CSCI4511W: MCTS in Othello

herbs070

April 2019

Abstract

Othello (also referred to as Reversi) is a timeless game that can be tracked all the way back to around the 1880's. The game is played between two players who face against each other to capture pieces in aims to end with more of their color than the other. The strategy involves the player to plan carefully and play with the end of the game in mind. In this paper, the Monte Carlo Tree Search will be used to play the game Othello. With the opponent as an agent that plays randomly, different roll out policies will be evaluated with respect to the iteration limit set.

Introduction

The game Othello is played typically with an 8x8 square board by two players. These two players play against each other, one white and the other black, where the goal for each player is to end the game with more pieces flipped to their color than the other player. The initial state of the board begins with the center 4 slots occupied with white in the top left and bottom right and black in the top right and bottom left (shown in figure 1). Each player then takes turns placing a piece of their color to sandwich the other player's pieces flipping them to match the player's color.

There are multiple strategies to playing this game. A new player may think that playing the move that will flip the most pieces to their color would be the best approach. This, in fact can be a hurtful strategy. According to an article that observes the different strategies in Othello, [2], there are two different types of strategies. These include position and a mobile strategies where the mobility strategy aims to keep as many options open as possible and the positional aims to capture key places on the board.

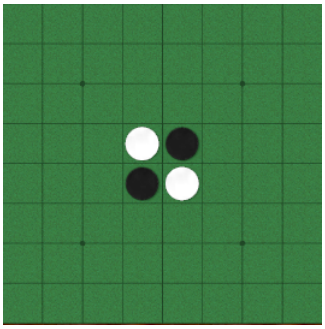

Figure 1: Starting Othello board

The mobile strategies focus on playing their pieces in a way that the player can adapt to whatever the opponent provides in terms of board states. This allows the player to respond to big plays made by the opponent. One strategy of this type makes a point of not playing good moves right away that cannot be taken away by the opponent. This gives the player the ability to save it for later and play a different play to supplement these good moves. Another example of one of these strategies is to play moves that will give the player the least amount of pieces. By doing this, the player leaves their pieces to be surrounded by the opponents giving them a large number of options and the ability to capture positions that are ideal while capturing pieces later in the game.

The positional strategies have a different approach. These types of strategies concentrate their moves to gain key positions on the board and combat dangerous ones. The corners and edges of the board are some of the most ideal places to control, this being said a player playing a positional strategy aims not only to capture these but to also not allow the opponent to gain access to these. One strategy of this type will avoid the positions adjacent to these as this could allow the opponent to gain control of it. The existence of these two strategies provide an interesting roll in the making of an agent that will combat an opponent.

MCTS

The Monte Carlo Search Tree algorithm provides a way for this game to be solved in a reasonable way. The algorithm takes the current state of the board (the representation array of pieces) and runs multiple simulations on it to find the next best move based on these simulations. A tree data

structure is used to keep track of the of the search with nodes that hold a state of the board as well as the Upper Confidence Bound (UCB) used in the search. This UCB is calculated after each simulation and is derived from the equation in figure 2 where x_i is the average win rate when this node is taken, n is the number of times that the simulation has been run, and n_i is the number of times that the node has been visited.

$$\bar{x}_i \pm \sqrt{\frac{2 \ln n}{n_i}}$$

Figure 2: UCB Equation

When the search begins, a node for the state is created and for each possible move that the agent can take next becomes a leaf node from this initial node. The beginning UCB for these nodes is infinity. The beginning tree for this appears like the tree in figure 3.

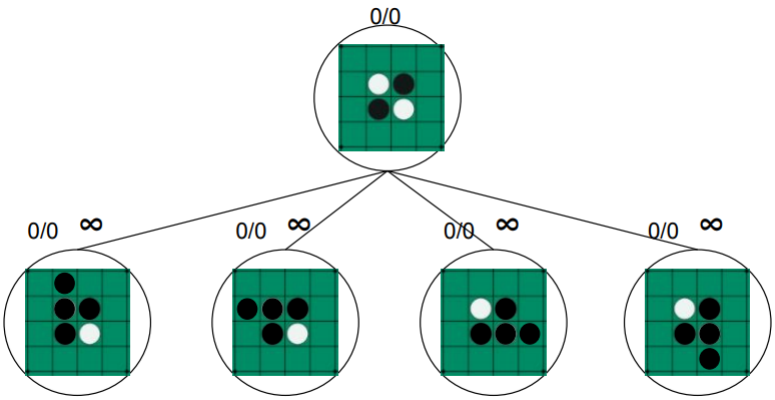

Figure 3: Example beginning MCTS tree

The algorithm then picks the node with the highest UCB value and simulates a game using a roll out policy. The equation that calculates the UCB does so in a way that both the win rate of a node and the number of times that it has been considered are balanced. This allows the search for the next possible action to not get stuck in a local maximum and miss the true next best move. One example of what an expanded search tree looks like can be seen in figure 4. We can see that the mid section of the tree is

where a large section of the search analyzed but the rightmost branches were expanded further with a greater depth suggesting that the best solution is down this path.

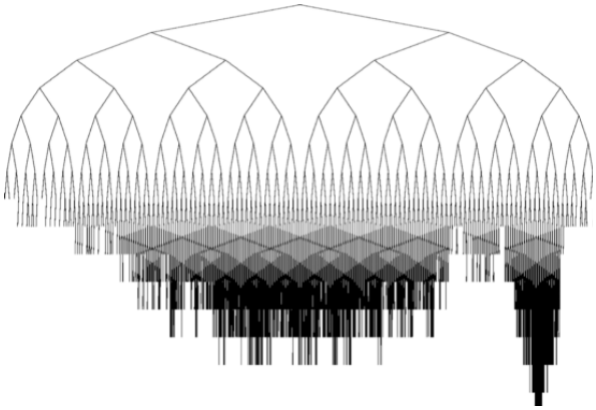

Figure 4: Expanded MCTS tree

The roll out policy is how the algorithm picks what moves to simulate such as random, most pieces, heuristically, etc. Once it is finished simulating the game it calculates to see if it has won or lost and updates the UCB value of the node using the equation above. This updates the UCB of the leaf node as well as the parent nodes. One example of an updated search tree might look like the tree in figure 4.

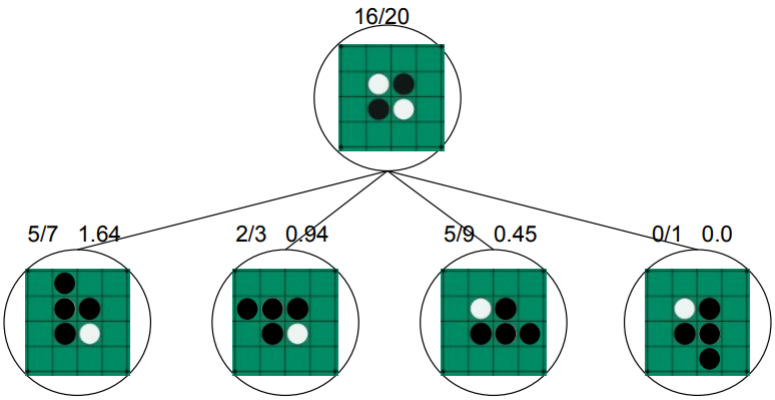

Figure 5: Example filled MCTS tree

Once the algorithm reaches a limit of games simulated or time spent simulating, the algorithm looks at the tree for the node with the greatest UCB and returns it to the game to be taken before the process begins all over again until the game is finished.

Related Works

The game console Atari has been around for many many years, In the article [5] from University of Michigan State, one of these simple games are used to test a Reinforcement Learning agent. This uses the MCTS to simulate the next best move for the agent but does so offline when the agent is not playing. This allows the MCTS to find a better solution as it has more time to simulate more games. So in games such as Pong, the agent can just look up the next best move rather than having to generate it.

Since the MCTS was introduced in 2006, it has been studied and applied to various games. One of the most popular games that this is applied to is the game Go. In the article, [4], different approaches to implementing an agent that utilizes this method are described. According to this article, carefully selecting the used roll out can improve the performance of the search and thus giving a better accurate next move. The article then proceeds to layout that a poorly crafted rollout policy can be hindering and even misguiding to the algorithm. This is because of the fact that the simulations would provide actions that the opponent would not logically make in the real game and when the agent's chosen action is employed, it is not working to beat what the real opponent has played. Choosing from a random selection from a subset of carefully selected moves can help the simulation to expand nodes that are worthwhile and avoid nodes that do not provide significantly promising results.

Another game that provides a fair application is the game Hex. Hex is similar to Othello in the way that players need to have a good long term strategy in order to gain an advantage over their opponent. The article [1], visits this application of MCTS and provides an interesting variation to help the algorithm. The roll out policy recognizes patterns on the board and is then able to explore certain promising moves more than others or prune the situations that appear not to be fruitful to the search. This allows the algorithm to avoid wasting time or being tricked into expanding to a path that is not best for the agent.

The nature of MCTS allows it to do very well at solving games with high branching factors. Modern board games are the perfect example of this.

Settlers of Catan is among these games and the primary focus of G.J.B. Roelofs' [6]. The UCB can be tweaked in order to change the balance of the exploration and the exploitation of nodes. This article deals with testing different ways of choosing to consider the exploration more heavily or the exploitation more heavily. This change can prompt more accurate results generated from the roll out policy and further impacts the next move taken.

There are many other expansions that can help to manipulate the MCTS to adapt to a certain problem. Usually with each expansion there is a cost and most of the time it is at the expense of speed. For example, in the article *A Survey of Monte Carlo Tree Search Methods*, a complex roll out procedure can bog down the time is takes to complete a simulation. This article also includes different ways that the MCTS is extended such as doing many simulations at the same time using a shared memory to increase speed, making the UCB more accurate so that paths that are not rewarding are expanded less, and nested MCTS.

Approach

To implement the test code base, John Fish's Othello game [3] was edited due to its simplistic approach yet with a clean and malleable interface. The Python library created by Paul Sinclair [7] was used to run the simulations. This provides the entire game to be passed in as a state as well as the roll out policy to be passed as a parameter. This Othello game provided the logic to make moves as well as manage which player's turn it was.

0.1 Roll Out

I implemented four different rollout policies for the simulation of the games. These are: the agent takes the first available move, the agent takes a random move, the agent takes a move based on maximizing the number of pieces taken, and the agent takes a move with consideration of the edges and corners.

This is a two agent game so the rollout policy must simulate both of the players meaning that each of the players must take on one of these policies. Every permutation of these in order to find which one of them is able to beat the random opponent.

0.2 Iteration limit

For the sake of consistency, the simulations will have an iteration limit of 75. This means that the search can simulate 75 games before expanding to the next node. This will keep the search consistent across the board and allow each of the policies to have an equal base to provide statistics.

0.3 Trials

Each of the 16 different roll out policies will play 50 games against the random opponent. This will allow the chance of a bad game not to hinder the results in an overly impactful way. After each run, the score will be recorded and whether the MCTS agent wins the game or not. This will give insight as to how well each of the policies preform.

Data

MCTS

Win Rate	Random	First	Dumb	Smart
Random	23	29	28	35
First	22	30	24	29
Dumb	24	24	40	27
Smart	28	28	22	25

Opponent

For the sake of simplicity, the labels on the chart and table are as follows:

- The Random move corresponds to when the agent plays a random move during simulation.
- The First strategy relates to when the agent plays the first move that it finds when searching from the top left corner.
- The Dumb move is when the agent takes the move that will give the player the most captures of other player's pieces.
- The Smart strategy is when the agent takes the move that gives them the best positional advantage given that the corners and edges are weighted.

Above are the number of wins by the MCTS agent against the random opponent. For each of the ways that a move can be chosen by the opponent

and MCTS agent there were 50 games simulated from the created roll out. This is graphically represented in figure 6 as a 3D histogram in order to obtain a better visual of what patterns are created.

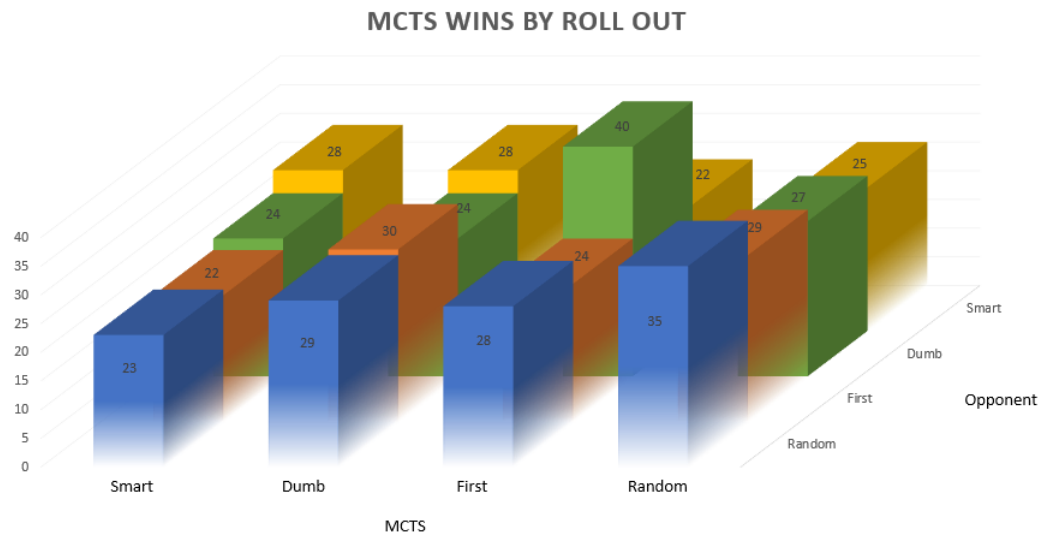

Figure 6: Wins by roll out

Analysis

With this data in mind, we see that there seems to be local maximums in both the corners of when both players play a random move and when both play a smart move. There is also an absolute maximum when the MCTS agent plays the first move and when the opponent plays a dumb move. The local minimums appear the most significant when one of the agents play the Random move and the other plays the smart move.

One of the reasons that the local maximums are created in the Random corner is mostly due to the fact that the actual opponent playing the game is playing a random move. By having the opponent play the Random move in the simulation, it is better represented in the roll out allowing the MCTS agent to try various moves to counter these without being drawn into the greed of taking the most pieces or edges based on a skewed representation of the opponent.

In the Smart corner, something interesting happens. The MCTS agent can focus on the best move that the opponent is able to take and play a

move that will then give them the best response. This impacts the MCTS agent by making sure that the real opponent's random move is not the best it can be in the future. This preventative maintenance gives the agent an edge over the opponent.

As for the local minimums, the corner where the MCTS agent plays a random move and the opponent plays a Smart move during the simulations does the opposite of the Smart corner. During the simulations, the MCTS agent assumes that the worst has happened and tries to make the best of it any way that it can. By doing this, it actually misses out on the big plays that it could have in the real game in terms of position and mobility.

When the MCTS agent plays a Smart strategy and the opponent makes a random move, we would expect to see the highest number of wins. We would assume that the opponent is represented as in the real game and the MCTS agent makes the best positional move it would reflect in the real game. This we can see is not the actual case. Reason being that the MCTS agent cannot find the actual best move it can make when the opponent makes a random move over and over. This inconsistent base matched with the large branching factor at the beginning of the game cause the MCTS agent's play to be built on sand and while it wins in the simulations, it may not be prepared for the actual game.

The absolute max can be explained a few different ways. When the MCTS agent plays the First approach and the opponent plays the Dumb policy, the agent is instructed to take the left edge of the board when it is able to. This combined with the opponent being greedy causes the real game to give the positional advantage to the MCTS agent by granting it access to the top leftmost corner and the ability to combat the random opponent.

The best rollout policy results in this absolute max as it aims the MCTS agent to gain a positional advantage by capturing a corner and edge by leading the opponent there and then taking it. This positional advantage early on lays out the ability to have control over what pieces it is able to take later in the game when crucial moves can be played from here.

Moving forward

There are several other ways that this agent can be improved to give it even more of an edge over its opponent. One of these ways would be to create an even more analytic roll out policy where the heuristic based on edge and corner pieces seen in the Smart policy weighs these positions even heavier. This would encourage an early positional advantage and the game would

play out in the agent's favor.

Another policy that can be analyzed is instead of the agent playing the same policy thought the simulations, it would be adaptive. During the beginning few moves it could look to make a move that is in the middle of the board as in the mobility strategy. After these initial moves it could use the Smart policy to gain access to the positional advantage so late game it could capture vital pieces. In the end game it could play the Dumb move to maximize the number of pieces that it has to compound with the positional strategy to finish off the opponent.

Patterns in Othello would be useful for the agent to recognize and improve results. It could recognize patterns of subsections of the board where moves are safe. One example of this is if the opponent has two edge pieces with a gap between them then it would be helpful to take that move. This could also help with pruning off branches that will not be worthwhile giving the agent more accurate results that are more accurate to the game.

Conclusion

In this article, different roll out policies have been tested such that the simulation can generate simulations in various ways. Local maximums and minimums have been explored through testing and their results are shown. The reason behind the results has been analyzed in aims to understand and intemperate what strategies work best.

After looking at related works in this field, there are many ways that this search can be improved and the direction that it can be expanded is suggested by the data. The best rollout shows that the most accurate representation of the game in the simulations is not always the best way. This is shown in the data collected that the rollout that won the most games won 10% more games then the Random representation.

References

[1] B. Arneson, R. B. Hayward, and P. Henderson. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, Dec 2010.

[2] Dorrit; David Shaman Billman. Strategy knowledge and strategy change in skilled performance: A study of the game othello.

[3] John Fish. Othello. https://github.com/johnafish/othello, 2015.

[4] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Commun. ACM.*

[5] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3338–3346. Curran Associates, Inc., 2014.

[6] G.J.B. Roelofs. Monte carlo tree search in a modern board game framework.

[7] Paul Sinclair. Monte carlo tree search library. https://pypi.org/project/mcts/, 2019.