Mason Hix

CS557

Checkers Player Using Monte Carlo Tree Search

Monte Carlo Tree Search

In the previous checkers player made, I used an alpha beta pruning algorithm which resulted in always winning against a random player and performed reasonably well against other players. This player will utilize the alpha beta pruning algorithm in conjunction with a Monte Carlo Tree Search (MCTS) algorithm to hopefully play an even better game of checkers.

The MCTS algorithm relies on playing several hundred to thousand of games using a very quick playout and evaluation on a random move available. With a large amount of data that started out as random, an optimal move can be selected by seeing how many times a move was selected and how many times it won and back propagating the data up the search tree. Figure 1 shows a cartoon showing in essence how this works. The tree is built with nodes, each with children and parents, and the nodes will eventually have utility values that get propagated up back to the root.

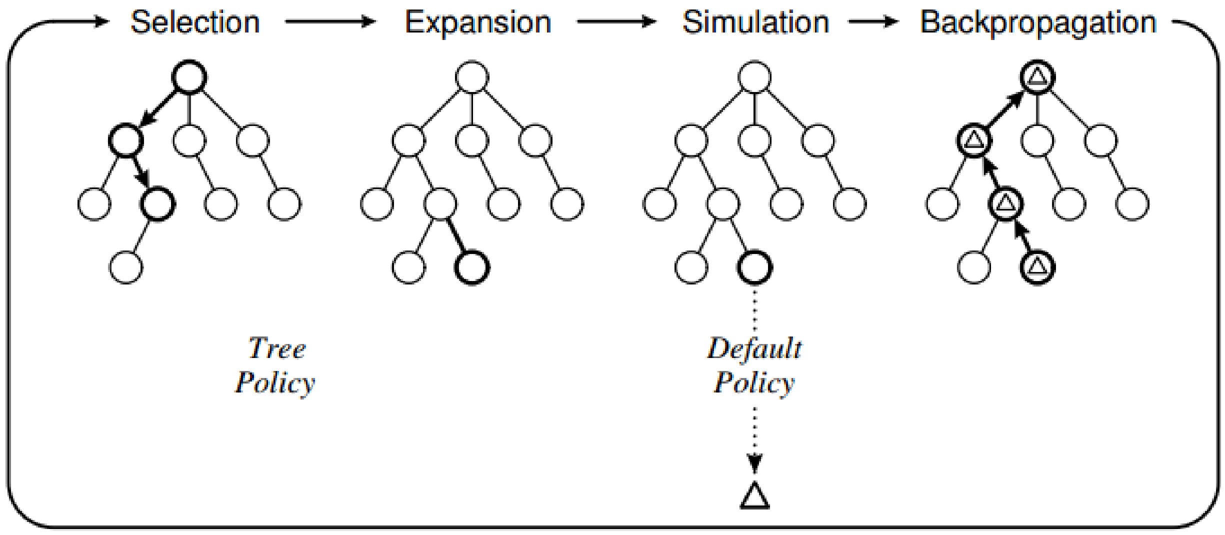


Figure 1: A cartoon showing the basics of MCTS

The concept of a Node was implemented using a class in java so I could easily keep track of the parent, number of plays, and number of wins. We went over most of this in class so I won't go into great detail about it. To briefly sum it up, when trying to find the best move the root node is created and selected. When a Node is selected, it will be expanded to generate the child nodes which contain the states (checkers moves) possible from the parent node. Each of these children is then simulated using a depth 3 alpha beta pruning algorithm to determine if this resulted in a win, loss, or tie. The values are propagated back up to the parent, and then this process repeats over and over again until the time limit is reached. With a number of wins/losses/games played, the utility of a node can be determined. The parent will then, in all likelihood, play more games on the node with the best utility to ultimately find the best move.

Implementation

Writing the code for the MCTS player was fairly straightforward, the difficult part was understanding the concepts of Nodes, parents, and children. The parent and children have opposite perspectives on what is a win or a loss, so this perspective is frequently swapped and can cause a lot of confusion. If even one of these is wrong, which happened in my case, it will have drastic consequences on the player. For instance, when writing the code that expanded a Node and generated the children, I understood it as the children only exist in an idea and the parent is still what is being played out; however, the way I have it set up is the children contain the state of the move to be performed. This means running the playout on the parent would not be practical as I already have the children created with the next state available, so it would make more sense to run the playout here and backprop the data up to the parent. This goes somewhat against the textbook, but follows Figure 1. Either way, they both end up doing the same thing so long as the perspectives are consistent. Once this was set up, it just has to run a playout using alpha beta pruning to see if it won, lost or draw, and then these feed into the utility function. To start, the utility function was set to the Upper Confidence Bound function as shown in Eq. 1, where Xn is the immediate reward of node n, C is the exploration constant, t is the timestep (number of times the parent was played), and n is the number of times node n was played out. The immediate reward is defined as the number of losses compared to plays for this node as the utility is from the perspective of the parent. To start out, C was set to the square root of 2. The impact of this value will be explained in the Utility section.

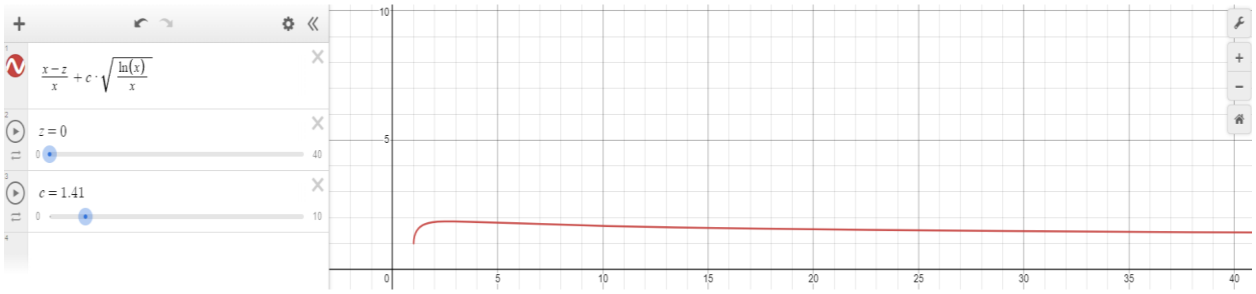
$$Utility = X_n + C\sqrt{\frac{\ln(t)}{n}}$$

Equation 1: Upper Confidence Bound Utility.

First Pass Results and Improvements

The results of the first draft MCTS algorithm were promising, but could be improved. In summary, in a simulation of 30 games against each player it always beat random, mostly beat d5, and sometimes beat diff and rat. I expected this to perform better, but it was a good start.

The first idea was to change the value of C to something other than the square root of 2. The exploration constant controls how often an exploratory move is selected even if it isn't optimal. Changing the value of C and running 30+ simulations would take too long, so I took a more mathematical approach. Eq. 1 was plotted graphically to show which factors most directly affect the utility. Fig. 2 shows the results of the UCB formula with C equal to 1.41, roughly the square root of 2, and t equal to the number of plays of the current node for simplicity. Assume the current node never wins so the immediate reward is always maximized. There is a small bump in utility when the number of plays is small, but after that it is mostly stable.



$$\frac{x-z}{x} + c \cdot \sqrt{\frac{\ln(x)}{x}}$$

$z = 0$

$c = 1.41$

Figure 2: Graphically plotting the UCB formula assuming the current node is the only node played from the parent, immediate reward is always maximized, and C = sqrt(2)

If I modify this to have a very large C value, say 8, it will increase the value of all utility and make the utility for small numbers of play more prominent. I am not very concerned with all utilities increasing because it is normalized so it mostly just affects the Nodes who have not been played very often.

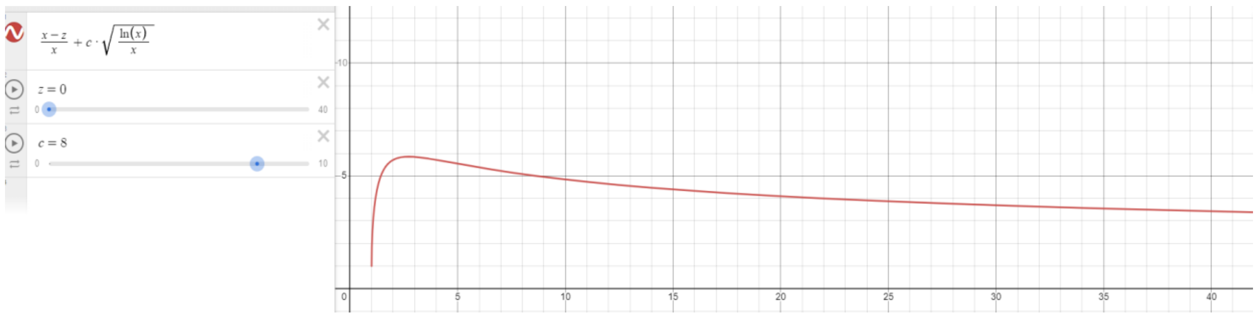


Figure 3: Graphically plotting the UCB formula assuming the current node is the only node played from the parent, immediate reward is always maximized, and C = 8

Changing the C value as well as the immediate reward and number of times the parent was played all produce different results, but the value for C appears to not matter as much as total amount of games played. It was very clear that the largest impact on the UCB equation was the number of plays in both the current Node and the parent.

With this information, I modified the playout to have a move limit of 30 rather than 50 as this will be much faster and allow me to get more moves. This was a great improvement as it gave me several more playouts in a 1 second game, from typically 1 playout per branch on a new game to 6. With more playouts, the algorithm should be able to find a better move judged on how many times it was played AND won.

In the original MCTS formula, the branch to be selected is the one with the most playouts, however I would argue after my simulations that this is only true for a large number of playouts. The checkers game has a relatively small number of playouts so it is safe to check, the win percentage of each branch. This shouldn't be done where there are generally a lot of playouts, because something that was played 2 times and won both times would outweigh something played 5000 times and won 4999 times. The number of plays was instead considered a secondary factor. If two branches have the same win percentage, the one who had more playouts was chosen instead. I expect a marginal improvement from this compare to the first pass.

With all the first improvements implemented, the player was tested against the 4 other players (random, d5, diff, and rat) 20 times: 10 times as player 1 and 10 times as player 2. The results from the simulations are shown in Table 1, and the win percentage is shown in Table 2.

Table 1: Results from the improvements listed above.

Player/Results	Random	D5	Diff	Rat
Win (Player 1)	10	9	3	0
Loss (Player 1)	0	0	0	0
Tie (Player 1)	0	1	7	10
Win (Player 2)	10	0	0	0
Loss (Player 2)	0	0	3	0
Tie (Player 2)	0	10	7	10

Table 2: Win % of the player against each player as player 1 or player 2

	Random	D5	Diff	Rat
Win %	100	45	15	0

Tables 1 and 2 show that the player is doing a decent game of checkers, but it is concerning that the win percentage for Diff and Rat are low and it does not always win against the d5 player. I would like to improve these and I think the best way to do that is with the Utility function.

Utility

The utility function is an integral piece of the program and should be optimized as much as possible in order to win. With the basic UCB formula the variable here is the value of C, the exploration constant, which was found to have a minimal impact on the game. Instead of trying to fine tune the value of C, I instead elected to tune the entire utility function. UCB is a great start, but there are many other utility functions available, one of which is a modification of UCB called UCB-tuned. UCB-tuned is a variation of the UCB function where C is replaced with a more complicated formula as shown in Eq. 2.

$$C = \sqrt{\frac{\ln(t)}{n} * \min(0.25, V(n))} \qquad \text{where} \qquad V(n) = \frac{1}{n}X_n^2 - (\frac{X_n}{n})^2 + \sqrt{\frac{2\ln(t)}{n}}$$

Equation 2: UCB-Tuned Equations

This is a much more complicated equation which makes C rely on the average immediate reward so far along with the numbers of plays of the current node and parent. Plugging this in for the utility function and rerunning the simulations provides the results shown in Table 3. In addition to the 4 provided players, I also ran it this time against my old alpha beta pruning player with iterative deepening. Unfortunately there was an unknown error when trying to run the MCTS player as player 2, so this just ran 10 times as player 1.

Table 3: Results from the new UCB formula

Player/Results	Random	D5	Diff	Rat	AB
Win (Player 1)	10	7	6	0	0
Loss (Player 1)	0	0	0	0	1
Tie (Player 1)	0	3	4	10	9
Win (Player 2)	10	3	0	0	X
Loss (Player 2)	0	0	0	0	X
Tie (Player 2)	0	7	10	10	X

Table 4: Win % of the player against each player as player 1 or player 2

	Random	D5	Diff	Rat	AB
Win %	100	50	30	0	0

The results largely show that the UCB-tuned is better than the original UCB formula, even if not by a substantial amount. The D5 player has interesting results as it has less wins when player 1 but more wins as player 2, resulting in an overall higher win percentage of 5%. The other attractive result is that it performs significantly better against the diff player when player 1. Against my old alpha beta pruning player, it is concerning that it actually appears to play worse. Maybe it was just bad luck, or maybe it needs improvements.

Final Improvements

The final improvement to be implemented is doing as much as I can to play more games. This can be done by reducing the depth of the alpha beta pruning algorithm, reducing the number of moves in the playout, or by using faster methods of evaluating. I don’t want to reduce the depth of the alpha beta pruning for all times as 3 is already a pretty low depth, so any lower might give me bad results. To counteract this issue, it will run at a depth of 4 if the number of legal moves is greater than or equal to 4, and run at a depth of 2 otherwise. The reasoning behind this is if there are a lot of legal moves, there are more possibilities of a better move than if only a few moves exist. If only a few moves exist, it likely won’t make much of a difference if I search at a depth of 2 versus 4 or larger.

Another likely improvement would be to get rid of all the recursive calls. After reworking the playout function to not use recursion, I was actually performing better likely because recursive calls are slower than iterative ones. I tried doing this but didn’t have quite enough time to get it figured out, but this could be implemented in the next checkers project that may improve it. Using the improvements mentioned before of changing the depth based on the number of legal moves, the results are shown in Table 5. I also ran the player against my old alpha beta pruning player with iterative deepening. There was an error why trying to run the new player second, so it only ran first.

Table 5: Results from the final improvements

Player/Results	Random	D5	Diff	Rat	AB
Win (Player 1)	10	1	6	0	3
Loss (Player 1)	0	0	0	0	2
Tie (Player 1)	0	9	4	10	5
Win (Player 2)	10	2	0	0	X
Loss (Player 2)	0	0	1	1	X
Tie (Player 2)	0	8	9	9	X

Table 6: Win % of the player against each player as player 1 or player 2

	Random	D5	Diff	Rat	AB
Win %	100	15	30	0	50

The results from this are very interesting. It still beats random 100% of the time which is good, does much worse on D5, slightly better on diff, slightly worse on rat, and performs mostly better against my old AB pruning player. It did lose a couple times, but also secured some wins in there which it previously didn't do. This is interesting because although in general the player did worse after my "improvements" I suspect after fine tuning this it can become better. I think the idea behind dynamically changing the depth based on the number of available moves is sound though and should be explored in the next project (if I still use MCTS) to get an even better player. For now, the improvements were reverted to go back to a safe and secure player that I know performs generally well.

Conclusion

In this project, I have modified my checkers player to use a Monte Carlo Tree Search algorithm in conjunction with my alpha beta pruning algorithm from the last project. This will simulate several games played and generate a utility for a move and continue playing the move with the best utility to overall maximize the return of the Player. It has been shown that this player can:

- Beat a random player 100% of the time.
- Generally beat a depth limited 5 alpha beta pruning player.
- Generally beat a difference heuristic iterative deepening alpha beta pruning.
- Perform as well as a ratio heuristic alpha beta pruning player.
- Perform roughly as well as my previous iterative deepening alpha beta pruning player.
    - Able to beat it with more tuning, but only about as good with code submitted.