CS63 Spring 2022
MCTS Checkers

Gyan Bains, Annie French

8 May 2022

1 Introduction

The goal of our project was to develop an intelligent checkers program. Our initial goal was to build a program that could beat a player that makes random moves 100 percent of the time, and beat a human about 50 percent of the time. We knew that beating a human player was an ambitious target, but at the minimum we wanted our checkers program to make reasonably intelligent moves.

We decided that using Monte Carlo Tree Search would give our program the best chance of being successful. Monte Carlo Tree Search is a decision making method for games that has garnered interest because of its success in playing Go, a game that has proven difficult for AI to tackle. Monte Carlo Tree Search consists of four main steps; Selection, Expansion, Simulation, and Backpropagation. In the context of checkers, Selection is the process of starting at the current board and recursively selecting child boards until a board is reached that is not fully expanded. In the selection stage, the program uses the UCB constant to determine how much to favor exploration, which involves trying new nodes, over expansion, which is when the agent follows nodes that have had good results in the past. During the Expansion step, if a board is not a terminal board, one of the children of that board is chosen. We can call that child board C. Simulation consists of running a specified number of simulated rollouts from C until a terminal board is reached. Finally, Backpropagation is the step where the tree is updated with the results from Simulation, so that the win/loss statistics can inform future moves.

Our board is shown in figure 1, with the red and blue circles representing pieces for each of the players. We represented a possible move by a tuple of the form `(row, col, vdelta, hdelta)` where `row` and `col` give the position of the piece on the board, and `vdelta` and `hdelta` give the vertical and horizontal movement of the piece respectively. A positive `vdelta` indicates the piece will move down on the screen, since its row number increases. A positive `hdelta` means the piece will move to the right, since its column number increases. One change we would have liked to implement but did not is including row and column numbers to make it easier for a human to select which piece they would like to move.

Once a player moves their piece to the row farthest from their starting side, it becomes a king, which we represented with a triangle. Players take turns either moving a normal piece forward diagonally to an open space or jumping a piece forward over an opponent, capturing the opponent's piece. A king can move and jump either forward or backward. The goal is to capture all of the opponent's pieces, or to get the game to a state where the other player has no available moves. Since the board becomes sparse toward the end of the game, making kings becomes easy to achieve. This

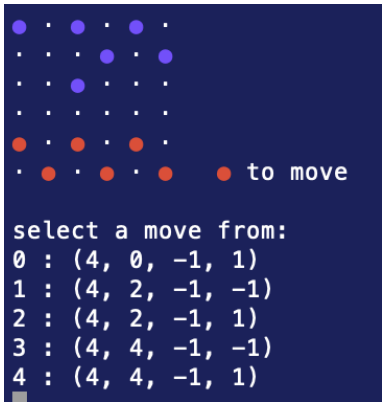

Figure 1: 6×6 checkers board where blue has made one move

means the game usually ends when one player has no more pieces, and the second end condition is much less common.

The rules of our implementation of checkers are similar to the traditional rules, with a few notable exceptions. As seen in figure 1, our board is only 6x6 instead of the usual 8x8. Given the smaller board size, each player starts with six pieces instead of the usual twelve. We found that using a smaller board drastically improves run time while still maintaining the integrity of the game. Furthermore, our rules for capturing pieces differ slightly from the traditional regulations. In our implementation, capturing a piece is optional and capturing multiple pieces in one turn is not allowed. These changes simplified the implementation and allowed our program to play to a slightly higher level without sacrificing the main principles of the game.

2 Methods

In our implementation of Checkers, we fully utilized king pieces, which are created once a piece reaches the far row and can move in any direction. We did not get to implement double or triple jumps, which are allowed when a player can make a series of legal jumps in a row. However, we do not think that the omission of multiple jumps significantly affects the outcomes of the game or how we assess the MCTS agent.

We primarily used a 6×6 game board, where each player starts with 6 pieces staggered across the two rows closest to the player. We did implement and test a traditional 8×8 board, but it took significantly longer to run through one game. While MCTS still worked correctly on the larger board, we felt the smaller board was sufficient to perform our tests, and the limited number of move options gave each game a more reasonable runtime of one to two minutes.

Initially, our Monte Carlo Tree Search used 1000 rollouts and a UCB constant of 1.0. The amount of training the agent received was determined by the number of rollouts we set in the `MCTSPlayer`, which it performed each time it decided which move to make next. We then performed experiments using 100, 1000, and 3000 rollouts, We also tested UCB constants of 1.0, 0.9, and 0.75 with 1000 rollouts.

3 Results

Initially, our program was able to beat a random player with ease. However, when faced with a human opponent, it became clear that the program was nowhere near winning. We were not able to find the parameters for an MCTS agent that could beat a human, but the MCTS agent did make reasonable moves. These reasonable moves include capturing an opponent's piece and making a king when possible.

We noticed that MCTS performed better at the beginning of a game than at the end. When there were only a few pieces left on the board, it had trouble making any "strategic" moves, and appeared to be moving the one or two pieces around the board randomly. Especially when playing against a random player, this caused games to go on for a larger number of turns than we would normally expect. We observed times when our MCTS agent would fail to capitalize on a jump opportunity when there were few pieces on the board.

Towards the beginning of games, we also noticed that while our program usually makes logical moves in short term, it fails to consider the broader strategy of the game. For example, when presented with the opportunity to capture a piece or make a king, it will do so, but it will make strategic errors like leaving one side of the board completely open, leaving easy paths for its opponent to make kings. Additionally, it fails to adhere to well established checkers strategies like controlling the center of the board and advancing en masse. These tactical errors prove to be detrimental to the MCTS agent's success beyond the opening moves.

The first experiment we performed was testing our MCTS with 1000 rollouts against a player that chooses random move. Our MCTS agent won 10 out of 10 games, usually by a significant piece differential. 9 out of 10 times, MCTS won by 4 or 5 pieces. In the following tables, APD is the average piece differential for the games that the specified agent won.

	MCTS 1000	Random
Wins	10	0
APD	4.3	0

Figure 2: MCTS with UCB = 1.0 and 1000 rollouts against Random player

MCTS with 1000 rollouts won 10 out of 10 games against MCTS with 100 rollouts, as we expected.

	MCTS 1000	MCTS 100
Wins	10	0
APD	3.6	0

Figure 3: MCTS 1000 rollouts vs 100 rollouts, both with UCB = 1.0

The difference between 1000 and 3000 rollouts is less clear, since 3000 rollouts only beat 1000 rollouts 7 out of 10 times.

	MCTS 1000	MCTS 3000
Wins	3	7
APD	3	3.7

Figure 4: MCTS 1000 rollouts vs 3000 rollouts, both with UCB = 1.0

Using 1000 rollouts, a UCB constant of 1.0 beat a UCB constant of 0.9 in 6 out of 10 games.

	MCTS 1.0	MCTS 0.9
Wins	6	4
APD	3.7	3

Figure 5: MCTS with UCB = 1.0 vs UCB = 0.9, both with 1000 rollouts

However, a UCB constant of 0.75 beat a 1.0 constant 6 out of 10 times, so the optimal UCB constant based on the experiments is 0.75.

	MCTS 1.0	MCTS 0.75
Wins	4	6
APD	2.8	3

Figure 6: MCTS with UCB = 1.0 vs UCB = 0.75, both with 1000 rollouts

We tried using 3000 rollouts to see if it could beat a human player. Unfortunately, our MCTS agent does not have enough strategy to win against human decisions, and the human player won 10 out of 10 games.

	MCTS 3000	Human
Wins	0	10
APD	0	2.7

Figure 7: MCTS with UCB = 1.0 and 3000 rollouts vs Human player

Overall, the best MCTS agent we tested used 3000 rollouts and a UCB constant of 0.75, but these parameters did not win 100% of the games against the other parameters we tested.

4 Conclusions

Ultimately, our Monte Carlo Tree Search Checkers player was unsuccessful in beating a human player but it still played to a respectable level, only losing by an average of 2.7 pieces across a sample of 10 games. The MCTS agent's endgame performance was particularly detrimental to its chances of overcoming a human opponent. This phenomenon has a few possible explanations but we believe that it can be attributed to the high number of kings and fewer total pieces on the board.

Since kings can move in four directions and there are fewer pieces to block potential moves, the agent has to consider a broader set of possibilities. However, just by trying to make as many kings as possible, avoiding moves to positions where your pieces can be captured, and keeping your kings close to the center, most players can play a strong endgame. Because the endgame can be played by trying to maintain a specified set of positions, there is potential for informed search methods to be successful in this portion of the game. Future research could explore the performance of informed search and Monte Carlo Tree Search in checkers endgames. As far as attaining our goals, we were successful in building an MCTS agent that beats a random player in every game. However, our agent did not perform as well against human opponents. It started games in a promising manner, but its failure to consider broader strategy and its poor endgame performance left it vulnerable to defeat.

Based on the experiments we performed, the most effective number of rollouts was 3000 and the most effective UCB constant was 0.75. This conclusion is based on 10-game experiments comparing 100, 1000, and 3000 rollouts, and tests that compared UCB constants of 1.0, 0.9, and 0.75.