CSCI 4511W Project: Quoridor Agent Efficacy

Mason Zarns

December 21, 2021

Introduction

When two humans play a board game like Quoridor, anticipating what the opponent might do given different board states comes naturally because both players understand the game in a similar way. For a software agent to defeat a human in such a game, it must first be able to anticipate how an opponent might affect its path to victory. Where single player games might be a cakewalk, the existence of a second agent presents a greater challenge. To understand a few different strategies in addressing this complication, it is key to understand different decision tree modeling systems and related search algorithms.

1 Related Literature

Minmax

Quoridor is a zero-sum game, meaning that if one player is benefitted, the other player is equally detrimented. In other words, the victory of the opponent is the loss of the agent. The minmax algorithm exploits this fact by estimating which player is in a better position to win in a given board state and predicting that the opponent will always choose the decision leading to the board state that most heavily favors them.

Mertens explores how depth of minmax search trees can effect performance in the game of Quoridor [6]. Mertens struggles with implementing heuristic functions that can build an agent capable of defeating a skilled human player. A comparison with other algorithms would be incomplete without a detailed look at many ideas for heuristics that can more properly characterize a game state. Mertens suggests the creation of some function to optimize the weighted values he used in the future. However, this would still not rule out the existence of more cleverly structured heuristics. Maybe certain shapes of placed fences near a player should be assigned values and multiplied in importance by the number of opponent fences. Comparisons can be performed, but there is no limit to the creative combinations of heuristics for this game.

Expectimax

Chowdhury explores a stochastic, single agent game, 2048 [4]. This problem is fundamentally different from Quoridor because it has only one agent and the result of each choice depends on chance as to where the next tile will spawn. Naturally, this breeds problems in comparing the algorithms for each game. However, this paper focuses on expectimax, anticipating the likelihoods

of convenient and highly inconvenient tile placement makes an enormous difference in longevity in the popular game 2048. However, while comparing the algorithms presented for Quoridor, it will be interesting to see how well expectimax fares without having a purely stochastic component to the game as in 2048.

Similarly, Yarasca compares the algorithms of Expectimax and Monte Carlo Tree Search for the game 2048 [8]. Expectimax was given five heuristics: maximum tile value in a corner, maximum empty cells, monotonicity in a snake path (bigger tiles closer to corner), max tile on an edge (interesting to combine with max tile in a corner), and matching neighbor tiles. A convenient path for analysis is provided when comparing two algorithms that can use identical heuristic functions. Comparing MCTS to the other proposed algorithms is worrisome because the outcomes are highly dependent on the heuristic strategies and weights. One obvious function for a board game like Quoridor is minimum distance from goal state. However, that is not nearly enough. 2048 uses 5 heuristics and some are fairly similar. This raises a question of how many different functions could be used in Quoridor before the added functions start to decrease performance more than they improve optimality and how much optimization is reasonable to compare the algorithms fairly.

Monte Carlo Search

Clearly, the necessity of heuristic functions make algorithms like Minmax and Expectimax a little flimsy or maybe just inconvenient. A more elegant strategy might require no heuristics at all. The Monte Carlo search The elegance of MCTS is that there is no need for this conjecture.

Coulom explains that MC search can be separated into a min-max phase alternating with a MC search phase [5]. To improve on this symbiosis, he proposes combining these phases and eliminating the distinction. For Crazy Stone, he examines how the error of the mean and max values compare over thousands of simulations with the "true" values. These "true" values he obtains by running twice as many simulations from that point. Then he tailors a linear combination of the mean and max values such that the average error from the "true" value is no longer biased positive or negative. This source is useful to understand how MCTS was first conceptualized, but it will likely not play a major role in my analysis due to being outdated.

Brenner provides a prolific in-depth backstory of the creation, theory and applications of the Monte Carlo Method [1]. This work actually focuses on a modified version of Quoridor, where spells and tile characteristics can be added or modified before and during the game. This means any agent made to play the game has to be equipped to deal with whatever rule modifications the human player sets. This dramatically increases game complexity with a whole new class of spell-related move options. Fun as this may be, much of these added complexities can easily be sidelined to help with my work.

Samothrakis implements MC search in a less traditional application, Ms Pacman [7]. This game can have four ghosts at a time, is played in real time instead of turns, and the ghosts do not seek to maximize or minimize anything. Usually, MC search is applied in two player, turn-based, zero-sum games. Naturally, a major choice had to be made about how long the tree can be expanded for; a human player might turn back and forth and effectively maintain their position. The proposed algorithm treats the game as turn-based with 50-60 milliseconds between each turn. That way, quick adjustments can be made while allowing the tree to be searched deeper. This source may be helpful in establishing handicaps for the different algorithms to match up runtimes so that the other variables can be examined more carefully.

Browne gives a comprehensive explanation of the Monte-Carlo Search algorithm and variations

with other strategies [2]. The length of the paper speaks to the universal utility of MC search that stems from its lack of heuristic functions. Browne emphasizes that one attractive aspect of MC search is that the algorithm is its own heuristic value, making application to different types of games incredibly easy. This source may be particularly useful in implementing alpha-beta pruning with MCTS.

Alpha-Beta Pruning

An interesting addition to the fray is a technique for "pruning" or disregarding portions of a tree that are deemed unnecessary or undesirable to search. It is commonly used in conjunction with Minmax to eliminate portions of a tree that are predicted never to be traversed (given a rational opponent).

Chaslot gives a brief overview of how MC search works [3]. Then, he outlines the general types of problems and games that are most appropriate for the algorithm. In traditional board games, often played sequentially with perfect information and two players, the algorithm is compared to alpha-beta pruning. One strength MC has over alpha-beta pruning is its performance in highly branched trees. In modern RTS video games, the MC search is instead suggested as a testing algorithm for other AI agents. It is interesting to see how quickly the algorithm landscape is changing: these algorithms were viewed as separate at the time, while alpha-beta pruning is now considered as more of an add-on to other algorithms.

2 Approach

Upon preliminary testing, several key observations were made that would alter the course of the experiment. First, there is no element of chance involved in a simulation between two of these algorithms at given depths and heuristic values, so repeated tests will always result in the same final board. Second, almost all simulation configurations resulted in player two winning (the agent that moves second). Third, the code from https://github.com/dimitrijekaranfilovic/quoridor was found to be far more buggy than advertised. A significant period of time was spent refactoring and debugging the code for testing.

3 Experimental Design

The goal of the experiment was to determine a loose threshold of where each of these algorithms outperformed their peers and where their performance dropped off sharply. Performance could be conceivably understood in terms of average turn time and win rate against different opponents. Measuring performance against a human player may be useful in future experiments with a great sample size, but for this experiment, the human factor was removed from consideration by pinning the agents against each other. Recognizing the trend of the second agent being favored dramatically and testing with reversed roles made a staggering difference. All data was collected such that one game was played with algorithm1 as the player1 and then a second game as player2. Every turn time for both games and both algorithms was recorded and averaged accordingly. In certain cases, the tree depths were adjusted to allow discovery of the regions over which the algorithms are in closer competition.

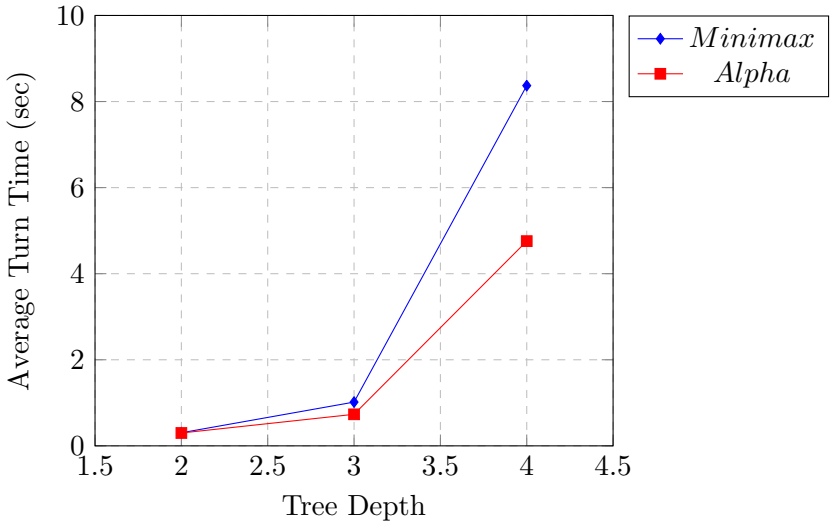
Comparing Minimax to Alpha-Beta Pruning was a simple case where increasing tree depths of both algorithms simultaneously led to the most valuable observations. Expectimax, on the other hand, required more tree depth comparisons to gauge its ability. Increasing its search depth beyond two caused it to lose nearly every game it played. Meanwhile, a depth of 2 for expectimax actually stood its ground against other algorithms. This data wasn't as neatly understood in graph form, so only a table was constructed. Finally, MCTS was fairly straightforward to compare with the other algorithms, because tuning its number of searches per turn only increases its turn time in linear time (unlike the other algorithms).

4 Results

4.1 Minimax v Alpha-Beta Pruning

Tree Depth	Minimax Time	Alpha Time	Alpha Win Ratio
2	0.3042	0.2998	0.5
3	1.017	0.733	0.5
4	8.378	4.757	0.0

Table 1: Minimax v Alpha-Beta Pruning



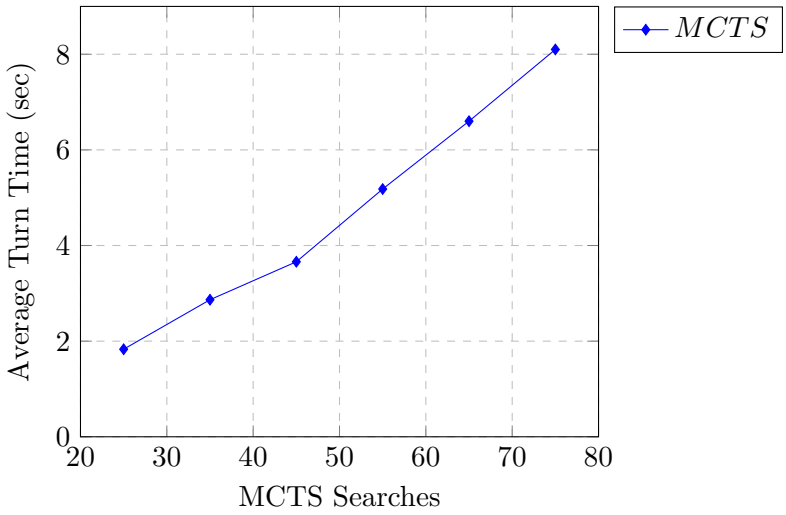
4.2 Minimax v Expectimax

Minimax Depth	Expectimax Depth	Minimax Time	Expectimax Time	Expectimax Win Ratio
2	2	0.508	0.257	0.5
3	2	0.986	0.228	0.5
4	2	12.43	0.299	0.5
2	3	0.509	0.354	0
2	4	0.484	7.798	0

Table 2: Minimax v Expectimax

Expectimax was modified to allow its depth to be increased from the preset value of 2. However, this modification caused it to perform progressively worse at higher depths, being defeated by the Minimax algorithm at a lower depth setting.

4.3 Minimax v MCTS



The implementation of MCTS lost every single game it played. It was even defeated by Minimax at depth = 1, which is effectively A*. Even at outrageous turn times, zero wins were observed.

5 Analysis

Perhaps the most significant observation was that in nearly all of the data samples, whichever algorithm was assigned player two won that game. The code was scoured for some sort of bug that would favor the second player, but none were found. Since three of the algorithms use the same heuristic function, there is a possibility that this state evaluation function is somehow more effective for player2. There is also a possibility that the game itself actually favors player2. However, board game enthusiasts have praised the game for years without complaint about one position being favored. Another possibility is that it has something to do with the jumping move mechanic. If both players move straight forward, player 2 will have the opportunity to jump over player 1 and will actually reach its goal first.
One theory is that, for human players playing as player 1, placing a wall before player 2 can jump over them is an obvious best choice. The heuristic tends to favor movement over wall placement even in situations like this one. Beyond that, the heuristic seems to struggle with making the "right" wall placement choice. Across samples with different algorithms at different search depths with the same heuristic, agents were observed to place walls in arguably ludicrous positions, sometimes behind the opponent or to their sides. Then, for the rest of the game they never built onto those walls. Furthermore, common patterns of wall placements were observed. For example, a strange "E" shape emerged fairly frequently even in situations where completing the "E" was clearly not an optimal use of the wall. Perhaps the fatal flaw of the experiment was that in most games agents would neglect to use walls in situations where they had more than plenty of walls, and the other agent was one move away from victory.

The first algorithm comparison yielded interesting results. At tree depth, the algorithms had nearly identical turn times and appeared to be matched evenly. When tree depth reached 4, Alpha-Beta Pruning really began to shine in turn time, it nearly cut its opponent's turn time in half. However, it lost both games played, which in this experiment is a clear sign that the winning algorithm played better. This all makes perfect sense from understanding how these two algorithms are related, Alpha-Beta Pruning apparently pruned a branch of the tree that was more optimal than its chosen option. This wouldn't be possible with an admissible heuristic function, but the current implementation is anything but.
Expectimax yielded interesting results, at a tree depth of only 2, it outperformed every other agent in with depths up to 4. It did this while keeping a staggeringly low turn time. With the current code, this is the clear winner, with the fastest turn times and the most successful performance. It appears that the implementation synergizes with the heuristic function; when the heuristic is called on similar states and then the values are averaged, the whole seems to become greater than the sum of its parts. Maybe with further work on the code, the current implementation can become more effective with higher tree depths.
MCTS yielded terribly disappointing results. In researching MCTS, its greatest weakness is arguably in environments with high branching factors. Unfortunately in Quoridor, on any given turn the pawn can be moved in 1-5 directions, but a wall can be placed in any of 128 positions. In its current state, the MCTS takes over five seconds just to explore 55 branches. This means it is guaranteed not to choose any of the routes it originally searched because the other positions have effectively infinity priority thanks to dividing by zero.

6 Conclusion and Future Work

The results obtained here were unsatisfactory overall, but some key expectations were observed. For example, the relationship between Minimax and Alpha-Beta Pruning was exactly as expected. For Expectimax and MCTS, further work needs to be done to the code to obtain more meaningful results. MCTS was taking very long turns, without even doing more than 100 searches. This performance needs to be heavily refactored to optimize for speed. Also, it would make a very interesting project to set a depth limit and use a heuristic whenever a search reaches that depth. It wouldn't be pure MCTS but the game has a deceptively high branching factor that seems to render this experiment's implementation useless. Perhaps another strategy would be to preferentially explore wall positions adjacent to other walls or players. This could greatly reduce the effective branching factor and may not decrease performance substantially because human players almost never place a wall that isn't either touching another wall or immediately next to one of the players.
The first thing I would improve in the heuristic is recognizing when the opponent is one move away from goal state. This would be a very cheaply handled special case because there are only two logical choices: a wall placement to the left or to the right. Ideally, this wouldn't need to be a special case and the heuristic would naturally prioritize the opponent's position as the goal state become closer. That said, it is a fairly common situation where placing a wall isn't immediately necessary, but if not placed, a great opportunity to increase the opponent's path to goal is lost. Of course, this complexity reveals the difficulty in making a heuristic.
A final note is that the expectimax code needs some further tweaking to allow for deeper trees without the dramatic dropoff observed here.

7 Coding Contributions

I should have tested the code base more rigorously before I chose to work with it. There were a number of problems that strayed from the game's rules and logic. For example, once in a while player 1 would place a wall but it would only have a length of one. MCTS as it was presented was nothing more than a random number generator because the branching factor is larger than the total number of searches (and turn time is already as high as tolerable).

References

[1] M. Brenner. Artificial intelligence for quoridor board game. 2015.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[3] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. *AIIDE*, 8:216–217, 2008.

[4] G. Chowdhury and V. Dhamodaran. 2048 using expectimax.

[5] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[6] P. J. Mertens. A quoridor-playing agent. *Bachelor Thesis, Department of Knowledge Engineering, Maastricht University*, 2006.

[7] S. Samothrakis, D. Robles, and S. Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154, 2011.

[8] E. N. Yarasca et al. Comparison of expectimax and monte carlo algorithms in solving the online 2048 game. *Pesquimat*, 21(1):1–10, 2018.

8 Appendix

8.1 Expectimax Algorithm

```
def expectimax(game_state: GameState, depth, maximizing_player, player_one_maximizer):
if depth == 0:
    return state_evaluation_heuristic(game_state, player_one_maximizer, True)
if maximizing_player:
    return max([expectimax(child[0], depth - 1, False, player_one_maximizer) for child in
                game_state.get_all_child_states(player_one_maximizer)])
else:
    values = [expectimax(child[0], depth - 1, True, player_one_maximizer) for child in
              game_state.get_all_child_states(player_one_maximizer)]
    return sum(values) / len(values)
```

8.2 Testing Method

```
    def test(self):
        while True:
            print()
            self.game_state.print_game_stats()
            print("\n")
            self.game_state.print_board()
            print()

            if self.check_end_state():
                p1times = []
                p2times = []
                print()
                self.game_state.print_game_stats()
                print("\n")
                self.game_state.print_board()
                print()
                for i in range(len(self.execution_times)):
                    if i%2==0:
                        p1times.append(self.execution_times[i])
                    else:
                        p2times.append(self.execution_times[i])
                return [sum(p1times)/len(p1times),sum(p2times)/len(p2times)]

            if self.game_state.player_one:
                if not self.game_state.is_simulation:
                    self.player_one_user()
                else:
                    res = self.player_simulation(1)
                    sleep(.1)
                    if not res:
                        break
            else:
                res = self.player_simulation(2)
                if not res:
                    break

            self.game_state.player_one = not self.game_state.player_one
```