value took significantly longer than calculating the Markov chain feature, and since the method of relaxation is an approximation to the Markov chain method, it would

Table 3.1: Features chosen for the Quoridor agent.

Feature Number	**Feature Name**	**Feature Description**	**Weight Hypothesis**
1	Shortest Path Player (SPP)	length of Dijktra's shortest path for the player	+
2	Shortest Path Opponent (SPO)	length of Dijktra's shortest path for the opponent	−
3	Markov Chain Player (MCP)	Markov chain value for the player	+
4	Markov Chain Opponent (MCO)	Markov chain value for the opponent	−
5	Manhattan Distance Player (MDP)	Manhattan distance for the player	+
6	Manhattan Distance Opponent (MDO)	Manhattan distance for the opponent	−
7	Pawn Distance (PD)	pawn distance	0
8	Goal Side Player (GSP)	goal side for the player	+
9	Goal Side Opponent (GSO)	goal side for the opponent	−
10	Number Fences Player (NFP)	number of fences of the player	0

only be useful if it was faster to calculate. Of course, it may be possible to implement the method of relaxations more efficiently. All of the chosen features can be calculated for both the player doing the evaluation and from the perspective of the opponent, except for pawn distance, which is independent of which player is doing the evaluation. I chose not to include the number of fences held by the opponent as a feature, because no move by the player can influence that value. The reason that I hypothesized that NFP would tend to a weight of zero is that fences can be viewed as potential energy. Having a lot of potential energy is good, but it can also be translated into kinetic energy without losing much value. In Quoridor terms, holding fences in reserve is good because then you have a lot of flexibility later in the game, but if you use your fences wisely then they do not lose

their value simply because they have been played.

For the learning algorithm, I chose a variant of a genetic algorithm (see Section 3.2). Usually there are only two main components of most genetic algorithms that are problem-dependent: the problem encoding and the fitness evaluation function [22]. For the problem of learning weights, a weight vector is considered to be a chromosome, so that each locus is a float value. The fitness evaluation function is the number of games that a chromosome wins against the other chromosomes of the population.

In my genetic algorithm, a population was created with the algorithm shown in Figure 3.12. The function RANDOM(a, b) returns a randomly generated float in the range $[a, b]$. POPSIZE is a constant describing the number of chromosomes to be created for the population. C_i is the ith chromosome of the population, and $C_i[j]$ is the jth locus (weight for feature j) of that chromosome. HASFEAT is the probability that the chromosome has a non-zero weight for each feature. I set HASFEAT to 0.3 for all of my populations. If a chromosome has a zero weight for a feature, then it does not waste time calculating the value of that feature.

```
function CREATE-POPULATION()
   for i = 1 ... POPSIZE do
      for j = 0 ... 9 do
         if (RANDOM(0, 1) <= HASFEAT)
         then C_i[j] ← RANDOM(-1, 1)
         else C_i[j] ← 0
```

Figure 3.12: Population creation algorithm.

To determine the fitness of each chromosome, each chromosome plays a *round* against every other chromosome. A round consists of two games, with the chromosomes alternating being the black player. This was done to avoid any bias for which chromosome moves first. The fitness of a chromosome is then simply the number of games it wins. During

a game, each player gets TIMELIMIT seconds to decide on a move. I quickly discovered during this process that, if left to their own devices, agents could make a game last indefinitely by, for example, repeatedly moving back and forth between two squares. Therefore it was necessary to impose a limit, MOVECAP, on the maximum number of moves a game could last. I hypothesized that 164 would be a reasonable value for MOVECAP, as this allows each player 10 moves to place its fences and 74 moves to get its pawn to the goal. If a game reached MOVECAP moves before ending, the game was declared a draw, and neither player won. So, to increase its fitness, a chromosome would have to not only win a game, but do so in a limited number of moves.

Once all $n(n-1)$ games were finished, where n is the number of chromosomes in the population, the chromosomes were ranked by their fitness. One danger of using recombination and mutation is that the best solution (chromosome) found so far could be modified irrevocably and lost. Therefore, I directly copied a few of the top-ranked chromosomes into the next population. The percentage of the top-ranked chromosomes that were copied was determined by the ELITISM variable. To fill the remaining slots of the next population, I used the SPAWN algorithm (Figure 3.13). The argument C_p is the population of generation p, and the argument f is an array of corresponding fitness values. The argument n is the number of offspring required. SELECT(C_p, f) returns a chromosome selected from C_p using the roulette wheel method. MUTATION is the probability that a weight will be mutated. LOSEFEAT is the probability that if a weight is non-zero it will be changed to zero. MUTFEAT is the maximum amount that a weight can be mutated by.

For example, if POPSIZE was 10 and ELITISM was 0.2, then the first and second top-ranked chromosomes were each copied once into the next generation. The SPAWN algorithm would then be used to create the remaining 8 chromosomes required.

For each learning population that was created, I also created a companion *test population*, which would never change. Every few generations of the genetic algorithm learning process, each chromosome of the learning population played a round against each chro-

```
function SPAWN(C_p, f, n) returns a population
    for i = 0...n-1 do
        mother ← SELECT(C_p, f)
        father ← SELECT(C_p, f)
        for j = 0...9 do
            if (RANDOM(0, 1) ≥ 0.5)
            then C_{p+1,i}[j] ← mother[j]
            else C_{p+1,i}[j] ← father[j]

            if (RANDOM(0, 1) ≥ MUTATION) then
                if (C_{p+1,i}[j] is zero)
                then C_{p+1,i}[j] ← RANDOM(-1, 1)
                else if (RANDOM(0, 1) ≥ LOSEFEAT)
                then C_{p+1,i}[j] ← 0
                else C_{p+1,i}[j] ← RANDOM(-1, 1) × MUTFEAT
    return C_{p+1}
```

Figure 3.13: Algorithm to create offspring for the next generation.

mosome of the test population. The total number of games won by the learning population against the test population was the *population fitness*. If the population fitness increased over time, that would be an indicator that the population was improving.

There were still a number of variables to be specified, so I ran 11 separate populations with different values for the variables shown in Table 3.2. Each population ran on a different CPU and was named for the machine and processor number it used. Time was a limiting factor, so I tried to make the relevant variables (POPSIZE, MOVECAP, TIMELIMIT) as small as possible. For example, if a population set POPSIZE to 20, MOVECAP to 164, and TIMELIMIT to 15, then a single generation of the learning algorithm could take up to 10.8 days ((20×19) games $\times$ 164 moves/game $\times$ 15 seconds/move).

Table 3.2: Variable values for the different populations.

Population Number	**Population Name**	**POPSIZE**	**MOVECAP**	**TIMELIMIT** (seconds)	**MUTATION**
1	digamma1	20	150	12	0.01
2	digamma2	10	150	12	0.05
3	heta1	20	182	10	0.01
4	heta2	15	100	8	0.05
5	omega1	10	100	8	0.01
6	omicron1	10	120	10	0.05
7	psi1	25	120	8	0.01
8	psi2	10	80	7	0.1
9	rho1	15	100	20	0.01
10	rho2	15	100	20	0.1
11	xi1	20	120	15	0.01
Population Number	**Population Name**	**LOSEFEAT**	**MUTFEAT**	**ELITISM**	
1	digamma1	0.1	0.1	0.2	
2	digamma2	0.1	0.3	0.1	
3	heta1	0.1	0.1	0.2	
4	heta2	0.2	0.3	0.1	
5	omega1	0.1	0.1	0.2	
6	omicron1	0.1	0.3	0.1	
7	psi1	0.1	0.1	0.2	
8	psi2	0.2	0.4	0.1	
9	rho1	0.1	0.1	0.2	
10	rho2	0.2	0.5	0.1	
11	xi1	0.1	0.1	0.2	

Chapter 4

Results

The learning process ran for a few weeks until a deadline was reached. During this time, some populations got more CPU time (due to machine downtime, human error, etc.) than others, and so the work spent learning was not equal for each population. Also, the populations ran on different machines with varying speeds. Figure 4.1 shows the change in population fitness (see Section 3.4) over the learning process. The X-axis is the generation number of the genetic algorithm on a logarithmic scale, and the Y-axis is the percentage of games won against the test population. The population fitness of most populations seems to have reached an early peak and then dropped or leveled off.

I next took a look at the individual fitness for each chromosome in every population. I chose a chromosome to represent each population, picking the chromosome with the highest fitness in its population. These chromosomes formed a new champion population. The weights for each feature of these chosen chromosomes, named for the population they represent, are shown in Table 4.1 and are plotted in Figure 4.2. The description of each feature abbreviation and number can be found in Section 3.4. Figure 4.2 also shows the average value of each weight across the champion population.

I wanted to determine which chromosome of the champion population had the highest

Table 4.1: Champion population weights.

#	Chromosome	SPP	SPO	MCP	MCO	MDP
1	digamma1	0.250	0.000	0.290	0.000	0.378
2	digamma2	0.000	0.331	0.606	-0.480	1.165
3	heta1	0.945	0.000	0.000	0.000	0.000
4	heta2	0.421	0.386	0.000	0.000	0.095
5	omega1	-0.452	-0.462	-0.711	0.393	0.627
6	omicron1	-0.396	-0.713	0.551	0.000	0.504
7	psi1	0.747	0.096	0.000	0.000	0.000
8	psi2	1.299	-0.258	1.303	0.000	0.000
9	rho1	0.993	0.083	0.000	0.000	0.000
10	rho2	1.187	-0.451	0.278	0.000	0.000
11	xi1	0.131	0.362	0.000	0.000	0.000
#	Chromosome	MDO	PD	GSP	GSO	NFP
1	digamma1	0.034	-0.974	0.071	0.000	0.000
2	digamma2	-0.300	0.903	0.522	-0.504	-0.265
3	heta1	0.000	-0.846	0.245	0.000	0.792
4	heta2	0.097	-0.201	0.757	-0.448	0.534
5	omega1	-0.908	0.000	0.000	0.000	-0.078
6	omicron1	0.000	0.392	0.000	-0.322	0.000
7	psi1	-0.792	0.000	0.000	0.000	0.327
8	psi2	0.497	-0.699	0.487	-0.346	0.555
9	rho1	0.000	-0.460	0.000	0.000	0.000
10	rho2	0.000	0.000	1.061	0.000	0.592
11	xi1	0.000	0.000	0.000	0.000	1.000

fitness, but, since each population was developed using different MOVECAP and TIMELIMIT parameters, the method for determining comparative fitness was not straightforward. I ended up doing two different competitions, a “fast” competition and a “slow” competition. In both competitions, the MOVECAP was set to 182, which was the largest value used in any of the populations. In the fast competition, the TIMELIMIT was relatively short (8 seconds), and in the slow competition, the TIMELIMIT was relatively long (15 seconds). The results of the fast competition are shown in Figure 4.3, where the X-axis is the chromosome number (see Table 4.1) of the champion population, and the Y-axis is

the fitness of each chromosome (number of games won against the other chromosomes). The results of the slow competition are shown in Figure 4.4. In both competitions, chromosome 7 (psi1) had the highest fitness and was declared the champion chromosome of all populations.

During the fast and slow competition, a total of 220 games, I tracked some statistics. The average game length was 91.1 turns, the average branching factor was 60.4, and the average depth reached by the search was 4.1. The average depth was the same for both the fast and slow competitions, suggesting that the difference in the time limits was not significant enough to allow for a deeper search.

While these results tell us the relative progress of the populations and which chromosome is the most fit, they do not tell us how "well" the Quoridor agent plays. For a more qualitative analysis of the agent, Joseph Farfel and I each played the champion chromosome, number 7 (psi1). We both easily beat it using both a 15 and 20 second turn time limit, and both concluded that it is unlikely that any human player would have trouble winning against it. While chromosome 7 is able to effectively place fences to lengthen the opponent's path, it is not very effective at moving towards the goal, especially near the end of the game, and does not play fences defensively. I also played the "average" chromosome of the champion population (see Figure 4.2), and easily beat it as well.

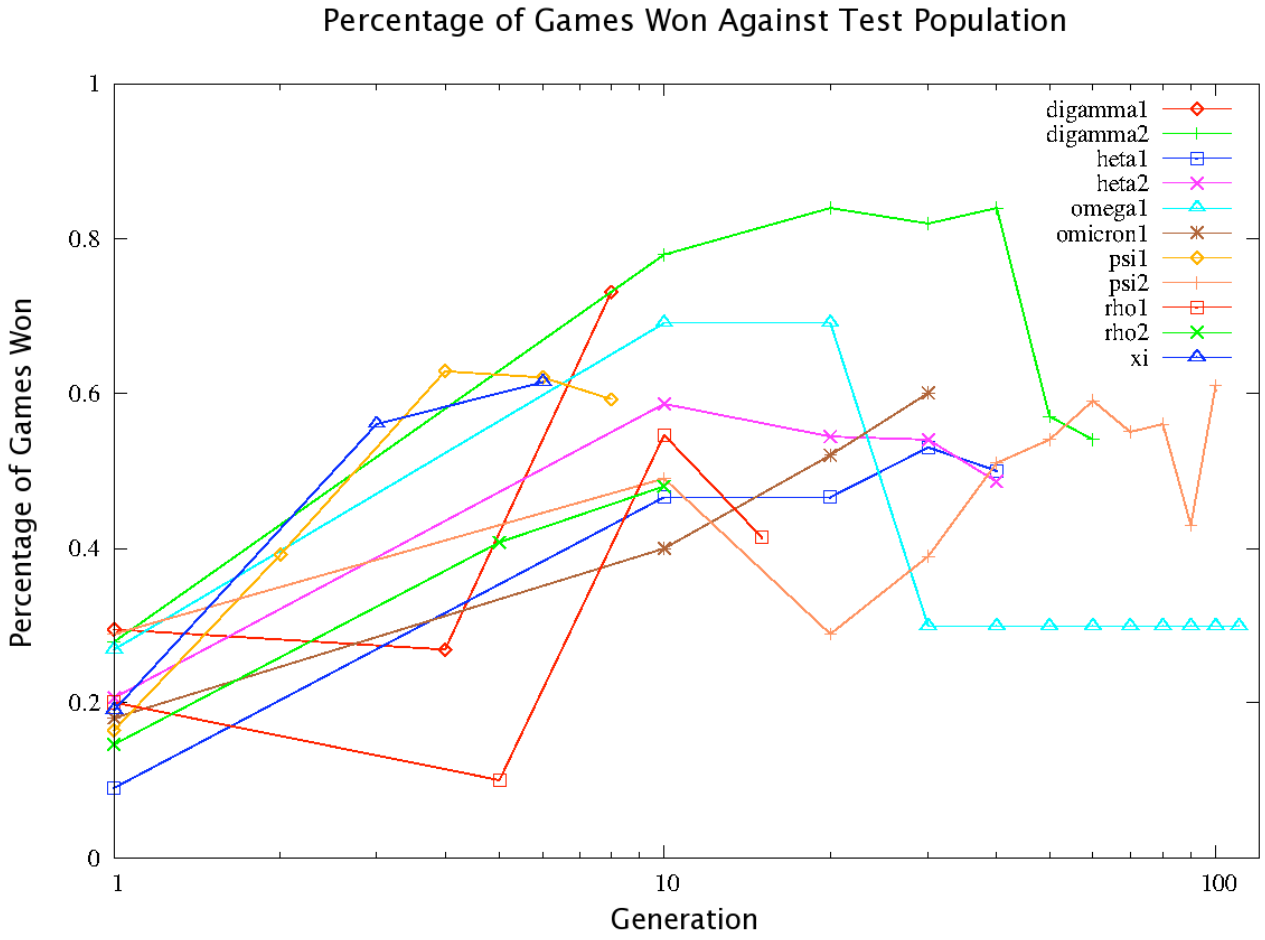


Figure 4.1: Population fitness.

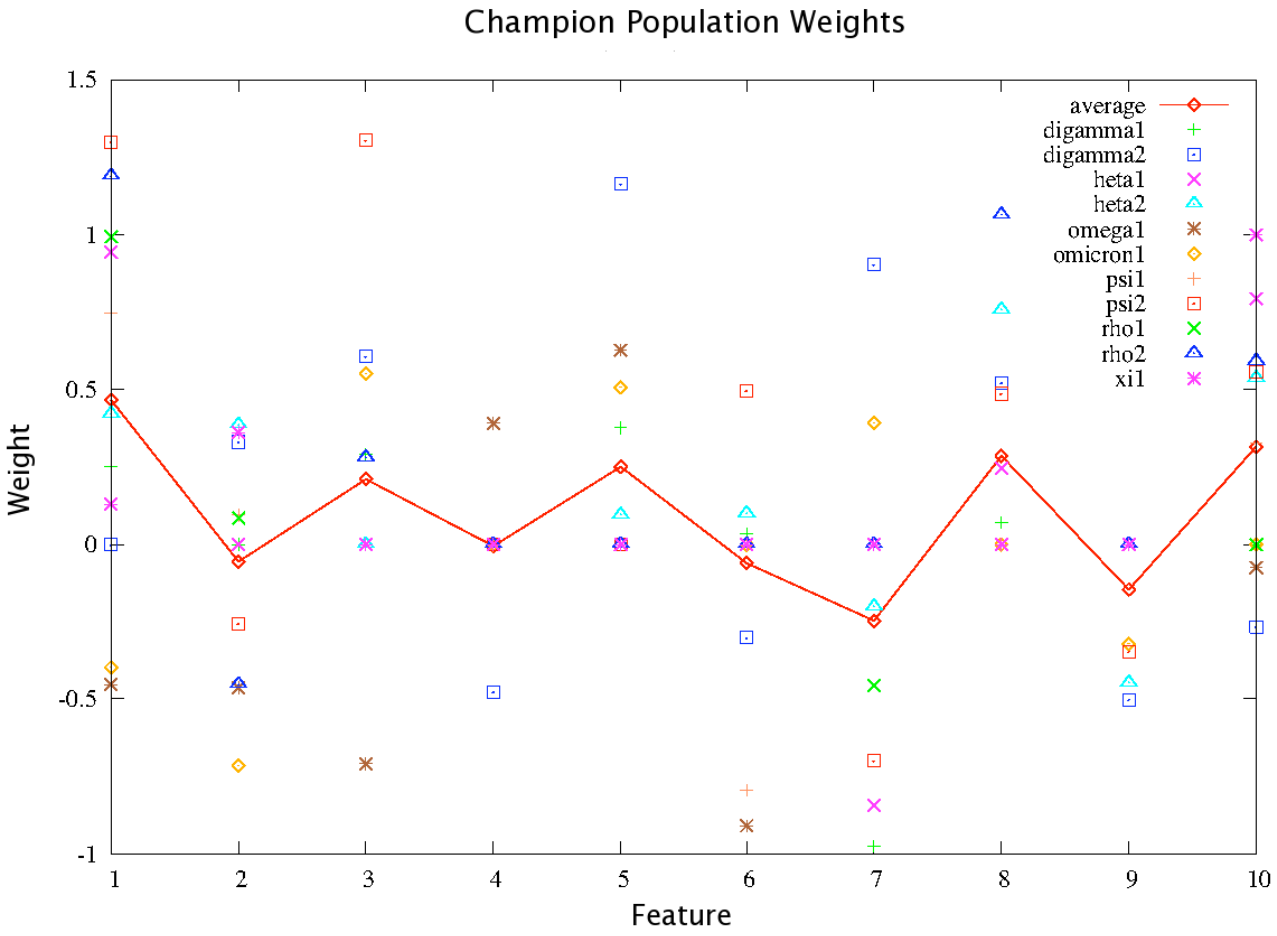


Figure 4.2: Champion population weights, and the average weights of the champion population.

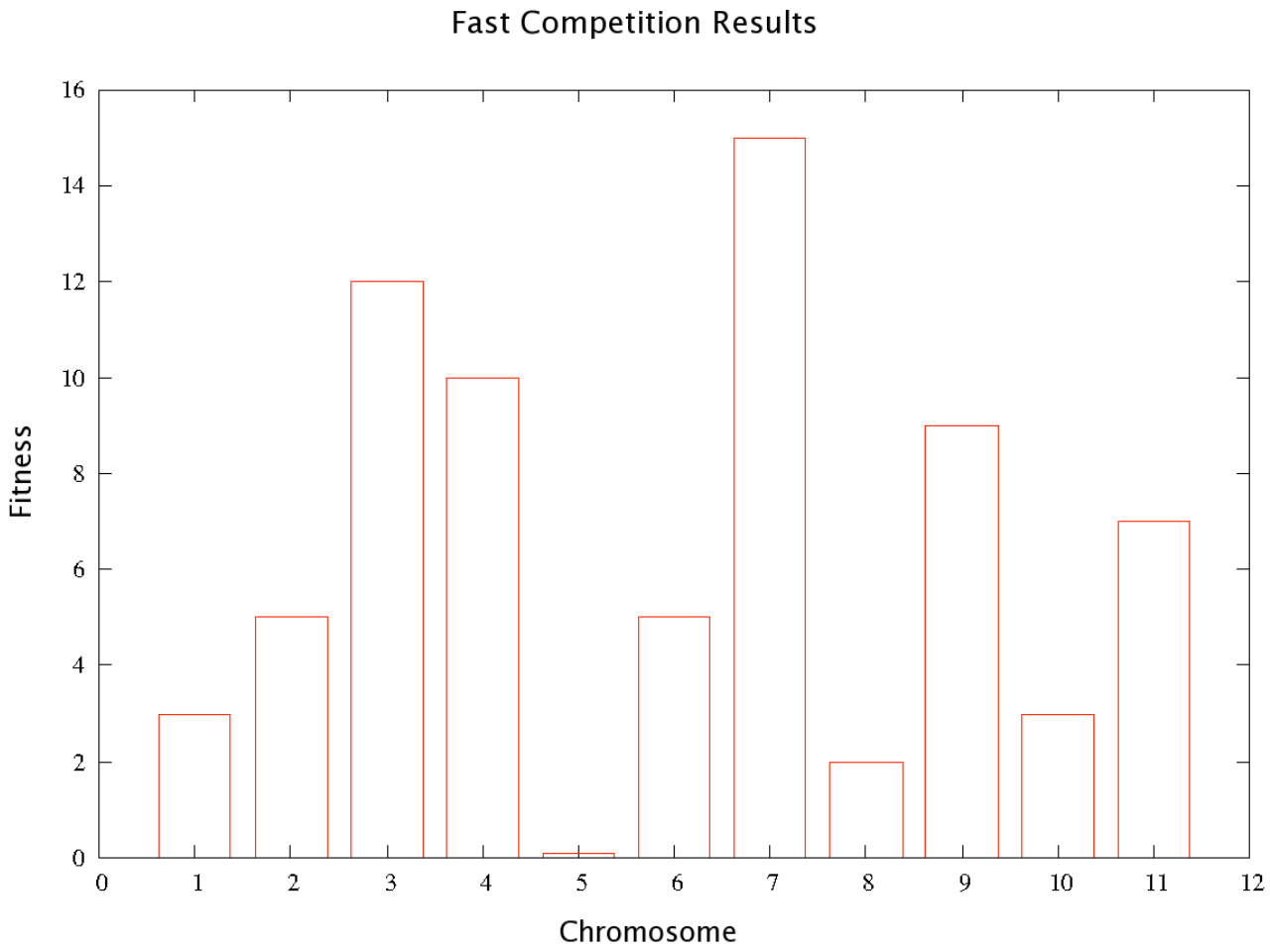


Figure 4.3: Fast competition results.

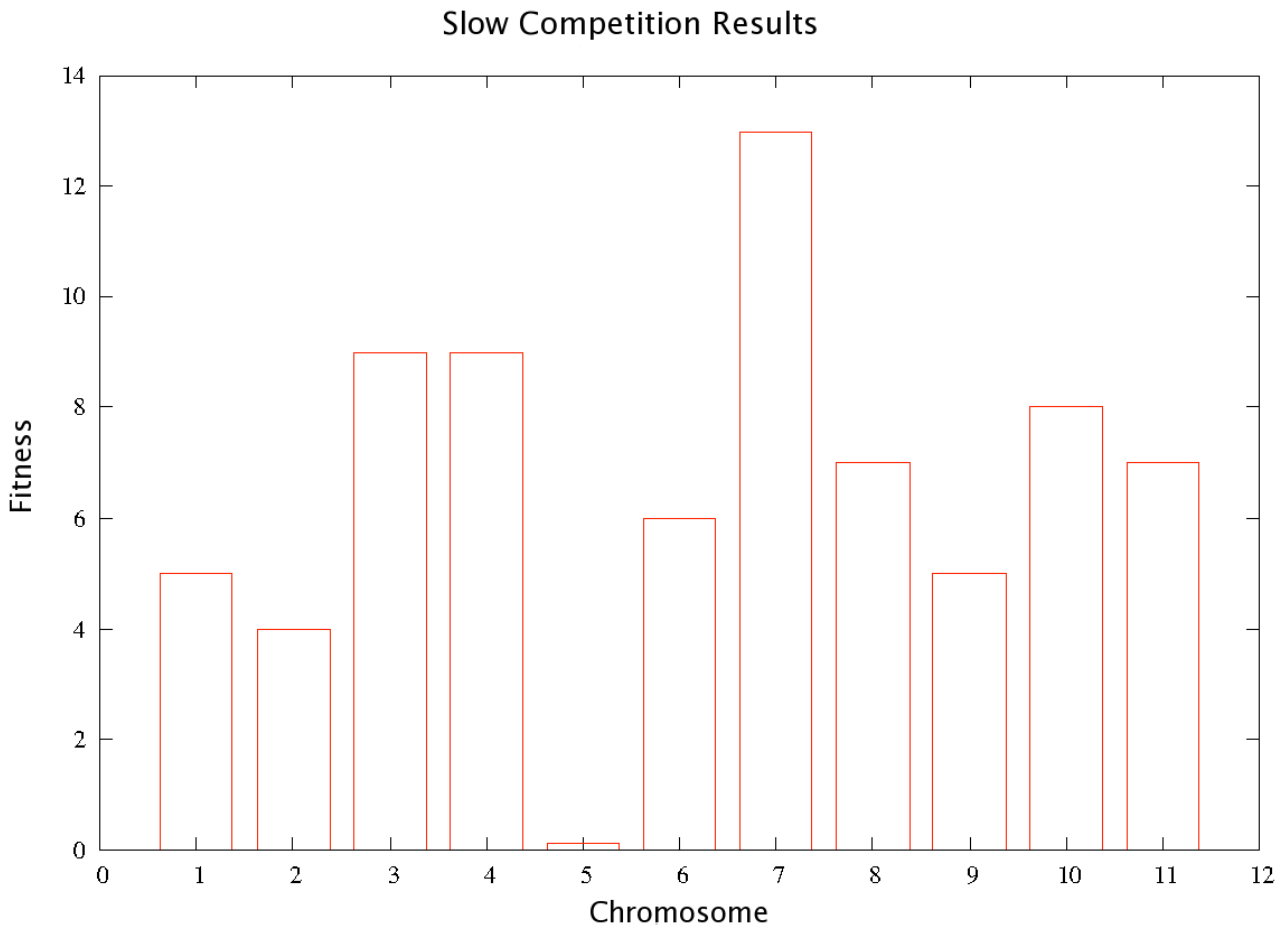


Figure 4.4: Slow competition results.

Chapter 5

Conclusions

The goals of this research project were to analyze the characteristics of the game of Quoridor, propose features for use in an evaluation function, use a learning algorithm to optimize the weights of a linear evaluation function, and evaluate the artificial player produced by this method.

As a game, Quoridor can be characterized as a deterministic, sequential, partizan, loopy, two-player, zero-sum game of perfect information with a restricted outcome. The game tree of Quoridor has a maximum branching factor of 133 (128 possible fence locations and 5 possible pawn move locations), and a minimum branching factor of 1. Over the course of 220 agent-agent games, I calculated the average branching factor to be 60. This is still significantly higher than the average branching factor of chess (35), but significantly lower than the average branching factor of Go (200) [3], or that of Amazons (500) [14]. Also, the average game length of these agent-agent games was 91 turns.

I proposed 10 features of Quoridor: Shortest Path Player, Shortest Path Opponent, Markov Chain Player, Markov Chain Opponent, Manhattan Distance Player, Manhattan Distance Opponent, Pawn Distance, Goal Side Player, Goal Side Opponent, and Number Fences Player. These features are discussed in Section 3.3.

How effective was the genetic algorithm I chose as a weight learning function? Based on my results, my algorithm is probably not the optimal algorithm for this problem. While the population fitnesses did immediately increase, they then peaked and tapered off. It is possible that more generations were needed for the genetic algorithm to be effective. There was a huge spread in the values of weights between the different populations, so the learning algorithm did not converge on a single optimum solution.

Something strange happened with population 5: after generation 5, the population fitness dropped to 0.3 and thereafter didn't change at all. Also, the champion of population 5 won zero games against the other champions. This may reveal a flaw in the learning algorithm: population 5 became nearly homogeneous, meaning that there was very little variation in the population. Without variation in the population or a high mutation rate, a genetic algorithm cannot generate new solutions. One drawback of genetic algorithms is possible early convergence on a solution.

We theorized that having different TIMELIMIT and MOVECAP parameters might produce populations that were better suited for "fast" or "slow" games. There was a difference in the performance of the champion chromosomes in the two competitions, for example, chromosomes 1, 8 and 10 won more games in the slow competition; however, chromosome 7 performed consistently well for both time limits.

The champion chromosome's population (7) had relatively low mutation rates compared to the other populations. Population 7 also had a short time limit, but was on one of the faster machines. Population 7 had a POPSIZE of 25 , while the worst population (5) had a POPSIZE of 10. Populations 3 and 4, which did well in the competitions, had POPSIZEs of 20 and 15, respectively. Perhaps large population sizes (≥ 25) would make this algorithm more effective.

Finally, a major drawback of the learning approach I used in this project is the sheer amount of time it needs. Another learning algorithm might require fewer games to be

played, and thus be more efficient.

The resulting artificial player (champion chromosome 7), given the turn time limit and machines used, is not able to beat human players. The goal of future research in this area should be to produce a Quoridor artificial player that can play competitively with humans.

Although the branching factor is quite large, it may still be possible to use the brute force search approach to develop an effective playing program. Further research in this area might investigate a more efficient search algorithm and pruning heuristics. Features that focus on the strategic placement of fences could be proposed. Improving the accuracy of the evaluation function could also dramatically increase the ability of the artificial player. A different weight learning algorithm could be used, perhaps with a nonlinear evaluation function. The server-client architecture of my code could be utilized to allow the artificial player to learn from playing humans over the internet. Without a deadline, much longer time limits for both learning and playing would improve the artificial player.

References

[1] Eric Baum, Dan Boneh, and Charles Garrett. Where genetic algorithms excel. *Evolutionary Computation*, 9:93–124, 2001.

[2] Yngvi Bjornsson. *Selective Depth-First Game-Tree Search*. PhD thesis, University of Alberta, 2002.

[3] Jay Burmeister. *Studies in Human and Computer Go: Assessing the Game of Go as a Research Domain for Cognitive Science*. PhD thesis, The University of Queensland, 2000.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2002.

[5] Mohammed Daoud, Nawwaf Kharma, Ali Haidar, and Julius Popoola. Ayo, the Awari player, or how better representation trumps deeper search. In *Congress on Evolutionary Computation*, 2004.

[6] James Davis and Graham Kendall. An investigation, using co-evolution, to evolve an Awari player. In *Congress on Evolutionary Computation*, pages 1408–1413, 2002.

[7] Richard Dawkins. *Climbing Mount Improbable*. W. W. Norton & Company, 1996.

[8] K. Deb, A. Anand, and D. Joshi. A computationally efficient evolutionary algorithm for real-parameter optimization. *Evolutionary Computation*, 10:371–395, 2002.

[9] Peter G. Doyle and J. Laurie Snell. Random walks and electric networks. The Mathematical Association of America, 2000.

[10] A. E. Eiben and G. Rudolph. Theory of evolutionary algorithms: A bird's eye view. *Theoretical Computer Science*, 229:3–9, 1999.

[11] Stephanie Forrest. Genetic algorithms. *ACM Computing Surveys*, 28:77–80, 1996.

[12] Aviezri S. Fraenkel and Ofer Rahat. Infinite cyclic impartial games. *Lecture Notes in Computer Science*, 1558:212–221, 1999.

[13] Graham Kendall and Glenn Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Congress on Evolutionary Computation*, pages 995–1002, 2001.

[14] Qian Liang. The evolution of Mulan: Some studies in game-tree pruning and evaluation functions in the game of Amazons. Master's thesis, University of New Mexico, 2003.

[15] Quinn McDermid, Anand Patil, and Touran Raguimov. Applying genetic algorithms to quoridor game search trees for next-move selection. `http://www.math.yorku.ca/Who/Guests/raguimov/trNew/html/work-software.html`, 2003.

[16] David Moews. Loopy games and Go. In *Games of No Chance: MSRI Publications, Volume 29*, pages 259–272. Cambridge University Press, 1996.

[17] Elaine Rich. *Artificial Intelligence*. McGraw-Hill, 1983.

[18] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., 2003.

[19] Dierk Schleicher and Michael Stoll. An introduction to Conway's Games and Numbers. `http://www.faculty.iu-bremen.de/stoll/schrift.html`, 2002.

[20] Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal difference learning of position evaluation in the game of Go. In J.D. Cowan et al., editor, *Advances in Neural Information Processing*, pages 817–824. Morgan Kaufmann, 1994.

[21] W.-Y. Lin T.-P. Hong, K.-Y. Huang. Applying genetic algorithms to game search trees. *Soft Computing*, 6:277–283, 2002.

[22] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[23] Darrell Whitley. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Journal of Information and Software Technology*, 43:817–831, 2001.