

Applying Genetic Algorithms to Quoridor Game Search Trees for Next-Move Selection

CS486 Final Group Project Report

Quinn McDermid
(99038083; kqmcderm@student.math)

Anand Patil
(00455661; apatil@student.math)

Touran Raguimov
(00120204; traguimo@student.math)

August 14, 2003

**School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada**

Table of Contents

1. Introduction	3
1.1 The Game: Quoridor	3
1.2 Genetic Algorithms	5
1.3 Simulated Annealing	5
2. Background	6
2.1 Problem	6
2.2 Literature Survey	6
3. Methods	9
3.1 Approach	9
3.2 Rationale	11
4. Plan	13
4.1 Hypotheses	13
4.2 Experimental Design	13
5. Experiments	15
5.1 Results	15
5.2 Critical Evaluation	16
6. Conclusion	18
7. References	19
8. Appendix A: Java Source Code for Genetic Algorithm Based AI Players	20
8.1 Total Fitness Selection Player	20
8.2 Tournament Selection Player	22
8.3 Rank-Based Selection Player	24
9. Appendix B: Java Source Code for Simulated Annealing AI Player	26

1. Introduction

1.1 The Game: Quoridor

Quoridor is a conceptually simple two- or four-player board game invented in 1997 by Mirko Marchesi, an Italian game designer working for Gigamic¹. Since Quoridor is a relatively unknown game in the literature, a brief description of the game setup and rules is given below:

The Board: There is a single game board with 81 squares (9x9). There are two storage slots for the 20 fences. In two-player mode, there are 2 pawns (see Figure 1.1), and in four-player mode, there are 4 pawns.

The Purpose: The purpose of the game is to be the first to reach the line opposite to one's base line (see Figure 1.7).

*The Rules*²: Since this report only considers the two-player form of this game, only the rules for the two-player configuration are given.

Starting: The fences are placed in their storage area, 10 for each player. Each player's pawn is placed in the centre of his base line (see Figure 1.1). A draw will determine who is allowed the first move.

Gameplay: Each player in turn, chooses to move his pawn or to put up one of his fences. When he has run out of fences, the player must move his pawn.

Pawn Moves: The pawns are moved one square at a time, horizontally or vertically, forwards or backwards (see Figure 1.2). The pawns must get around the fences (see Figure 1.3).

Positioning of the Fences: The fences must be placed between 2 sets of 2 squares (see Figure 1.4). The fences can be used to facilitate the player's progress or to impede that of the opponent, however, an access to the goal line must *always* be left open (see Figure 1.5).

Face to Face: When two pawns face each other on neighbouring 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 1.6). If there is a fence behind the said pawn, the player can place his pawn to the left or the right of the other pawn (see Figures 1.8 and 1.9).

¹See <http://www.gigamic.com/jeuxanglais/quoridore.htm> for more details.

²As described at <http://www.gigamic.com/regles/anglais/rquoridore.htm>. (c) Copyright 1997 GigamicS.A. from a concept of Mirko Marchesi, developed by EPTA srl and QUALITY GAME srl.

End of Game: The first player who reaches one of the 9 squares opposite his base line is the winner (see Figure 1.7).

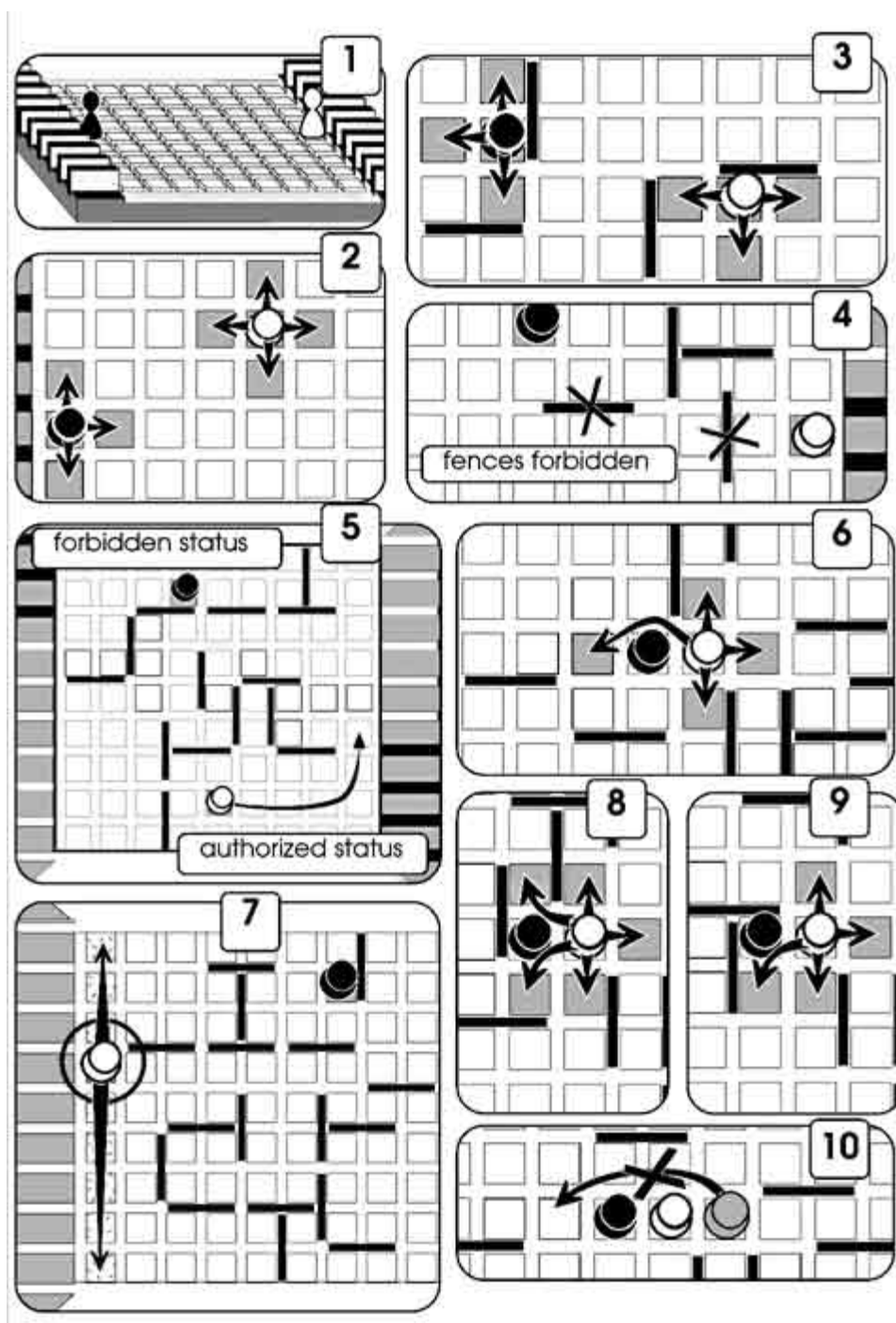


Figure 1³: (1) The initial configuration of the board prior to game play, with each player's pawn placed at the centre of his or her baseline. (2) Legal pawn moves without fences. (3) Legal pawn moves around fences. (4) Legal placements of fences. (5) Legal game states; must allow access to goal line at all times. (6) Jumping an opponent's pawn. (7) The goal

³ Image taken from <http://www.gigamic.com/regles/anglais/rquoridore.htm>. No explicit copyright was declared for this image at its source, but its use is nonetheless covered under the fair-use (nonprofit educational purposes) statute present in the governing copyright legislation.

state; reaching any of the 9 squares on the opponent's goal line. (8) and (9) Legal moves in the event there is a fence behind the opponent. (10) In a four-player game, it is forbidden to jump more than one pawn.

1.2 Genetic Algorithms

Introduced by John Holland in 1975, a genetic algorithm is an approach that mimics biological processes for evolving optimal or near optimal solutions to problems. It performs a multi-directional search, maintaining a population of potential solutions. Beginning with a random population (group of chromosomes), it chooses parents and generates offspring using operations analogous to biological processes, usually crossover and mutation. Adopting the *survival of the fittest principle*, all chromosomes are evaluated using a fitness function to determine their fitness values, which are then used to decide whether the chromosomes are eliminated or retained to propagate. The more adaptive chromosomes, i.e. the "good" solutions, are kept to reproduce again and the less adaptive ones, i.e. the "bad" solutions, are discarded in generating a new population. Traditionally, this new population replaces the old one and the whole process is repeated until specific termination criteria are satisfied. During each iteration step, called a generation, the structures in the current population are evaluated, and on the basis of those evaluations, a new population of candidate solutions are formed (Özyildirim, 1997). The chromosome with the highest fitness value in the last population gives the solution (Forrest, 1996; Hong, *et al.*, 2001).

Genetic algorithms have been successfully applied to the fields of optimization, machine learning, neural networks, fuzzy logic controllers (Hong, *et al.*, 2002), and to mainstream economics problems (Özyildirim, 1997; Riechmann, 2001). More interestingly to this report, genetic algorithms have been widely used in a wide range of game playing issues (Buro, 1995; Davis and Kendall, 2002; Forrest, 1996; Hong, *et al.*, 2001; Hong, *et al.*, 2002; Kendall and Whitwell, 2001; Rosin and Belew, 1996).

1.3 Simulated Annealing

First proposed by Kirkpatrick in 1983, simulated annealing was initially designed for solving combinatorial optimization problems. As its name implies, simulated annealing exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system. It is a global optimization method that distinguishes between different local optima, that is, it is an optimization of greedy local search. Starting from an initial point, the algorithm takes a step and a heuristic function is evaluated. The algorithm employs a random search which not only accepts changes that decrease an objective heuristic function, but also some changes that increase it (Fleischer, 1995). Essentially, bad moves are randomly taken. If the neighbour of a current assignment is better than the current assignment, a move to the neighbour is taken. Otherwise, if the neighbour is worse, a move to the neighbour is still considered with a probability defined by the objective function. The major

advantage with simulated annealing over other methods is an ability to avoid becoming trapped at local minima and, hence, has a greater chance of finding the global optima (Li and Lim, 2002).

2. Background

2.1 Problem

The purpose of this report is to explore the use of genetic algorithms in creating an artificial player for two-player Quoridor. Specifically, genetic algorithms will be developed to traverse the game search tree in order to select the "best" next-move. In selecting the next generation in the algorithm, various fitness and selection criteria will be explored and compared empirically. Also, for the purposes of empirical comparison, a player will be developed using simulated annealing techniques.

2.2 Literature Survey

As mentioned before, Quoridor is a relatively unknown game in the realm of academic research. As such, literature on similar two-player, adversarial games such as awari (Davis and Kendall, 2002), checkers (Rosin and Belew, 1996), chess (Kendall and Whitwell, 2001), draughts, and Othello (Buro, 1995) were used to provide guidance in designing the game search algorithms, genetic and otherwise, for this report.

The general representation of a two player game is a rooted tree (called the *game search tree*) as follows: Every possible position in the game is denoted by a node. The root represents the starting position, while every leaf denotes a possible end. An edge from a node x to another node y (here, y is the child of x) represents a possible move from position x to position y . The game starts at the root, and each player, in turn, moves from the node on which it is located to any child of this node. The game is finished when a player ends up on any of the leaves (Tarsi, 1983).

The key mechanism for effective computer game playing is the game search algorithm. This algorithm is responsible for exploring possible moves in searching for the best next moves. Usually, the deeper the game search tree, the more accurate the prediction (Hong, *et al.*, 2002). Traditional two-player game search algorithms, such as *minimax* and α - β *pruning*, suffer great time and space inefficiencies when exploring deeply into search trees to find better playing strategies (Hong, *et al.*, 2001). The main focus of this report is to apply genetic algorithms to game-tree searches to improve this performance.

The evolving ability of genetic algorithms to find global, or nearly global, optima in limited time seems promising. However, they have been rarely used for game-tree search due, primarily, to their fitness functions. While traditional fitness functions find

only maximum or only minimum values, game-tree search for two-player adversarial games requires fitness functions to compute iterative minimax values. Hong, *et al.* (2001) proposed a genetic minimax search strategy for determining good next moves employing conventional genetic algorithms. It uses evaluation values calculated from a partial game-tree search to generate offspring. In their model, after generation, the best solution is output as the next game move. Their experiments showed that the proposed method finds more accurate solutions more quickly than the original minimax search strategy. They claim that their algorithm is thus practical and can be applied to real computer games.

Essential to the selection of the next move is the evaluation of the current game position. These evaluation functions estimate the players' winning chances in the positions at the leaves of game trees, and these values are propagated upwards in order to choose a move which leads to the highest score. According to Buro (1995), these evaluation functions normally combine features that measure properties of the position correlated with the winning chance, such as material in chess or mobility in Othello. However, the techniques employed to determine the feature weights were laborious. Buro offers the following strategies to enhance this calculation. First, the strategy of move adaptation. In this strategy, evaluation function parameters are tuned to maximize the frequency with which searches yield moves that occur in the lists of moves belonging to training positions. The idea is to get the program to mimic experts' moves. The other main strategy Buro offers is that of value adaptation. Given a set of labeled example positions, parameters are determined such that the evaluation function fits a specific model. For instance, evaluation functions can be constructed in this way to predict the final game result. Buro suggests that in games such as Othello and chess, value adaptation is more promising as move adaptation does not offer any global interpretation for optimized evaluation functions. With value adaptation, evaluations from different phases are comparable if the example position labels have a phase-independent meaning.

Also essential to the search process in any genetic algorithm are the stages of crossover and mutation. Kendall and Whitwell (2001) proposed an interesting approach for crossover. In their model, there is no crossover. That is, reproduction is completely asexual. After the selection and competition of two population members, there will be at most one breeding parent - the fittest one of the pair. The population member that wins the competition becomes the breeding parent and creates a clone to replace its rival. In the case of a drawn competition, Kendall and Whitwell denied both candidate parents from breeding. They also suggest an interesting approach to mutation. Their proposed mutation system is dependent on the distribution of the population to create a more explorative search. After each competition between pairs of candidate evaluation functions, they mutate each of their parameters by a proportion of the standard deviations for those parameters for those parameters depending on the outcome of competition. They claim that when the population is diverse at the start of the learning process, they mutate by a larger amount than when the population begins to converge towards the end of learning. Thus, all mutation is controlled by the population's diversity, and the user does not need to specify a mutation scheme. Kendall and Whitwell developed a chess player that, according to their results, had exhibited better learning under their design than the "before" design.

Designing the fitness function is important to create an effective genetic algorithm. It is the fitness function that evaluates each chromosome and generates values representing their degree of fitness. Chromosomes with smaller fitness values are usually discarded to increase the probability of retaining population superiority. Hong, *et al.* (2001) propose the use of a *reservation tree* to aid in fitness evaluation. They claim that since the minimax principle must be used to justify the fitness of any new individual in a two-player game-search tree, chromosomes must be therefore compared with other chromosomes in the population to get accurate fitness values. As such, this reservation tree is used to retain the fitness of all chromosomes ever seen to aid in fitness evaluation. Their design connects all inspected individuals to a tree structure, evaluates each individual's leaf node, and then propagates the board evaluation values upward along the current partial game tree. During propagation, the value of each node is evaluated using the minimax principle. In a two-player game-search tree, they claim that the best move sequence (path) is the one whose leaf value is propagated along the path to the root node in the reservation tree. That is, each node on the path has the same value as the leaf node. A path whose leaf value can be propagated to a higher level can thus be thought of as more important. The fitness function in their proposed genetic minimax algorithm is defined as the height from the bottom that the leaf value of a chromosome (path) can attain. Hong, *et al.* concluded that their genetic algorithm-based game-search method was remarkable for the introduction of this reservation tree notion. Their experimental results showed that their method not only accelerated the search speed but also found a more accurate solution in a limited amount of time.

Baum, *et al.* (1995), among other things, explored the interesting question of whether or not breeding proportional to fitness is optimal. They examined standard genetic algorithm breeding methods and compared them with an alternate sampling method they termed *culling*. This breeding method chooses parents randomly and uniformly, and breeds them. If the resulting child has fitness statistically greater than the expected fitness, μ , (plus one standard deviation, σ) of a member of the current generation, the child is accepted. Otherwise, the child is rejected. This means that they only reject a constant fraction of children, but in the process, achieve a gain per generation of at least one standard deviation, σ . More generally, they propose, they could reject children of fitness less than $\mu + \lambda \sigma$ and optimize the parameter σ . Their experimental results suggested that culling was near optimal on the problem they were interested in (additive search problem), far better than the standard genetic algorithms. Incidentally, they found that their culling method was also highly noise tolerant.

3. Methods

3.1 Approach

While researching the game, we had come across an open-source Java implementation of the game, called HardQuor⁴ that was written specifically to interface easily with new and different AI players. As packaged, the game allowed for a single human player to play against either another human player through a network, or a provided AI player. We modified this game implementation to allow for two AI players to play against each other.

In developing the genetic algorithm-based AI player, the following components were given special attention:

Evaluation Function:

Our AI players, both genetic and annealing, use the same heuristic function to evaluate their current position in the game. This heuristic considers both the number of walls each player has left to place and the difference in the distances of each player to reaching the goal side. It is a linear function in both of these characteristics.

Fitness Function:

The fitness of an individual is calculated to be the value of the evaluation (heuristic) function in their current state after they have completed all their moves.

Generating Populations:

We considered all of the search tree moves possible, selected a random move from these. Then, we assume the opponent myopically selects the *best* move from its point of view. Then we select a random move from the available moves at that point. This continues until our depth bound within the tree is reached. We have designed the genetic algorithm player in such a way that it is trivial to alter various parameters associated with a population, including number of individuals and generations.

Selection Methods:

The selection method is key in determining how quickly a game search converges versus the breadth of the search. For example, a selection method that always selected the best individual would simply converge to that individual, and there would be no breadth to the search. A selection method that would select individuals from the population with equal probability would have a large breadth, but would not likely converge to the best solution. We implemented and compared the following selection methods:

- (i) *Total Fitness:* The fitness of all individuals in the population is summed.

⁴See <http://www.swank.ca/slacker/hardquor/index.html> for more information. Please note that at the time of submission, the given webpage was undergoing reconstruction and the HardQuor page content had been moved offline temporarily. They claim the content will be available again August 15, 2003.

Then, the probability for one individual being selected is their fitness divided by the total population's fitness.

(ii) *Tournament Selection*: Two individuals are selected randomly and uniformly from the population so that each individual has equal probability of being selected. Then, the individual with the higher fitness is selected from these two.

(iii) *Rank-Based*: Individuals are sorted and ranked based on their fitness. The ranks are then summed (mathematically, $\frac{\text{size}(\text{population}) * [\text{size}(\text{population}) + 1]}{2}$). The individual's probability of being selected is then their rank over the total sum of ranks.

Crossover:

As suggested by Kendall and Whitwell (2001), we decided not to implement crossover. This seemed appropriate as crossovers in the Quoridor game search tree would, with high probability, result in illegal moves. As described in Hong, *et al.* (2001), when there are a variable number of possible moves at each node on the same level of a game tree, implementing crossover so that the maximum number of possible moves at each level can be used, but many illegal offspring states may be derived. Although appropriate repair mechanisms can still be designed to resolve the conflicts, much additional and unnecessary computational time may be spent if many or most offspring chromosomes are illegal. As such, we felt that adding crossover would have no beneficial effects on our implementation, only demanding more overhead work to recover from the illegal moves.

Mutation:

This was a two-step process. We first select a point in the path for the selected individual. Once the point was selected, we regenerate a path as if we were generating a new individual starting at that point. The probability of any one move in the path being selected is that move's index in the path divided by the total sum of the indices. This implies that an earlier move in the path has less probability of being selected for mutation in the path than a later move. This was done because an earlier move would suggest that the individual associated with the move would be relatively more fit than later individuals. Mutating these earlier moves would have a more drastic effect than mutating later moves. For example, if the first move in a path is selected for mutation, then it results in a completely new individual that has no relation to its parent.

In developing the simulated annealing algorithm-based AI player, the following components were given special attention:

Evaluation:

Since simulated annealing is an optimization of greedy local search, the evaluation function for nodes would assume nodes to be myopic. Specifically, it would not incorporate any adversarial strategy of an opponent player into evaluating a node, such as assuming that the opponent will select a move that is best for itself. As such, the

application of simulated annealing to two-player game search trees requires some expansion in the design of the evaluation function. We have decided to expand the evaluation function of nodes for a given player by having it traverse down the tree from the given move, taking into consideration an opponent's myopic move at that level, and then selecting the move under traditional simulated annealing. This would conceptually translate into the simulated annealing player looked downward three levels from a given position in the game tree to evaluate this position. The same linear evaluation heuristic function is used for the evaluating the game position itself, taking into consideration both the number of walls left for each player and the distance of each player to the goal side.

Objective Function:

The objective function is responsible for determining the probability with which worse moves are taken from a given position. Our objective function is of the form:

$$e^{[h(\text{current})-h(\text{neighbour})]/\text{time}}$$

where h is the same heuristic evaluation function used in the genetic algorithm. Here, our parameter controlling randomness versus greediness is time, and not a constant. This was inspired by Fleischer (2001), which described the use of temperature for this parameter under traditional annealing with metals. This is the aspect of our simulated annealing which avoids becoming trapped in local optima. When we reach later stages in the game search tree where the player is close to its goal state, the difference between $h(\text{current})$ and $h(\text{neighbour})$ will be relatively large (conversely, it would be relatively small at the beginning of the game). This would suggest that using a constant parameter in place of time would result in the player taking bad moves more often near the end. Since time would also be a relatively large value near the end of the game, it would offset this large difference and get us to a goal state quicker.

3.2 Rationale

First, we offer our rationale for selecting a genetic algorithm-based player rather than implementing other search tree algorithms. It is characteristic of most two-player games that, if one could search the entire depth of the game search tree down to the leaf nodes (i.e. goal nodes) with some given algorithm to determine the next move, then this algorithm would constitute a perfect player that would make perfect moves. However with Quoridor, as with many other two-player games, this is infeasible given the magnitude of the search trees. As mentioned in Hong, *et al.* (2001), genetic algorithms offer the ability to go deeper into the search tree, with reasonable space and time requirements, to find optimal solutions.

As far as our rationale for the specifics of the genetic and simulated annealing algorithms implemented, we refer the reader to Section 3.1 above which offers the reasoning for our design decisions among the descriptions.

Lastly, we selected simulated annealing as a final comparison for our genetic algorithm partially based on the advice of Dale Schuurmans, and partially based on our realization (during our literature search and survey) that very few efforts have been made at applying simulated annealing to two-player game search trees. As mentioned in the introduction, simulated annealing was initially designed for solving combinatorial optimization problems. We noticed that simulated annealing is quite similar to the motivation behind genetic algorithms, but much simpler. As such, we felt it would serve as a good comparison tool. Its performance as an optimization to greedy local search is quite celebrated in the literature (Li and Lim, 2002). We believe that our modification of standard simulated annealing may provide some advances in how simulated annealing may be applied to game search problems.

4. Plan

4.1 Hypotheses

H1(a): Tournament selection *will outperform* selection using total fitness.

H1(b): Rank-based selection *will outperform* selection using total fitness.

H1(c): Rank-based selection *will outperform* tournament selection.

H2: Performance will worsen as the number of individuals per generation is increased and the number of generations is decreased.

H3: The simulated annealing player will outperform the genetic algorithm player.

4.2 Experimental Design

As mentioned in Section 3.1 above, we selected an open-source Java implementation of Quoridor we came across on the Internet that was built in such a way to allow for easy interface with automated AI players. This package was setup so that a human player could face off against a computer AI player. We modified the functionality so that it would allow two computer AI players to face off against each other.

For all of the hypotheses presented, we define the notion of *performance* to be strictly based on the win-loss ratio when two players face off over multiple games. We assume that a player has outperformed the other player when it wins more than half of the trial games. We require multiple trials because of the notion of randomness inherent to both the genetic and simulated annealing algorithms.

To address the first set of hypotheses (H1) exploring the different selection methods used, three separate Java classes were created to represent players implementing each of the selection methods. Please see Appendix A for the Java source code. To gather evidence for evaluating each hypothesis, we simply ran the modified game with the appropriate AI players. For each hypothesis, we ran six such games. To avoid any first-move bias, we allowed each AI player to have the first move an equal number of times. Since we are using the same standard random number generator (`java.util.Random`) for all AI players, we assume that no bias is introduced when selecting random individuals. Only the selection methods are different between the players, all other common components (e.g. game state representation, board representation, etc.) were kept the same.

For the second hypothesis (H2) exploring the performance of the genetic AI player with respect to changing population parameters, we used the genetic AI player

from our H1 tests that was best. That is, we used the genetic AI player whose selection method outperformed the other two selection methods implemented. The default implemented values for the (individuals, generation) pair are (25, 3). We used this default-settings player as the common opponent to all of the various degrees of increased individuals and decreased generation players we tried. Specifically, we tried the following values for (individuals, generations): (50, 2), and (10, 5). For each value pair (and its associated modified-settings player), we have the default-settings player face off against the modified-settings player 6 times. As with the H1 testing, we have each player make the first move an equal number of times to avoid any first-move bias.

To explore the third hypothesis (H3), we implemented a Java class to represent a player using the simulated annealing strategy described above. Refer to Appendix B for the Java source code. We selected the genetic AI player from our H2 tests that yielded the best performance to face off against the simulated annealing player. As with the testing of the previous hypotheses, we let these two players face off against each other 6 times, each player getting the first move an equal number of times.

5. Experiments

5.1 Results

H1 Testing Results:

Refer to Table 1 for the results of the H1 tests.

<i>Selection Method</i>	<i>Opponent Selection Method</i>		
	Total Fitness	Tournament	Rank-Based
Total Fitness	--	2	2
Tournament	4	--	3
Rank-Based	4	3	--

Table 1: The results of H1 testing, comparing the performance of different selection methods for the genetic algorithm-based AI players. The values shown represent the number of games won by the player using selection method in the left-column against the specified opponent.

H2 Testing Results:

Refer to Table 2 for the results of the H2 tests. The tournament-based selection player was used.

<i>Parameter Values</i> <i>(individuals, generations)</i>	<i>Games Won</i>
(50, 2)	4
(10, 5)	5
(10, 5) [vs (50,2)]	6

Table 2: The results of H2 testing, comparing the performance of different population parameters for the genetic algorithm-based AI players against the default-valued genetic AI player. The values shown represent the number of games won by the modified-parameter player in the left-column against the default-valued opponent. The default values for the (individuals, generations) parameter are (25, 3). The third test was run for curiosity, and was not part of the original experimental design.

H3 Testing Results:

Refer to Table 3 for the results of the H3 tests.

<i>AI Player</i>	<i>Games Won</i>
Genetic	3
Simulated Annealing	3

Table 3: The results of H3 testing, comparing the performance of the *best* genetic AI player against the simulated annealing AI player. The values shown represent the number of games won by each of the players in the 6-game playoff. The genetic AI player selected was tournament-based selection with (individuals, generations) values (10,5).

5.2 Critical Evaluation

H1 Testing Evaluation:

The results suggest that selection by total fitness was the worst of the three implemented methods, as hypothesized. This is expected because this selection method implements the narrowest search in the tree. Thus, it is more likely to be dominated by one individual with high fitness individually, converging to it too quickly. This quick convergence yields poor results because a good first move can be quickly overshadowed by poor moves selected later due to random selection. A broader search would offer a slower convergence to a solution, as it would not focus on a single individual with high probability. This can be shown mathematically:

For total fitness selection:

- assume the distribution of fitness among individuals can be modeled by the function $f(x) = 1/x$ between $x = 1$ and $x = n$
 - this is a reasonable assumption by observation
- note that the integral of $f(x) = 1/x$ is given by $F(x) = \ln x$
- P(selecting an individual from the weaker half)
 - $= 1 - P(\text{selecting an individual from the stronger half})$
 - $= 1 - [(\text{area under } f(x) \text{ between } 1 \text{ and } n) - (\text{area under } f(x) \text{ between } n/2 \text{ and } n)] / [\text{area under } f(x) \text{ between } 1 \text{ and } n]$
 - $= 1 - [(F(n) - F(1)) - (F(n) - F(n/2))] / [F(n) - F(1)]$
 - $= 1 - [F(n/2) - F(1)] / [F(n) - F(1)]$
 - $= 1 - [\ln(n/2) - \ln(1)] / [\ln(n) - \ln(1)]$
 - $= 1 - [\ln(n/2)] / [\ln(n)]$
 - $= 1 - [\ln(n) - \ln(2)] / [\ln(n)]$
- as n gets infinitely large, we can apply l'Hopital's Rule so that this probability is:
 - $= 1 - \lim_{n \rightarrow \infty} \{ [1/n] / [1/n] \}$
 - $= 1 - 1$
 - $= 0$
- thus, the probability of an individual being selected from the weaker half of a population as the size of the population grows is 0

For tournament selection:

- P(selecting an individual from the weaker half) $= 1/2$
- an individual from the weaker half will only be able to win if it is competing with another individual from the weaker half.
- P(selecting two individuals from the weaker half to compete) $= 1/2 * 1/2 = 1/4$
- thus, the probability of an individual being selected from the weaker half of a population is $1/4$

For rank-based selection:

- P(selecting an individual from the weaker half)
 - $= (\text{sum of indices over first } n/2 \text{ ranks}) / (\text{sum of indices over all } n \text{ ranks})$

$$= [(n/2)(n/2+1)/2] / [n(n+1)/2]$$

$$= (n^2 + 2n) / [4(n^2 + n)]$$

-as n gets infinitely large, we can apply l'Hopital's Rule so that this probability is:

$$= \lim_{n \rightarrow \infty} \{(2n + 2) / (8n + 4)\}$$

-again, applying l'Hopital's rule

$$= 2/8$$

$$= 1/4$$

-thus, the probability of an individual being selected from the weaker half of a population is 1/4

The results of the experiment also suggest that tournament selection and rank-based selection offered similar performance. Both outperformed selection by total fitness because they involve much broader searches. Although we expected rank-based to outperform tournament selection, the calculations above justify their like performance as both methods result in the same probability of selecting an individual from the weaker half.

H2 Testing Evaluation:

The results suggest that the (10,5) player was by far the best, beating both the default (25,3) player and the (50,2) player. This is consistent with our hypothesis; performance improves as the number of individuals is decreased and the number of generations is increased. This hypothesis does not appear to have held in the (25,3) vs. (50,2) test, where the (50,2) player won more games. However, we believe that of the two parameters we altered, the number of generations has a greater effect on performance. The (10,5) player combines the locality of more generations with the larger breadth of search that tournament selection offers (as described above) to yield such good performance.

H3 Testing Evaluation:

The results suggest that both the tournament-selection based genetic AI player and the simulated annealing AI player performed at an equal level. This is encouraging, as it suggests that if we had designed our evaluation function for the simulated annealing player so that it explored the tree to a greater depth (currently, 3 levels down), it may offer greater performance.

6. Conclusion

This project involved exploring genetic algorithms to great depths. We designed and implemented a variety of selection techniques, and relied on current research in the field to design other aspects such as crossover and mutation. We also explored simulated annealing to some depth, and through our modifications, we may have introduced it to the academic world as a viable technique to apply to adversarial, two-player zero-sum games.

The game we chose to use, Quoridor, is also quite novel to the academic world. We believe Quoridor is a good model to explore game search techniques, as it has certain characteristics that distinguish it from similar two-player games such as chess. Since this game does not appear to have been used in the literature, we were forced to examine it closely to learn of and exploit its characteristics to make our AI players “smarter.” For example, we realized that we had to assign differing weights to wall moves and pawn moves. This newly acquired domain knowledge proved to be quite effective in fine-tuning our algorithms.

Further testing would serve to confirm our results, for example running more games for each test. In our exploration of the effects of altering the population parameters (individuals and generations), future work could be done to confirm that it is indeed the number of generations that has a greater affect than the number of individuals. This could be achieved by isolating and altering one of the parameters, while examining the effect on the player’s performance. Also, very little testing was done with respect to the effectiveness of our heuristic function. More testing of the weights given to the number of walls left for each player and difference of distances to a goal state may serve to fine tune the heuristic. Also, it is conceivable that other factors could improve our heuristic. For example, the number of possible paths or considering the next shortest path from a given position may make our heuristic more representative of the optimal heuristic function.

7. References

- Baum, E. B., Boneh, D., Garrett, C. (1995).** On Genetic Algorithms. In proceedings of the Eighth Annual Conference on Computational Learning Theory, Santa Cruz, California, USA, pp 230-239.
- Buro, M. (1995).** Statistical Feature Combination for the Evaluation of Game Positions. *Journal of Artificial Intelligence Research*, vol 3, pp 373-382.
- Davis J. E. and Kendall G. (2002, May 12-17).** An Investigation, using Co-Evolution, to Evolve an Awari Player. In proceedings of Congress on Evolutionary Computation (CEC2002), Honolulu, Hawaii, USA, pp 1408-1413.
- Fleischer, M. (1995).** Simulated Annealing: Past, Present, and Future. In proceedings of the 1995 Winter Simulation Conference, Arlington, Virginia, USA, pp 155-161.
- Forrest, S. (1996, March).** Genetic Algorithms. *ACM Computing Surveys*, vol 28(1), pp 77-80.
- Hong, T. P., Huang, K. Y., Lin, W. Y. (2001).** Adversarial Search by Evolutionary Computation. *Evolutionary Computation*, vol 9(3), pp 371-385.
- Hong, T. P., Huang, K. Y., Lin, W. Y. (2002).** Applying genetic algorithms to game search trees. *Soft Computing*, vol 6, pp 277-283.
- Kendall, G. and Whitwell, G. (2001, May 27-30).** An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics. In proceedings of the 2001 IEEE Congress on Evolutionary Computation, Seoul, Korea, pp 995-1002.
- Li, H. and Lim, A. (2002, March).** Local search with annealing-like restarts to solve the vehicle routing problem with time windows. In proceedings of the 2002 ACM Symposium on Applied Computing, Session: Evolutionary Computing and Optimization, Madrid, Spain, pp 560-565.
- Özyildirim, S. (1997).** Computing open-loop noncooperative solution in discrete dynamic games. *Journal of Evolutionary Economics*, vol 7, pp 23-40.
- Riechmann, T. (2001).** Genetic algorithm learning and evolutionary games. *Journal of Economic Dynamics & Control*, vol 25, pp 1019-1037.
- Rosin, C. D. and Belew, R. K. (1996).** A Competitive Approach to Game Learning. In proceedings of the Ninth Annual Conference on Computational Learning Theory, Desenzano del Garda, Italy, pp 292-302.
- Tarsi, M. (1983, July).** Optimal Search on Some Game Trees. *Journal of the Association for Computing Machinery*, vol 30(3), pp 389-396.

8. Appendix A: Java Source Code for Genetic Algorithm Based AI Players

8.1 Total Fitness Selection Player

```
package ca.swank.hq.ai;

import ca.swank.hardquor.*;

/**
 * @author Quinn McDermid
 */
public class fitgenAIUser
    implements hardquorUser {
    public static final int AI_TYPE_LOOKAHEAD = 4;
    public static final int AI_TYPE_TREE = 5;

    private String serverIp;
    private String userName;
    private hardquorClient client;
    private int selection;
    private int maxmin;

    /**
     * constructs a new aiUser which attempts to connect to the server as username
     */
    public fitgenAIUser(String serverIp, String userName, int select, int maxmin) {
        this.serverIp = serverIp;
        this.userName = userName;
        this.selection = select;
        this.maxmin = maxmin;

        client = new hardquorClient(this);
        client.connect(serverIp);
    }

    /**
     * called upon connection to a hardquor game server. Upon having this method
     */
    public void notifyRequestUserName(String reason) {
        System.out.println("username requested for reason: " + reason);
        client.tryUsername(userName);
    }

    /**
     * provides a list of online users upon connection to a hardquor game.
     */
    public void notifyUserList(String[] users) {
        System.out.println("received userlist: " + users);
    }

    /**
     * notification that another user has disconnected from the hardquor game
     */
    public void notifyUserGone(String username) {
        System.out.println("user gone: " + username);
    }

    /**
     * notification that another user has connected to the current hardquor game
     */
    public void notifyUserNew(String username) {
        System.out.println("user new: " + username);
    }

    /**
     * a chat message received from the specified user. Chat messages should
     */
    public void notifySpeak(String user, String message) {
```

```

        System.out.println(user + ": " + message);
    }

    /**
     * an incoming game request from another user on the current hardquor server.
     */
    public void notifyGameRequest(hardquorGameRequest request) {
        System.out.println("game requested");
        client.acceptGameRequest(request);
    }

    /**
     * requests the user create an instance of an implementation of the
     */
    public hardquorUserGame startGame(hardquorGameClient gameClient,
                                      int playerNumber) {
        System.out.println("game start");
        hardquorUserGame userGame = null;
        userGame = new GenAIGame(gameClient, playerNumber, userName, selection,
                                maxmin);

        return userGame;
    }

    public void startAIvsAI() {
        hardquorGameRequest gameRequest = new hardquorGameRequest(0, "GEN-FIT", 33);
        client.makeGameRequest(gameRequest);
    }

    /**
     * starts a new WalkerUser which accepts all requests on specified server and name
     */
    public static void main(String[] args) {
        if (args.length != 4) {
            System.out.println(
                "usage: aiUser [server_ip] [aiName] [selection] [maxmin]");
        }
        else {
            int value = 0;
            int maxmin = 0;
            char[] test = args[2].toCharArray();
            value = (int) test[0];
            if (value == 49) {
                value = 1;
            }
            else if (value == 50) {
                value = 2;
            }
            else {
                value = 3;
            }
            test = args[3].toCharArray();
            maxmin = (int) test[0];
            if (maxmin == 49) {
                maxmin = 1;
            }
            else if (value == 50) {
                maxmin = -1;
            }
        }

        System.out.print("FITNESS*****");
        fitgenAIUser user = new fitgenAIUser(args[0], args[1], 1, 1);
    }
}

```

8.2 Tournament Selection Player

```
package ca.swank.hq.ai;

import ca.swank.hardquor.*;

/**
 * @author Quinn McDermid
 */
public class tourgenAIUser
    implements hardquorUser {
    public static final int AI_TYPE_LOOKAHEAD = 4;
    public static final int AI_TYPE_TREE = 5;

    private String serverIp;
    private String userName;
    private hardquorClient client;
    private int selection;
    private int maxmin;

    /**
     * constructs a new aiUser which attempts to connect to the server as username
     */
    public tourgenAIUser(String serverIp, String userName, int select, int maxmin) {
        this.serverIp = serverIp;
        this.userName = userName;
        this.selection = select;
        this.maxmin = maxmin;

        client = new hardquorClient(this);
        client.connect(serverIp);
    }

    /**
     * called upon connection to a hardquor game server. Upon having this method
     */
    public void notifyRequestUserName(String reason) {
        System.out.println("username requested for reason: " + reason);
        client.tryUsername(userName);
    }

    /**
     * provides a list of online users upon connection to a hardquor game.
     */
    public void notifyUserList(String[] users) {
        System.out.println("received userlist: " + users);
    }

    /**
     * notification that another user has disconnected from the hardquor game
     */
    public void notifyUserGone(String username) {
        System.out.println("user gone: " + username);
    }

    /**
     * notification that another user has connected to the current hardquor game
     */
    public void notifyUserNew(String username) {
        System.out.println("user new: " + username);
    }

    /**
     * a chat message received from the specified user. Chat messages should
     */
    public void notifySpeak(String user, String message) {
        System.out.println(user + ": " + message);
    }
}
```

```

    * an incoming game request from another user on the current hardquor server.
    */
    public void notifyGameRequest(hardquorGameRequest request) {
        System.out.println("game requested");
        client.acceptGameRequest(request);
    }

    /**
     * requests the user create an instance of an implementation of the
     */
    public hardquorUserGame startGame(hardquorGameClient gameClient, int playerNumber) {
        System.out.println("game start");
        hardquorUserGame userGame = null;
        userGame = new GenAIGame(gameClient, playerNumber, userName, selection, maxmin);

        return userGame;
    }
    /**
    public void startAIvsAI() {
        hardquorGameRequest gameRequest = new hardquorGameRequest(0, "GEN-FIT", 33);
        client.makeGameRequest(gameRequest);
    }
    */
    /**
     * starts a new WalkerUser which accepts all requests on specified server and name
     */
    public static void main(String[] args) {
        if (args.length != 4) {
            System.out.println(
                "usage: aiUser [server_ip] [aiName] [selection] [maxmin]");
        }
        else {
            int value = 0;
            int maxmin = 0;
            char[] test = args[2].toCharArray();
            value = (int) test[0];
            if (value == 49) {
                value = 1;
            }
            else if (value == 50) {
                value = 2;
            }
            else {
                value = 3;
            }
            test = args[3].toCharArray();
            maxmin = (int) test[0];
            if (maxmin == 49) {
                maxmin = 1;
            }
            else if (value == 50) {
                maxmin = -1;
            }

            System.out.print("HERE STARTS THE BATTLE\n");
            System.out.print("TOURNAMENT *****\n");

            // value 1,2 maxmin 1,-1
            tourgenAIUser user = new tourgenAIUser(args[0], args[1], 2, 1);
        }
    }
}

```

8.3 Rank-Based Selection Player

```
package ca.swank.hq.ai;

import ca.swank.hardquor.*;

/**
 * @author Quinn McDermid
 */
public class rankgenAIUser
    implements hardquorUser {
    public static final int AI_TYPE_LOOKAHEAD = 4;
    public static final int AI_TYPE_TREE = 5;

    private String serverIp;
    private String userName;
    private hardquorClient client;
    private int selection;
    private int maxmin;

    /**
     * constructs a new aiUser which attempts to connect to the server as username
     */
    public rankgenAIUser(String serverIp, String userName, int select, int maxmin) {
        this.serverIp = serverIp;
        this.userName = userName;
        this.selection = select;
        this.maxmin = maxmin;

        client = new hardquorClient(this);
        client.connect(serverIp);
    }

    /**
     * called upon connection to a hardquor game server. Upon having this method
     */
    public void notifyRequestUserName(String reason) {
        System.out.println("username requested for reason: " + reason);
        client.tryUsername(userName);
    }

    /**
     * provides a list of online users upon connection to a hardquor game.
     */
    public void notifyUserList(String[] users) {
        System.out.println("received userlist: " + users);
    }

    /**
     * notification that another user has disconnected from the hardquor game
     */
    public void notifyUserGone(String username) {
        System.out.println("user gone: " + username);
    }

    /**
     * notification that another user has connected to the current hardquor game
     */
    public void notifyUserNew(String username) {
        System.out.println("user new: " + username);
    }

    /**
     * a chat message received from the specified user. Chat messages should
     */
    public void notifySpeak(String user, String message) {
        System.out.println(user + ": " + message);
    }
}
```

```

/**
 * an incoming game request from another user on the current hardquor server.
 */
public void notifyGameRequest(hardquorGameRequest request) {
    System.out.println("game requested");
    client.acceptGameRequest(request);
}

/**
 * requests the user create an instance of an implementation of the
 */
public hardquorUserGame startGame(hardquorGameClient gameClient,
                                   int playerNumber) {
    System.out.println("game start");
    hardquorUserGame userGame = null;
    userGame = new GenAIGame(gameClient, playerNumber, userName, selection,
                              maxmin);

    return userGame;
}

public void startAIvsAI() {
    hardquorGameRequest gameRequest = new hardquorGameRequest(0, "GEN-TOUR", 33);
    client.makeGameRequest(gameRequest);
}

/**
 * starts a new WalkerUser which accepts all requests on specified server and name
 */
public static void main(String[] args) {
    if (args.length != 4) {
        System.out.println(
            "usage: aiUser [server_ip] [aiName] [selection] [maxmin]");
    }
    else {
        int value = 0;
        int maxmin = 0;
        char[] test = args[2].toCharArray();
        value = (int) test[0];
        if (value == 49) {
            value = 1;
        }
        else if (value == 50) {
            value = 2;
        }
        else {
            value = 3;
        }
        test = args[3].toCharArray();
        maxmin = (int) test[0];
        if (maxmin == 49) {
            maxmin = 1;
        }
        else if (value == 50) {
            maxmin = -1;
        }
    }

    // value 1,2 maxmin 1,-1
    rankgenAIUser user = new rankgenAIUser(args[0], args[1], 3, -1);
    System.out.print("RANK*****\n");

    try {
        Thread.sleep(1000 * 10);
    }
    catch (Throwable t) {}

    System.out.print("HERE STARTS THE BATTLE\n");
    System.out.print("RANK *****\n");
    user.startAIvsAI();
}

```


9. Appendix B: Java Source Code for Simulated Annealing AI Player

```
package ca.swank.hq.ai;

import ca.swank.hardquor.*;
import java.util.TreeSet;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.Random;
import java.util.Vector;

import java.util.Comparator;

/**
 * @author Touran Raguimov
 */
/**
 * An intelligent "AI" player that uses a simulated annealing to decide upon a move.
 *
 * The player generates all possible next moves and then takes the best one or random
 * one.
 */
public class SimAnnAIGame
    extends hqSimpleAIGame {
    private static final int POPULATION_SIZE = 25;
    private static final int NUM_GENERATIONS = 3;
    private static final int INFINITY = Integer.MAX_VALUE;
    public static Random selector = new Random(System.currentTimeMillis());
    public static double _SCORE;
    private static final double WALK_PROB = 0.25;

    public SimAnnAIGame(hardquorGameClient gameClient, int me, String userName) {
        super(gameClient, me, userName);
    }

    /**
     * notification that a move required by the current player.
     *
     * The player initially speaks a comment to the board, and then examines the board.
     * If a loss is inevitable, the sore-loser player gives up. Otherwise a 'best' move is
     * made.
     */
    public void notifyTurn() {
        /* Get all the possible moves then:
         * if neighbour is better than current then move to the neighbour
         * else move to neighbour with prob  $e^{(curr-neigh)/temp}$ 
         * where temp controls the randomness vs. greediness
         */

        double h_curLoc, h_curGlob;
        double h_nextLoc = 0;
        double h_nextGlob = 0;
        double h_diff = 0;
        double exp, temp, prob, randProb;
        final double e = (double) 2.7182818284590452354;
        Move posbMove1 = null;
        Move posbMove2 = null;
        Move posbMove3 = null;
        Move nextMove = null;

        int TIME1 = 1;
        int TIME2 = 1;

        //-- GET ALL THE POSSIBLE MOVES --//
        Vector _possibleWalkMoves = board.findPossibleWalkMoves();
        Vector _possibleWallMoves = board.findPossibleWallMoves();

        int index = -1;

        //-- GENERATE PATH OF LENGTH 3 --//
    }
```

```

Move[] movePath = genPath(3);
h_curGlob = _SCORE;

//GLOBAL ANNAL//
for (int i = 0; i < INFINITY; i++) {

    TIME1++;
    double wallORwalk1 = selector.nextDouble();
    //-- RANDOM SELECTED POSSIBLE 1st MOVE --//
    if (wallORwalk1 > 0.5 && _possibleWallMoves.size() > 0) {
        index = selector.nextInt(_possibleWallMoves.size());
        posbMove1 = (Move) _possibleWallMoves.elementAt(index);
    }
    else {
        index = selector.nextInt(_possibleWalkMoves.size());
        posbMove1 = (Move) _possibleWalkMoves.elementAt(index);
    }

    //-- MAKE MOVE 1--//
    board.makeMove(posbMove1);
    //-- MAKE MOVE 2 OPONENT --//
    posbMove2 = selectWorstMove(board.findPossibleMoves());
    board.makeMove(posbMove2);
    //-- GET CURRENT SCORE AFTER OPONENT MOVE--//
    h_curLoc = board.genScore();

    //LOCAL ANNAL//
    for (int j = 0; j < INFINITY; j++) {

        TIME2++;

        double wallORwalk2 = selector.nextDouble();
        //-- RANDOM SELECTED POSSIBLE MOVE --//
        if (wallORwalk2 > 0.5 && _possibleWallMoves.size() > 0) {
            index = selector.nextInt(_possibleWallMoves.size());
            posbMove3 = (Move) _possibleWallMoves.elementAt(index);
        }
        else {
            index = selector.nextInt(_possibleWalkMoves.size());
            posbMove3 = (Move) _possibleWalkMoves.elementAt(index);
        }

        // LOCAL ANNEALING TO GET THE 3rd MOVE //
        board.makeMove(posbMove3);
        h_nextLoc = board.genScore();
        h_nextGlob = board.genScore();
        board.unMove(posbMove3);

        //-- DIFFERENCE --//
        h_diff = h_nextLoc - h_curLoc;
        //-- flip to minimize the score --//
        h_diff = -1 * h_diff;

        if (h_diff > 0) {
            nextMove = posbMove3;
            break;
        }
        else {
            temp = (double) TIME2;
            exp = (double) h_diff / temp;
            prob = (double) Math.pow(e, exp);
            randProb = selector.nextDouble();

            if (randProb <= prob) {
                nextMove = posbMove3;
                break;
            }
        }
    }
} // for local annealing //

//unmove
board.unMove(posbMove2);

```

```

board.unMove(posbMove1);

//GLOBAL ANNEALING 3rd move of PATH vs 3rd move of NEW nextMove//
//-- DIFFERENCE --//
h_diff = h_nextGlob - h_curGlob;
//-- flip to minimize the score --//
h_diff = -1 * h_diff;

if (h_diff > 0) {
    nextMove = posbMove1;
    break;
}
else {
    temp = (double) TIME1;
    exp = (double) h_diff / temp;
    prob = (double) Math.pow(e, exp);
    randProb = selector.nextDouble();

    if (randProb <= prob) {
        nextMove = posbMove1;
        break;
    }
}
} // for global annealing //

// wall move
if (nextMove instanceof WallMove) {
    WallMove wallMove = (WallMove) nextMove;
    gameClient.tryMove(wallMove.a1(), wallMove.a2(), wallMove.b1(),
        wallMove.b2());
}
// walk move
else if (nextMove instanceof WalkMove) {
    WalkMove walkMove = (WalkMove) nextMove;
    gameClient.tryMove(walkMove.source(), walkMove.target());
}

System.out.print("\n\n-----\n");
System.out.print("ANNEALING AI MOVE: Player 1\n");
System.out.print(board);
System.out.print("\nSCORE: " + board.genScore() + "\n");

} //notifyTurn()

//-- GENERATE PATH OF MOVES --//
private Move[] genPath(int depth) {

    int MAX_DEPTH = 3;
    Move[] path = new Move[MAX_DEPTH];
    Move nextMove;
    int i;
    for (i = 0; i < MAX_DEPTH; i++) {
        if ( (i % 2) == 0) {
            nextMove = selectRandomMove();
            board.makeMove(nextMove);
        } //if
        else {
            nextMove = selectWorstMove(board.findPossibleMoves());
            board.makeMove(nextMove);
        } //else

        path[i] = nextMove;
        if (board.getWinner() != hardquorBoard.PLAYER_NONE) {
            break;
        }
    } //for
    i--;
    _SCORE = board.genScore();

    for (; i >= 0; i--) {
        board.unMove(path[i]);
    } //for
    //END GEN PATH //
}

```

```

        return path;
    }

    //__ RANDOM MOVE __//
    public Move selectRandomMove() {
        Vector walkMoves = board.findPossibleWalkMoves();
        Vector wallMoves = board.findLocalWallMoves();
        if (wallMoves.size() > 0) {
            double someNumber = selector.nextDouble();
            if (someNumber < WALK_PROB) {
                return (Move) walkMoves.elementAt(selector.nextInt(walkMoves.size()));
            } //if
            else {
                return (Move) wallMoves.elementAt(selector.nextInt(wallMoves.size()));
            } //else
        } //if
        return (Move) walkMoves.elementAt(selector.nextInt(walkMoves.size()));
    } //selectRandomMove

    //-- WORST MOVE --//
    public Move selectWorstMove(Vector possibleMoves) {
        int bestIndex = 0;
        double bestScore;

        board.makeMove( (Move) possibleMoves.elementAt(0));
        bestScore = board.genScore();
        board.unMove( (Move) possibleMoves.elementAt(0));

        for (int i = 1; i < possibleMoves.size(); i++) {
            board.makeMove( (Move) possibleMoves.elementAt(i));
            double score = board.genScore();
            if (score < bestScore) {
                bestScore = score;
                bestIndex = i;
            } //if
            board.unMove( (Move) possibleMoves.elementAt(i));
        } //for

        return (Move) possibleMoves.elementAt(bestIndex);
    } //selectBestMove
} //SimAnnAIGame

```