# Monte Carlo Tree Search for Quoridor

**Conference Paper** · September 2018

CITATIONS
3

READS
6,493

**3 authors**, including:

Victor Massagué Respall
Innopolis University
**9** PUBLICATIONS **44** CITATIONS
SEE PROFILE

Joseph Alexander Brown
Thompson Rivers University
**136** PUBLICATIONS **569** CITATIONS
SEE PROFILE

# Monte Carlo Tree Search for Quoridor

Victor Massagué Respall, Joseph Alexander Brown and Hamna Aslam
Artificial Intelligence in Games Development Lab, Innopolis University
Innopolis, Republic of Tatarstan, Russia, 420500
Email: v.respall@innopolis.ru, j.brown@innopolis.ru and h.aslam@innopolis.ru

**KEYWORDS**
Monte Carlo Tree Search, Quoridor, Genetic Algorithm, Agent

**ABSTRACT**

This paper presents a preliminary study using Monte Carlo Tree Search (MCTS) upon the board game of Quoridor. Quoridor is an interesting game for expansion of player agents in MCTS due to having a mechanically simple rule set, however, Quoridor has a state-space complexity similar to Chess with a higher game-tree complexity. The system is shown to perform well against current existing methods, defeating a set of player agents drawn from an existing digital implementation as well as a previous method using a GA.

## INTRODUCTION

Monte Carlo Tree Search (MCTS) (Coulom 2006) (Kocsis and Szepesvári 2006) is a technique well-known these days (Browne, C. B. et al. 2012) due to the efficient results obtained in the board game *Go* (Silver D. et al. 2016). This game, produces difficulty for an AI expert to create an agent, due to its space-complexity, branching factor and difficulties to evaluate the state of the game in the middle. To deal with these, Monte Carlo tree search was used because of its following properties. It uses *UCT* (Upper Confidence Bound applied to Trees) (Gelly et al. 2006) for evaluating the final states of the game. Also, it consists of Monte Carlo rollouts, explained later in Section V, to estimate the value of each state in the search tree. As the tree grows larger more accurate values are generated. The average of these rollouts can provide an effective position evaluation achieving accurate performance in games such as Backgammon (Tesauro and Galperin 1997) and Scrabble (Sheppard 2002). This paper presents the research on the board game Quoridor to develop an artificial player agent using the Monte Carlo tree search algorithm and compares it with current existing agents.

## QUORIDOR

Quoridor (Marchesi 1997) is played in a nine by nine board. We focus only on the two-player version. Each player is represented by a pawn which begins at the center space in opposite edges of the board, the baselines. The goal is to be the first player to move their pawn from its side to the opposite side of the board, the opposite baseline.
The main feature that makes this game interesting and tactical is its fences. Fences are flat two-space-wide pieces which can be placed in the groove between the squares of the board. Fences have the ability to facilitate the player's progress or block the path of the pawns, which must go around them. Each player has ten fences at the start of the game, and once placed, cannot be moved or removed.
Each player at his turn can choose to move his pawn or to place one of his fences. Once the player runs out of fences, its pawn must be moved. Pawns are moved one square at a time, horizontally or vertically, forwards or backwards. When two pawns face each other on neighboring squares which are not separated by a fence, the player whose turn is it can jump over the opponent's pawn and place himself behind the opponent's pawn, thus advancing an extra square. If there is a fence behind the pawn, the player can place his pawn to the left or the right of the opponent's pawn. Fences may not be jumped, including when moving laterally due to a fence being behind a jumped pawn, see Figure 1.

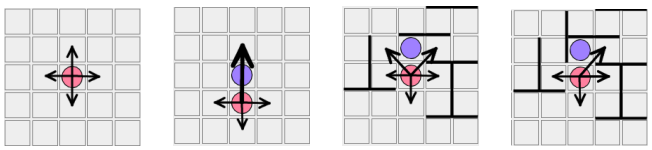

Figure 1: Allowed movements for the lower red pawn

Fences can be placed directly between two spaces, in any groove not already occupied by a fence. However, a fence may not be placed which cuts off the only remaining path of any pawn to the goal. The first player who reaches one of the 9 squares of his opponent's base line is the winner.
There is no official notation for Quoridor, so for this project, we use a notation from a community of players (Quoridor Strats 2014). The notation proposed is similar to algebraic Chess notation. Each square gets a unique letter-number designation. Each column is given a letter *A-I* and each row is given a number *1-9*. A move

is recorded as the column followed by the row. The first player always starts on *E1* and the second player always starts on *E9*. This marks the top and the bottom of the board, which are needed for recording fence placement. Each fence move is defined by the lower left square they touch along with their direction, horizontal or vertical. Every fence touches four squares, so we denote the position of the fence by the square closest to the *A1* corner of the board.

## PREVIOUS AGENTS FOR QUORIDOR

*Mastering Quoridor (Glendenning et al. 2005)*
Agent development has been done by implementing a search algorithm, using iterative-deepening alpha-beta negamax search algorithm with few modifications. For the evaluation function, the ten features chosen are summarized as: the Shortest path to the goal, Markov Chains, Goal Side, and Manhattan Distance for both players, and Pawn Distance and Number of fences of the current player. For the learning algorithm, a variant of a GA is using as a fitness evaluation function, the number of games that a chromosome wins against the others inside the population. This agent was shown to be easily beaten by a human.

*A Quoridor-playing Agent (Mertens 2006)*
MiniMax algorithm is used in this case with Alpha-Beta pruning, but the game tree is to large to perform MiniMax search all the way down to the leaves of the tree. Therefore, the solution is limiting the depth of the MiniMax search. Further, an evaluation function is applied to determine the value of a position in order to allow for a quicker return. The result obtained is a weak Quoridor agent as it is unable to see depth in the game. MCTS balances the want for a fast evaluation mixed with the ability to see beyond a set horizon depth.

## GAME COMPLEXITY

Quoridor has the number of possible ways to determine a fast upper bound on complexity, such as pawns moves multiplied by the number of possible ways to place the fences. Since the board has eighty one squares, we can place the first pawn in any of them, and the second one in eighty, due to the first pawn already placed on the board. Hence, the total number of positions, $S_p$, with two pawns is given by the following equation:

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

Further, for the fences, since each fence occupies 2 squares, there are eight ways to place a fence in one row. Given that there are eight rows, there are sixty four possible places to put a fence horizontally. Since the board is a square, we have the same number of rows and columns, one fence can be put in one hundred and twenty eight places. We have to take into account that one fence occupies four fence positions, except for the squares on the border. So the total number of positions of the twenty fences, $S_f$, and the upper bound of the size of the state space, $S$, are given by the following equations(Mertens 2006):

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

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

Quoridor has a state-space complexity similar to Chess and a higher game-tree complexity, shown to be $10^{162}$ in (Mertens 2006).

## MONTE CARLO TREE SEARCH

The Monte Carlo Tree Search algorithm is used for building the AI agent for Quoridor, as it appears to be an efficient algorithm for this type of board game and game tree size. It is a probabilistic search algorithm with a unique decision-making ability because of its efficiency in open-ended environments with an enormous amount of possibilities. To deal with the size of the game tree, it applies Monte Carlo method (Metropolis and Ulam 1949). As it is based on random sampling of game states, it does not need to use brute force.
We have built a game tree with a root node, then it is expanded with random simulations. In the process, we maintain the number of times we have visited a specific node and a win score, used to evaluate the state of the board. In the end, we select the node with best results and higher win scores.
This algorithm consists of four phases:

1. *Selection*: In this initial phase, the algorithm starts with a root node and selects a child node such that it picks the node with maximum win rate. In order to make sure that each node is given a fair chance to be selected and to balance the situation between exploration and exploitation, we use *UCT* (Gelly et al. 2006).

$$\frac{w_i}{n_i} + c * \sqrt{\frac{ln(t)}{n_i}} \tag{4}$$

Where

- $w_i$ = number of wins after the i-th move
- $n_i$ = number of simulations after the i-th move
- $c$ = exploration parameter (theoretically equal to$\sqrt{2}$ (Kocsis and Szepesvári 2006))
- $t$ = total number of simulations for the parent node

This formula ensures that agent will play promising branches more often than their counterparts, but will also sometimes explore new options to find a better node, if exists.

It selects the best node of the entire generated tree, traverses down the tree and selects a leaf node.

2. *Expansion*: When it can no longer apply UCT to find the successor node, it expands the game tree by generating all possible states from the leaf node.

3. *Simulation*: After Expansion, the algorithm picks a node randomly and it simulates the game until the very end, randomly for both players.

4. *Backpropagation*: This last phase consists of updating the nodes according to the result of the simulation. It evaluates the state to figure out which player has won and traverses upwards to the root incrementing visit counts and win scores, i.e. a count of if the player of that position has won, of each node visited.

   The algorithm keeps looping these four phases until some fixed number of iterations. Higher the number of iterations, more reliable the estimate becomes.

## IMPLEMENTATION

This project is implemented in the Java programming language, using standard libraries. This section explains main classes and data structures used.
The class *Board* has all the information related to the board game, squares, fences and both players. All squares are stored in an *ArrayList*, and to navigate through the board, we use a *Map* that each key is a square and it returns a *Set* of all adjacent squares to the key.
For checking if a certain fence can be placed, we initially start the game with a *Set* containing all possible places that a fence can be placed on the board. Then, each time a fence is placed, we update the *Set* removing "forbidden" positions. After checking that the position of that fence is inside the *Set*, we must check that both players are not completely blocked from reaching the opposing baseline. For this, a *BFS* (Breadth First Search) is applied which finds the shortest path to the goal for each player and considers it an illegal move if the distance is undefined.
The system explained before for storing fences is also useful for returning all possible movements available to the player.
Another important class is *Monte Carlo Tree Search*, that contains the tree and all the algorithm explained in the section before. The first attempt to implement the tree was including in each node the board of the game. The problem was since it contains a huge amount of information and data structures, the program ran out of memory just with 5000 simulations, because MCTS was generating too many nodes. The solution is, instead of saving the board in each node generated, we store the move that we should perform and the scores of the node. Then each time when we visit a node we need to compute the move in a temporal board. The score added to each winning node is 10.
Finally, instead of using random decisions in the simulation phase of Monte Carlo Tree Search, we improved our system by adding a heuristic. The heuristic helps us to balance the placement of fences and the moves of the pawn. Running MCTS with random simulation shows that our agent spends all fences at the beginning of the game and on average is better to save them until the middle of the game. Basically this heuristic consists of calculating shortest path for each player, and if the player at the turn, has a shorter path than the opponent, then it gives more chances to move the pawn rather than placing another fence.

## EXPERIMENTAL SETTINGS

Due to limited research on the game of Quoridor and no human Elo rankings, we cannot measure the level of the agent globally. However, we have tried to evaluate our player against self play with more simulation steps and against other player agent types to have an idea of the agent's ability:

*120k Simulations Agent:*
The default agent for running the experiments uses MCTS with the heuristic described before. It performs 120000 simulations of the game per decision.

*60k Simulations Agent:*
This agent was created to see the influence in the number of simulations of the game per decision. The only difference between this agent and the one stated in subsection *A* is that this agent is doing 60000 simulations per decision.

*Alternative Agent:*
To further evaluate the MCTS method, an alternative Quoridor agent base was found with four different agent levels *(Brain1, Brain2, Brain3 and Brain4)* (van Steenbergen 2006). *Brain1* simply moves the pawn at every turn without placing any fence. *Brain2*, places all fences at the beginning of the game, wasting all resources. *Brain3* places fences more strategically, but still the problem of placing all fences at the beginning of the game. That gives a lot of advantage to the opponent then. *Brain4* is the smartest agent, it focuses on reaching the goal and it uses fences during the middle of the game. It is sadly impossible to describe the algorithm that is using, because of the lack of information about the API from the developers.

*Genetic Algorithm Agent:*
This agent is based on a previous work, mentioned in Section III, Mastering Quoridor of Lisa Glendenning (Glendenning et al. 2005). For developing such an agent we used Minimax algorithm (Stockman 1979) with some modifications to improve its performance. It is possible to modify the game tree values to use just maximization operations, negating the returned values from the recursion. This approach is called *Negamax algorithm* (Campbell and Marsland 1983). However, the problem with Minimax search (Stockman 1979) is that the number of states it has to examine is exponential in the number of moves. For reducing this amount of moves it is applied *alpha-beta pruning* (Knuth and Moore 1975) (Pearl 1982) technique, that basically prunes away branches of the tree that cannot influence the final decision. Last modification applied was iterative-deepening due to the agent plays with time limit decision for every move. This allows us to return the best value computed until that point (Nilsson 1996).
After each execution of alpha beta, it is required an evaluation function for selecting the best state of the board. Eight features are proposed for Quoridor:

- Shortest Path Player (SPP), length of Breadth First Search path for the player
- Shortest Path Opponent (SPO),length of Breadth First Search path for the opponent
- Manhattan Distance Player (MDP), Manhattan distance for the player (straight distance from the player to the goal)
- Manhattan Distance Opponent (MDO), Manhattan distance for the opponent (straight distance from the opponent to the goal)
- Pawn Distance (PD), the distance between pawns using Breadth First Search
- Goal Side Player (GSP), boolean that tells if the player is between the midpoint of the board and the goal
- Goal Side Opponent (GSO), boolean that tells if the opponent is between the midpoint of the board and the goal
- Number Fences Player (NFP), number of fences of the player

Finally a genetic algorithm (GA) is used for weighting each feature described before. A chromosome is represented as a vector of weights and the fitness of each one is determined by the number of games that a chromosome wins against the other chromosome of the population. To create a population it is initialized with random float point values. It is established with a probability of 0.3 that a chromosome has a non-zero weight in some feature.

## RESULTS

For selecting the Genetic Algorithm between all chromosomes created inside a population, we performed a tournament consisting about creating a population of 10 chromosomes and each one plays against each other. This will give us the fitness of the chromosomes. The maximum number of moves to avoid infinite loops was set to 120, and the decision time to 10 seconds per move. For playing against our 120k agent we took the two best chromosomes of the tournament.
In all experiments performed, our player was the default, 120k Agent. The experiment with 60k Agent was done automatically for 250 games since it was easy to adapt the code to play against itself. However, as we do not have the source code for the alternative agents, the experiments with all of them were done manually, taking the moves from our agent and playing them against the four brains. In the evaluative process, first ten games were played against each of the four brain types, when there was no an obvious mercy situation, i.e. in the case of *Brain4*, then we extended this to 100 games in order to allow for a statistical evaluation.
Table 1 shows that reducing the amount of simulations to 60000, our agent performs a little bit better than 120k agent, and much faster to decide each movement. Though this result is not significant at $p < 0.05$. Note that in playing these games, in 40 instances the MCTS players began to alternate moving a pawn back and forth, not seeing an obvious solution to winning the game. We therefore can infer there is a situation in the game which such a delaying tactic has some amount of value, or for which there is not an obvious good strategy. This only happened in self play, and more analysis is required to understand what developed this situation, as there is no obvious tie state.
The first three brains of the alternative agent and Chromosome 2 and 5 of the GA agent were easy to beat, with significant results over the ten evaluations each the MCTS would win 100% of the time. However, with *Brain4*, it is not possible for our agent to win all the time, but still performed better that the opponent, significant at $p < 0.05$. There is no information about which method is used by this agent so we are unable to make a deep evaluation as to the play method which is able to at least give some challenge to MCTS. Finally, using (Glendenning et al. 2005) best chromosome achieved - Psi1, the MCTS was able to defeat it in all of the ten games played.

## CONCLUSIONS

We have created an MCTS agent for the board game Quoridor and compared it to a number of previous agent types, including reimplementation of a GA. This research work completed thus far focuses only in the two players version, Quoridor can be played with four play-

Table 1: Comparison of the 120k Agent to Other Agent Types (Significant at 95% Confidence in Bold using a Binomial exact test)

| Opponent | Number of Games Played | Percentage (Count) of Wins for 120k | Lower Confidence Bound (95%) | Upper Confidence Bound (95%) |
|---|---|---|---|---|
| 60k Agent | 210* | 46% (97 games) | 39.3% | 53.2% |
| **Brain1** | 10 | 100% | 69.2% | 100% |
| **Brain2** | 10 | 100% | 69.2% | 100% |
| **Brain3** | 10 | 100% | 69.2% | 100% |
| **Brain4** | 100 | 66% (66 games) | 55.9% | 75.2% |
| **Chromosome2** | 10 | 100% | 69.2% | 100% |
| **Chromosome5** | 10 | 100% | 69.2% | 100% |
| **Psi1** (Glendenning et al. 2005) | 10 | 100% | 69.2% | 100% |

*250 games in total were played with 40 being undecided, we only examine the difference of wins in decided games.

ers, each with the goal of taking their pawn the opposite end of the board, the two other players take their pawns horizontally from the side baselines.

We have used Monte Carlo tree search as the main algorithm. It is a probabilistic search algorithm, and a unique decision making because of its efficiency in open-ended environments with an enormous amount of possibilities. Also, we have added a heuristic to balance the placement of fences and the moves of the pawn, in the simulation phase of the MCTS algorithm. The results obtained from the experiments are not sufficient to determine precisely the level of our agent, but it gives us an estimation of how it will perform against humans. The 60k agent as shown in Table 1 appears to perform better than the 120k agent and has a shorter runtime. Though this study will need to extend in order to prove this trend to hold in larger cases.

Future work should take into account the improvement in the heuristic, to decide a better quality movement and consider more features of the game such as a number of fences of each player. This will build a solid strategy for an agent. Moreover, as used in AlphaGo, deep learning can also be used (Silver D. et al. 2016). Finally, it is the goal of the authors to show that this system is competitive against the ranked human play. There are a number of human play strategies which are used commonly in competitive play, much along the same lines as chess openings, and perhaps it would be best to add an evaluated game tree as an initialization step to ensure competitive play.

## REFERENCES


Browne, C. B. et al., 2012. *A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games*, 4, no. 1, 1–43.

Campbell M.S. and Marsland T.A., 1983. *A comparison of minimax tree search algorithms. Artificial Intelligence*, 20, no. 4, 347–367.

Coulom R., 2006. *Efficient selectivity and backup operators in Monte-Carlo tree search.* In *International conference on computers and games*. Springer, 72–83.

Gelly S.; Wang Y.; Teytaud O.; Patterns M.U.; and Tao P., 2006. *Modification of UCT with patterns in Monte-Carlo Go.*

Glendenning L. et al., 2005. *Mastering quoridor. Bachelor Thesis, Department of Computer Science, The University of New Mexico.*

Knuth D.E. and Moore R.W., 1975. *An analysis of alpha-beta pruning. Artificial intelligence*, 6, no. 4, 293–326.

Kocsis L. and Szepesvári C., 2006. *Bandit based monte-carlo planning.* In *European conference on machine learning*. Springer, 282–293.

Marchesi M., 1997. *Quoridor.* Family Games, Inc.

Mertens P.J., 2006. *A Quoridor-playing agent. Bachelor Thesis, Department of Knowledge Engineering, Maastricht University.*

Metropolis N. and Ulam S., 1949. *The monte carlo method. Journal of the American statistical association*, 44, no. 247, 335–341.

Nilsson N.J., 1996. *Artificial intelligence: A modern approach: Stuart Russell and Peter Norvig,(Prentice Hall, Englewood Cliffs, NJ, 1995); xxviii+ 932 pages.*

Pearl J., 1982. *The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. Communications of the ACM*, 25, no. 8, 559–564.

Quoridor Strats, 2014. *Notation.* Https://quoridorstrats.wordpress.com/notation/.

Sheppard B., 2002. *World-championship-caliber Scrabble. Artificial Intelligence*, 134, no. 1-2, 241–275.

Silver D. et al., 2016. *Mastering the game of Go with deep neural networks and tree search. Nature*, 529, 484–503. URL `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`.

Stockman G.C., 1979. *A minimax algorithm better than alpha-beta? Artificial Intelligence*, 12, no. 2, 179–196.

Tesauro G. and Galperin G.R., 1997. *On-line policy improvement using Monte-Carlo search.* In *Advances in Neural Information Processing Systems*. 1068–1074.

van Steenbergen M., 2006. *Quoridor.* online. Http://martijn.van.steenbergen.nl/projects/quoridor/.