A Quoridor-playing Agent

P.J.C. Mertens

June 21, 2006

Abstract

This paper deals with the construction of a Quoridor-playing software agent. Because Quoridor is a rather new game, research about the game is still on a low level. Therefore the complexities of the game are calculated, whereafter depending on the complexities a suitable algorithm is chosen as a base for the Quoridor-playing agent. This algorithm is extended with some basic heuristics. Though these heuristics do not lead to spectacular results a better insight in Quoridor is gained from these heuristics.

1 Introduction

Since the early beginning men played games for practice of hunting and fighting skills. Later on games were being played for fun. Ever since people always want to be the best in these games, to gain respect. For example, in medieval times warlords participated in sword fighting to show both their strength and courage. In times where intelligence gained importance over power, games like chess got more popular to show mental skills. The invention of the micro-processor changed things in game-playing competitions. Some people did not want to be the best themselves anymore but wanted to create a software agent which would beat the best human player in the world or even solve the game. Today many games are being studied or studying has already stopped because the game is solved. One game that is not solved and has hardly been studied so far is Quoridor.

Quoridor is a 2-player board game, played on a 9x9 board. Each player has one pawn and 10 fences. At each turn, the player has to choose either to:

1. move his pawn to one of the neighboring squares.
2. place a fence on the board to facilitate his progress or to impede that of his opponent.

Quoridor was invented in 1997 by Gigamic. Compared to well-known games like chess and go, Quoridor is a relatively new game. Not much investigation is done and few specific information can be found about winning strategies. Research about strategies will be done in this thesis. The problem statement can be formulated as follows:

> What algorithms and heuristics can be used to develop a strong Quoridor-playing agent?

This leads to the following research questions:

1. What is Quoridor's state-space complexity and game-tree complexity?
2. Depending on these complexities, which algorithms are the most promising to develop a software agent that is capable of playing Quoridor?
3. What adjustments and extensions of these algorithms make the agent more advanced?

The reminder of this paper is structured as follows. Section 2 describes the little amount of research that is already done on this topic. This section also includes the pre-investigation. Some complexities of the game are measured here and depending on these complexities the type of algorithm that will be used is chosen. The experimental setup is described in section 3. Implementation choices are also being explained here. In section 4 the results of the tests performed are discussed. Section 5 gives the results of the experiments. Finally, conclusions are drawn and possible future work is discussed in section 6.

2 Background

First the rules of Quoridor will be specified. Then we will investigate the game's complexities, and an answer to the first and second research question will be given. Thereafter the related research will be described.

2.1 Rules of the game

The start position is depicted in figure 1. The objective of the game is to be the first to reach the other side of the 9x9 board. Each player starts at the center of his base line. A draw will determine who starts.

Each player in turn chooses to move his pawn or to put one of his 10 fences on the board. When a player runs out of fences, the player must move his pawn. The pawns are moved one square at a time, horizontally or

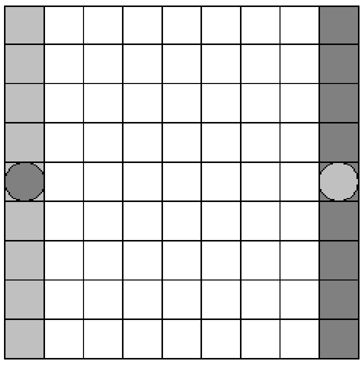

Figure 1: Quoridor board at the start.

vertically, forwards or backwards. The pawns must get around the fences, jumps are not allowed. The fences must be placed between 2 sets of 2 squares. They can be used to facilitate the players progress or to impede that of the opponent. However, at least one access to the goal line must always be left open. When two pawns face each other on the neighboring squares which are not separated by a fence, the player whose turn it is can jump the opponent's pawn (and place himself behind him), thus advancing an extra square (see figure 2).

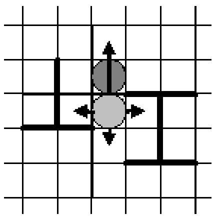

Figure 2: Allowed moves of the lower pawn.

If there is a fence behind the opposing pawn, the player can place his pawn to the left or the right of the other pawn (see figure 3), unless there is a fence beside the opposing pawn (see figure 4). The first player who reaches one of the 9 squares of his opponent's base line is the winner.

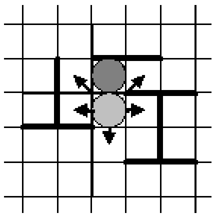

Figure 3: Allowed moves of the lower pawn.

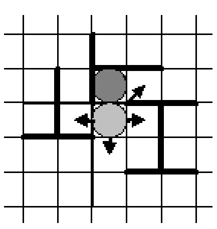

Figure 4: Allowed moves of the lower pawn.

2.2 Complexity of the game

Before developing a Quoridor-playing agent, few things about the game have to be known to start investigating in the right direction. Two things we want to know are the state-space and the game-tree complexity. The state-space complexity is the number of different possible positions that may arise in the game. The game-tree complexity is the size of the game tree, i.e., the total number of possible games that can be played. The game can then be mapped on one of four categories. The categories are shown in figure 5.

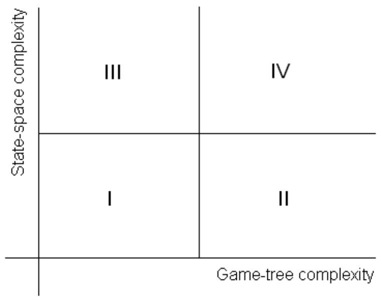

Figure 5: Categories of complexity.

The first category exists of solvable games like tic-tac-toe. These games have both a small game-tree and state-space complexity. Games in category II are games with a high game-tree complexity but a small state-space complexity. Brute-force algorithms are commonly used for this kind of games [9]. This is exactly the opposite of games in category III where the state space is too large to use brute-force methods but the game-tree complexity small enough to perform tree search. Most algorithms in this category are knowledge based. In category IV are games like chess and Go, with both a high state-space and game-tree complexity and therefore hard to master.

The state-space complexity is the number of possible positions (states) of the game. In Quoridor this is the number of ways to place the pawns multiplied by the number of ways to place the fences. However, since the number of illegal positions is hard to calculate, an upper bound will be estimated. There are 81 squares to place the 1st pawn on, 80 squares are left to place the second. So the total number of positions S_p with 2 pawns, disregarding fences, is given by eq. (1).

$$S_p = 81 * 80 = 6480 \tag{1}$$

To calculate the total number of states obtained by the fences, the number of ways to put one fence on the board has to be known. Since each fence has a length of 2 squares, there are 8 ways to place a fence in one row. Given that there are 8 rows, there are 64 positions to put a fence horizontally on the board. And because there are as much rows as columns, one fence can be put on 128 (64+64) different ways. But one fence occupies 4 fence positions (with exception to squares on the border). This can be seen in figure 6.

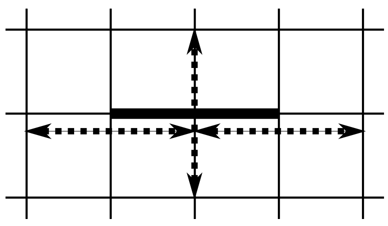

Figure 6: Occupation by a fence.

So the total possible number of positions of fences S_f can be estimated by equation (2).

$$S_f = \sum_{i=0}^{20} \prod_{j=0}^{i} (128 - 4i) = 6.1582\ 10^{38} \tag{2}$$

To get an estimation of the size of the state space, this number has to be multiplied with the number of pawn positions S_p, so the total state-space complexity S is given by equation (3).

$$S = S_p * S_f = 6480 * 6.1582\ 10^{38} = 3.9905\ 10^{42} \tag{3}$$

The game-tree complexity is estimated by raising the average branching factor by the power of the average number of plies. The branching factor is the number of branches a node in the game tree has. A ply is a move by one player, so the total number of plies is the sum of the number of steps made by both players together. According to Glendenning [2] the average branching factor can be estimated as 60.4. Also according to Glendenning, the average game length is 91.1. Now the game-tree complexity G can be estimated by equation (4).

$$G = 60.4^{91.1} = 1.7884\ 10^{162} \tag{4}$$

Now we have found the answer to the first research question. The state-space and game-tree complexity of Quoridor can be compared with complexities of well-known games, see table 1 [10]. From table 1 we can conclude that Quoridor has a similar state-space complexity as Chess and even a higher game-tree complexity, hence Quoridor belongs to the difficult games of category IV.

Game	log(state-space)	log(game-tree)
Tic-tac-toe	3	5
Nine Men's Morris	10	50
Awari/Oware	16	32
Pentominoes	12	18
Connect Four	14	21
Checkers	33	50
Lines of Action	24	56
Othello	28	58
Backgammon	20	144
Quoridor	42	162
Chess	46	123
Xiangqi	52	150
Arimaa	42	190
Shogi	71	226
Connect6	172	140
Go	172	360

Table 1: Game complexities.

2.3 Algorithms and Heuristics

In the previous section, the complexity of the game was measured. Now the object, as stated in research question 2, is to find algorithms that fit well, given the complexities. An algorithm that is often used is MiniMax search with Alpha-Beta pruning [7]. However, the game tree is too large to perform a MiniMax search all the way down to the leaves of the game tree. Yet, the algorithm is not useless. The problem of the large game tree can be tackled by limiting the depth of the MiniMax search. When this is done, a function to determine the value of a position has to be used. This kind of functions are called evaluation functions. The value returned by the evaluation function is the sum of weighted values obtained by several evaluation features (see eq. (5))[7].

$$Eval(s) = w_1 f_1(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{5}$$

Since not much research is done on Quoridor, there is not much known about evaluation features that perform well. Glendenning [2] proposed some evaluation features. Most of these features use the distance to the goal as an estimator.

3 Experimental setup

This section describes how the simulation environment, built in Java 2 Standard Edition 1.5.0, is set up. Also some evaluation features will be proposed. With these evaluation features, we will try to answer research question 3.

3.1 Simulation environment

One of the first decisions to be made was the representation of the Quoridor position. The most natural way to represent the board is to represent it as an undirected graph [1]. The squares on the board are vertices and the borders of two squares are edges (see figure 7). With these vertices and edges, it is easy to construct a graph of a Quoridor board. The graph datatype [3] makes it very easy to add and remove vertices/edges. This graph is the base of the QuoridorBoard object, which allows users to move their pawn and to place fences. The graph has to be built in the following way:

- Construct 81 vertices and add them to the graph.
- Add edges between each neighboring pair of vertices.
- Delete 2 edges, for each fence that is placed.
- Add temporary edges, and remove one edge temporarily, when the pawns are facing.

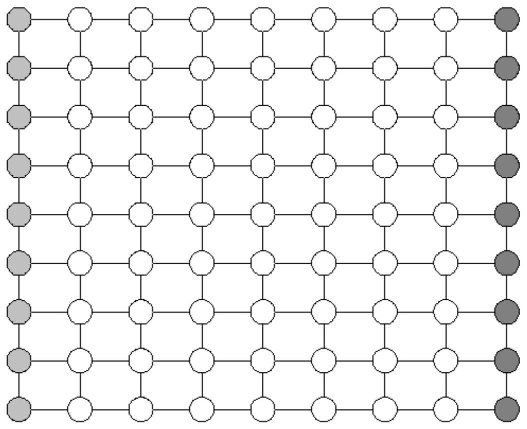

Figure 7: Quoridor graph for the start position.

When a move is to be made, the board checks whether this move is possible by searching the corresponding edge in the graph. If this edge is an element of the graph, the move is legal. The same holds when a fence is to be placed. The board will search for those two neighboring edges which are to be deleted. If these edges are an element of the graph and there is still a route left for the opponent to his goal, the fence move is legal. When the fence move is legal, the board will adjust the graph by deleting the two corresponding edges. After several steps, the board may look like figure 8. The corresponding graph of the board in figure 8 looks like figure 9.

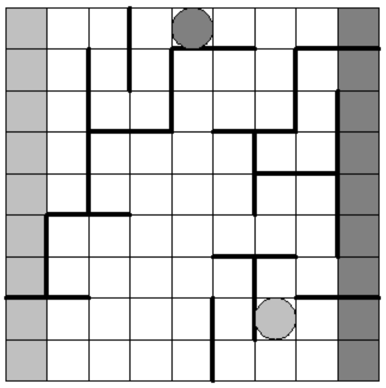

Figure 8: An example Quoridor board.

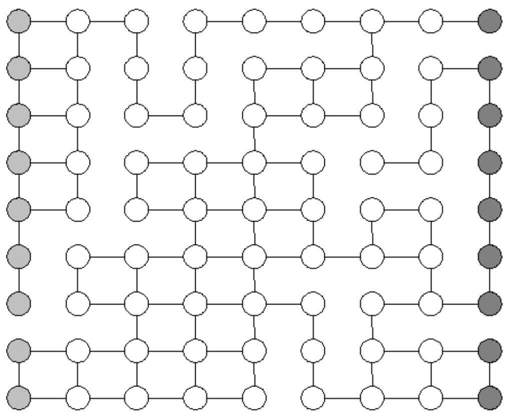

Figure 9: Quoridor graph corresponding to figure8.

One of the advantages of the graph datatype is that this datatype makes it easy to perform graph search. These search algorithms are used for two reasons. First, depth-first search [8], is used to check whether there is a route to the goal left when a fence is placed. Second, search algorithms are used to determine the shortest route to the goal from the pawn's position.

As mentioned earlier, MiniMax with Alpha-Beta pruning [7] is used to determine the move. Minimax uses a tree datatype. A tree consists of a list of nodes. These nodes keep track of their relationship (parent, child) to other nodes. Because MiniMax builds up a tree during search, the most important features of a tree are:

- Constructed nodes are added to the tree.

- Each node knows the position of both his parent and children in the tree.
- The tree takes care of the relationship between each node.

Finally there are the agents. Each type of agent is another object. An agent can only do one of two things, i.e., move his pawn or place a fence. The best move is obtained by applying MiniMax search on the game tree. Except when all fences are placed, the agent will automatically search for the shortest route to the goal. For this the agent uses a Breadth-First Search [8]. However, before the agent object can be used, the following things have to be done:

- activate one or more evaluation features.
- set weights for the activated evaluation features.

Next the evaluation features will be described.

3.2 Evaluation functions

One of the simplest evaluation features is the number of columns that the pawn is away from his base line column. We call this the position feature. So if the pawn is on his base line, the value is 0. If the pawn is on the goal line, the value is 8. This can easily be seen in figure 10.

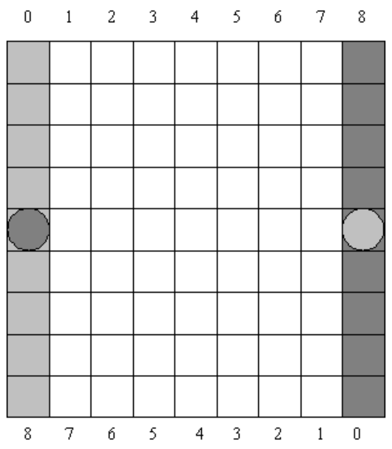

Figure 10: Position evaluation feature.

The next feature does not differ much from the previous one. This feature returns the difference between the position feature of the Max player and the position feature of the Min player. It actually indicates how good your progress is compared to the opponent's progress. This feature is called positionDifference.

When playing a game, each player will try to place fences in such a way that his opponent has to take as many steps as possible to get to his goal. To achieve this, the fences have to be placed so that the opponent has to move up and down the board. A feature derived from this fact is the movesToNextColumn feature. This feature calculates the minimum number of steps that have to be taken to reach the next column. For example, the pawn in figure 11a has to take at least 4 steps to reach the next column.

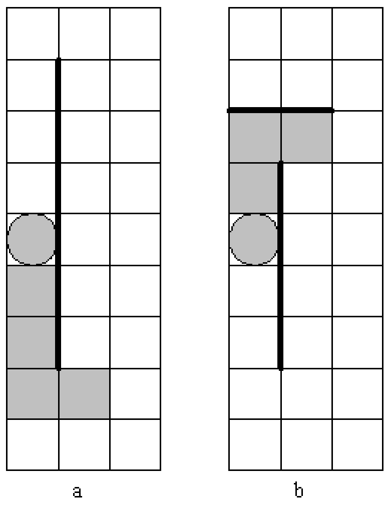

Figure 11: MovesToNextColumn evaluation feature.

This feature can also be used for the Max player. The Max player wants to minimize the maximum number of steps he has to take to the next column. Figure 11b shows that by placing the horizontal fence, the player is guaranteed a minimum of 3 steps to the next column. The Max player wants to minimize this distance. A small amount of steps has to give a higher evaluation. So the number of steps, of the Max player to the next column, is raised by the power of -1.

3.3 Test setup

To test which feature is better, the different features have to be tested against each other. However, because there is no variation in the MiniMax algorithm, the game will always run the same way. To bring some variation in the game, a random number [6], from the uniform distribution on the interval [0,1] (denoted U(0,1)), will be added to the evaluation value. The evaluation function is given in eq. (6)

$$Eval(s) = \sum_{i=1}^{n} w_i f_i(s) + U(0,1) \qquad (6)$$

There are 4 features to test, namely:

- position feature (f_1)
- position difference feature (f_2)
- Max-player's moves to next column (f_3)
- Min-player's moves to next column (f_4)

Because testing each combination of features against each other combination would take too much time, the features are tested as follows.

- $f_1 + f_2 + f_4$ vs $f_1 + f_2 + f_3$ (c_1 vs c_2)
- $f_2 + f_3 + f_4$ vs $f_1 + f_3 + f_4$ (c_3 vs c_4)
- The best of the second match will be tested against all features c_5.

This test setup is chosen to get a view on the difference between f_1 and f_2 on the one hand and the difference between f_3 and f_4 on the other hand. The last match is played to know the performance of the agent when all features are used. To get a good view on the performance of the features, each of the 3 test matches is done 100 times.

4 Results

In this section the results of the earlier described test matches will be given. Next, the results will be discussed.

4.1 Test results

In each test, 100 games were played. Normally a draw decides which player will start, however in these tests each player started 50 times. This way the importance of the starting position can be derived. Also, each game with more then 300 plies was not taken into account since it will probably never stop. Games not taken into account do not influence the average game length.

For all tests, the search depth of the MiniMax algorithm was set at 2. After some preliminary experiments the feature weights were set as in table 2.

Feature:	Weight:	
f_1	w_1	0.6
f_2	w_2	0.6001
f_3	w_3	14.45
f_4	w_4	6.52

Table 2: Feature weights.

The results of the 3 test matches, are given in table 3.

Match 1.1	Wins (on 50 played):	Av. plies
$f_1 + f_2 + f_4$ (c_1)	27	98.70
$f_1 + f_2 + f_3$ (c_2)	23	
Match 1.2		
$f_1 + f_2 + f_3$ (c_2)	49	76.60
$f_1 + f_2 + f_4$ (c_1)	1	
Match 2.1	Wins (on 50 played):	Av. plies
$f_2 + f_3 + f_4$ (c_3)	24	110.00
$f_1 + f_3 + f_4$ (c_4)	26	
Match 2.2		
$f_1 + f_3 + f_4$ (c_4)	2	89.64
$f_2 + f_3 + f_4$ (c_3)	48	
Match 3.1	Wins (on 50 played):	Av. plies
$f_2 + f_3 + f_4$ (c_3)	37	113.94
$f_1 + f_2 + f_3 + f_4$ (c_5)	13	
Match 3.2		
$f_1 + f_2 + f_3 + f_4$ (c_5)	9	92.82
$f_2 + f_3 + f_4$ (c_3)	41	

Table 3: Test Results.

4.2 Discussion of the results

Given the results of match 1.1 and 1.2, it is clear that the combination $f_1 + f_2 + f_3$ (c_2) is better than combination $f_1 + f_2 + f_4$ (c_1). The reason for this is that c_2 has the feature f_3. Feature f_3 is an attacking feature since it is used to minimize the number of steps to the next column. This is the opposite of feature f_4 which is a defending feature because it is used to maximize the minimum number of steps to the next column for the opponent. So we can say that f_4 keeps track of the opponent's progress rather than taking care of the player's progress. This also explains the small win of combination c_1 in match 1.1. Combination c_1 will probably always start defending because the opponent's number of steps to next column at the start is 1, which is obviously the smallest number of steps. Also combination c_2 will defend less, because of f_3, and use its fences to facilitate its own progress.

Combination $f_2 + f_3 + f_4$ (c_3) is the overall winner of matches 2.1 and 2.2. Combination $f_1 + f_3 + f_4$ (c_4) equals c_3 in match 2.1 but only wins 4% in match 2.2. Since c_3 is the starting player in match 2.1, c_3 will always have a small lead regarding to c_4. Because c_3 has got 2 defensive features, f_2 and f_4, c_3 is a more defending player compared to c_4. However the defensive nature of c_3 disappears when moving first. This is probably why c_4 has more wins in match 2.1. Now it is clear that the defensive nature of c_3 is of at most importance in match 2.2, when playing second. Combination c_3 defends better and therefore has more wins in match 2.2.

From matches 3.1 and 3.2 we can see that combination c_3 achieves more wins in both matches 3.1 and 3.2. In match 3.2, c_3 has even more wins than in match 3.1. The reason for this is the same as for which c_3 has more wins in match 2.2 than in match 2.1, which is mentioned above. Another fact that can be derived from match 3.1 and 3.2 is that combination $f_1 + f_2 + f_3 + f_4$ (c_5) wins more matches when not being the starting player. The reason for this fact is the same reason why c_4 has more wins in 2.1 than in 2.2. This is also mentioned above.

Another remarkable fact is the importance of being the starting player or not. A defensive combination like c_3 seems to have an advantage when being the second to move. On the other hand, a more attacking combination as c_2 has an advantage when being the first player.

Finally, we can confirm that the average number of plies is around 91 (96.85 from our results) as stated by Glendenning [2].

5 Discussion

In this section, two issues will be treated. First we will go into detail on the use of features and second the optimization of the weights will be discussed.

5.1 Features

The use of features in the evaluation function is quite simple. However, finding a good feature is not that simple. A good feature is a feature that tells the player how well he is performing and how big his chances are to win from the current position. Also, the calculation of the feature values must not take too much time since otherwise it is better to raise the search depth. So the features have to catch the lack of search depth of the MiniMax algorithm. Therefore features should tell more about how well the player will perform further in the game rather than telling how well the current performance is.

5.2 Optimizing weights

One of the major problems is the setting of the weights. The difficulty lies in the fact that it is hard to estimate the importance of each feature. A rough estimation can be made but an exact setting is very hard. Therefore a function to optimize the weights should be used.

6 Conclusions

In this section conclusions based on the results will be drawn. Also the future research will be discussed.

6.1 General conclusions

The extensions of the MiniMax algorithm, namely the proposed features, do not lead to a strong Quoridor-playing agent. The level that is reached by these heuristics is an amateur level. One of the reasons for this is the low depth used in the MiniMax search. Deeper search would have led to better moves and thus better results.

Another reason for the fact that no higher level is reached is the adjustment of the weights. However, with the weights that were used, it is clear that feature f_1 is a weak feature since combinations with f_1 only won 28% of the games played against combinations without f_1. The problem with feature f_1 is that it tries to force a step forward, even if this step is not on the shortest route to the goal or worse if this step leads to a dead end. Feature f_2 also has this characteristic but less than f_1. This is why combination c_3 is better than combination c_4.

Features f_3 and f_4 performed well but could perform better if they represented the shortest distance to the goal and not the shortest distance to the next column because the shortest route to the next column is not always on the shortest route to the goal. However this would cause a time penalty because generally finding the shortest route to the goal takes more time than finding the shortest route to the next column. We can say that features f_3 and f_4 are efficient in time usage, but they have a lack in their effectiveness.

The simplicity of the proposed features can be ascribed to the fact that research in Quoridor is still in its infancy. And hopefully the research that is done imparts to a better insight in the game.

6.2 Future research

One of the major problems is the lack of good features. So in future research, better features must be found. One possible method for new features is pattern recognition. A pattern can be derived from the position of the fences on the board and the positions of the pawns. Hebbian learning [4] can then be used to train the linear associator perceptron. The output of the perceptron will then indicate how good the pattern is and so give an estimation of how well the player is performing. The use of a hard limit perceptron can be considered. The output would then indicate if the pattern is a winning or losing configuration. This would be very useful since, as mentioned in the discussion, this is one of the important tasks of a feature. Another advantage of the perceptron is the few computational time it needs to evaluate the pattern since the perceptron is based on simple matrix calculations.

Another possibility for future research is the use of evolutionary algorithms for weight optimization. Glendenning [2] also used evolutionary algorithms to set the feature weights.

Of course the use of a fast computer is strongly advised. Game computers like Deep Blue [5] are able to explore 200.000.000 positions per second, which means they are able to search very deep in the game tree.

References

[1] Buckley, F. and Lewinter, M. (2003). *A friendly introduction to Graph Theory*. Prentice Hall, New Jersey.

[2] Glendenning, L. (2002). Mastering quoridor. B. Sc. thesis, University of New Mexico.

[3] Goodrich, M.T. and Tamassia, R. (2002). *Algorithm design*, pp. 288–308. Wiley, New Jersey.

[4] Hagan, M.T., Demuth, H.B., and Beale, M. (1996). *Neural Network Design*, pp. 7.1–7.31. PWS Publishing, Boston.

[5] IBM (2006). Deep blue. `http://www.research.ibm.com/deepblue/meet/html/d.3.shtml`.

[6] Law, A.M. and Kelton, W.D. (2000). *Simulation modeling and analysis*, pp. 402–403. Mc Graw-Hill, Singapore.

[7] Russel, S. and Norvig, P. (1995a). *Artificial Intelligence, a modern approach*, pp. 161–167. Prentice Hall, New Jersey.

[8] Russel, S. and Norvig, P. (1995b). *Artificial Intelligence, a modern approach*, p. 75. Prentice Hall, New Jersey.

[9] Herik, H.J. van den, Uiterwijk, J.W.H.M., and Rijswijck, J. van (2002). Games solved: Now and in the future. *Artificial Intelligence*, Vol. 134, pp. 277–311.

[10] Wikipedia (2006). Game-tree complexity. `http://en.wikipedia.org/Game-tree_complexity`.