Monte Carlo Tree Search and Its Applications

Max Magnuson
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
magnu401@morris.umn.edu

ABSTRACT

Keywords

Monte Carlo Tree Search, Heuristics, Upper Confidence Bounds, Artificial Intelligence

1. INTRODUCTION

In 1997 the field of artificial intelligence(AI) experienced a monumental breakthrough when IBM's Deep Blue defeated Garry Kasparov, a reigning grand master, in a chess match[**?**]. They were able to achieve this by using brute force deterministic tree searching methods combined with human knowledge. The human knowledge allows for the computer to evaluate moves properly, and then populate a tree to search for the best move. This event really demonstrated to the world the power of computers and artificial intelligence.

While computers are capable of outplaying the top players of chess, they struggle when it comes to board games like Go[**?**]. Go is a board game that originated in China, and it has a strong professional community. It is a game about positional board advantage which is something traditional AI approaches struggle with evaluating. This is because moves in Go tend to have very long dependencies. A single move may have major effects on moves 50 to 100 moves down the line[**?**]. Also Go has significantly more moves available to the player at anyone time than chess. These problems cause deterministic approaches to perform poorly. It is just too much for those approaches to efficiently handle.

People have started turning to alternative methods to approach Go. One such method, Monte Carlo tree search(MCTS) has had a lot of success in Go. MCTS eschews the typical brute force tree searching methods, and it utilizes statistical processes and heuristic approaches to decide what move to make. In 2009, for the first time ever, a computer defeated a top professional Go player in a 9x9 game[**?**]. It took twelve years since Deep Blue defeated Garry Kasparov for AI to achieve its first major victory in Go, and it was only on the smallest board that Go is played on.

MCTS has been growing in popularity in recent years, and it demonstrates a lot of promise. In this paper we will be examining MCTS and a few of its applications.

2. BACKGROUND

MCTS combines the random sampling of traditional Monte Carlo methods with tree searching. The random sampling is used to construct a game tree. This tree will be traversed based on statistical processes, and the MCTS method relies on the convergence of the tree to reliably choose the best move. MCTS is a heuristic method and as such it will not always find the most optimal move, but it has a reasonably high success of choosing moves that will lead to greater chances of winning.

2.1 The Tree Structure

MCTS structures the game state and its potential moves in a tree. Each node in the tree represents the state of the game with the root node representing the current state. Each line represents a legal move that can be made from one game state to another. In other words, it represents the transformation from the parent node to the child node. Any node may have as many children as there are legal moves. For example, at the start of a game of Tic-Tac-Toe the root node may have up to nine children. One for each possible move. Each following child can only have one less child than its parent since the previous moves are no longer available as options.

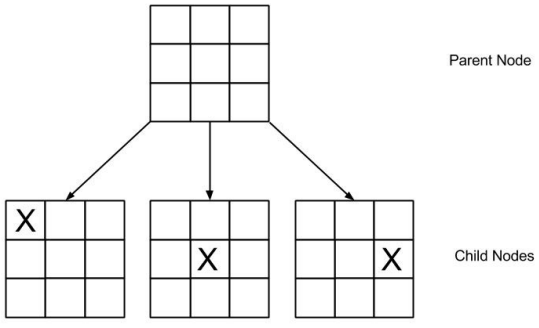

Figure 1: A small portion of what a tree represents

Figure 1 represents the top portion of a tree for the game Tic-Tac-Toe. The AI is making the first move, so the source

UMM CSci Senior Seminar Conference, May 2015 Morris, MN.

node is the first game board. Each child node represents the potential moves that can be made from the current game state. It is important to note here that those are only three of the potential nine child nodes.Todo: This somewhat contradicts what I say later on in the paper about UCT MCTS does not have to look at every potential child node, nor will it want to for sake of efficiency. Once MCTS has decided which move to make, the source node of the tree will then become the child it chose. For example, if MCTS chose the left child in figure 1, then the new MCTS tree would start at that child and everything else that was branching off of the original parent node is discarded.

Along with the game state, each node encodes for a value that represents how favorable choosing a particular node is. This value comes from the respective values of the nodes that branch off of it. In the example of Tic-Tac-Toe we could assign any simulation at a node that ends in a loss a zero, and any simulation that ends in a win a one. When any of these values are discovered, then the rest of the tree that branches into that node can be updated with its value. So in this case, choosing the node with the greatest value, leads to a path with the greatest ratio of wins to losses. By doing this, it gives the AI the greatest chance of choosing a winning outcome. This is what the MCTS algorithm relies on to be effective.

2.2 The Four Steps of MCTS

The process of MCTS is split up into four processes: Selection, Expansion, Simulation, and Backpropagation. These four processes are iteratively applied until a decision from the AI must be made.

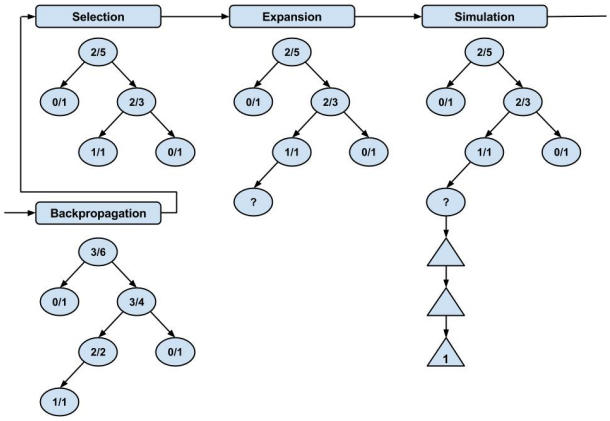

Figure 2: The four steps of MCTS

Todo: This section still needs a lot of work

Selection - In the selection process, the MCTS algorithm traverses the current tree using a tree policy. A tree policy uses an evaluation function that prioritize nodes with the highest estimated value. In 2 the MCTS algorithm traverses to the 2/3 node then the 1/1 because those are the nodes with the greatest estimated value.

Expansion - In expansion a new node is added to the tree as a child of the node reached in the previous step. There is only one node added to the tree in each iteration, and it is at this step. In 2 the newly added node is indicated by the ?.

Simulation - In this step, a simulation(also referred to as a playout or rollout) is played out according to the simulation policy[?]. The simulation may use either a weak or strong policy. A weak policy would use little to no predetermined strategy. It would simply playout the simulation randomly. A strong policy would use a more guided approach to choose the moves. A strong policy may make the simulation too deterministic or make it more prone for error[?]. The policy plays out moves until either an end state or a predefined threshold is reached. Then based on the result of the simulation, the value of the newly added node is established. For example, a simulation of a node for Go would reach the end of a game(the end state), and then determine a value based on whether the player won, or lost. In 2 the simulation ended in a 1. Therefore, the value of the new node is 1/1.

Backpropagation - Once the value of the node is determined, then as the name of this process implies, the rest of the tree can be updated. The algorithm will traverse back to the root node only updating the values of the nodes that it passes through. Only those nodes are effected because each node's respective value is an estimation of values of the nodes after them. In 2 there are two nodes that are left untouched since those do not branch into the newly added node.

2.3 Upper Confidence Bound Applied to Trees(UCT)

Todo: Get across that each unexplored node is explored before moving on, but only at each node that is visited. If a node is not visited again, that node's children will not be explored. The UCT is what the MCTS algorithm uses as a tree policy to traverse the tree. The goal of the UCT is to balance the idea of exploration versus exploitation. The concept of exploration promotes exploring many unexplored areas. This approach may explore many different paths on its way to finding the best decision. While this approach is useful to ensure that MCTS is not overlooking any potential paths, it can become very inefficient very quickly with games with a large number of moves. This is balanced with exploitation. The exploitation approach will tend to stick to one path that has the greatest estimated value. UCT balances these two ideas by exploring every unexplored node that it visits, but then it only visits nodes that yield the greatest estimated value.

$$UCT(node) = \frac{W(node)}{N(node)} + \sqrt[c]{\frac{ln(N(parentNode))}{N(node)}} \quad (1)$$

When traversing the tree, Equation 1 is applied to each of a node's children to evaluate the estimated value of that node[?]. N() represents the total number of simulations made at that node and the nodes branching off of it. W() represents how many of those simulations ended in a winning state. C represents an exploration constant that is found experimentally. The first part of the UCT takes into consideration the known estimated value of the node by determining the ratio of simulations won to total simulations. The second part of the UCT takes into consideration the unexplored nodes of the parent node by taking the ratio of the parent node's total simulations to the node's simulations.

3. USING MCTS TO PLAY GO

Todo: I could talk about a variation in their playout policy, but it would require a lot of explanation of Go, and I think it wouldn't add nearly enough to the paper to warrant the added explanation of Go. MCTS has seen a lot of success in its applications in Go. The computer Go programs MoGo and Crazy Stone both use MCTS, and they have had the best performance of any computer Go programs[?]. Those programs use the traditional MCTS algorithm with the UCT approach, and they apply their own variations specifically in the context of Go.

3.1 All Moves as First(AMAF)

The all moves as first(AMAF) approach treats all moves as the first move. That means that AMAF does not care about any move in relation to any other move. Moves have no context in which they are played. The value of a move is determined by the end values of the subtrees in which that move was played. For example, move a is made five times within a given subtree. Move a being played first from the current node resulted in two losses, but move a was played in three other parts of the subtree that ended in a win. Therefore, the overall value of a would be 3/5, and this value would be treated as if a were the next move played. So if a's value of 3/5 is greater than any other move, a would be selected as the next move action.

The AMAF approach works in Go because many of the situations only affect what is happening locally. If a move is made elsewhere on the board, it does not really have any affect on the move being examined.

3.2 Rapid Action Value Estimate

Rapid action value estimate(RAVE) takes the concept of AMAF and stores the value of every move at every node in the tree. This means that each node contains the combined value of each move that has been played out in a simulation branching from that node. This is how the information must be stored for the MCTS algorithm to work because MCTS works on the assumption that each value that the node contains comes from the simulations performed at that node and the ones in the subtree of that node. This assumption must remain true since MCTS traverses the tree selecting for nodes with the greatest potential value.

The RAVE approach is very powerful and allows us to retrieve much more information out of every simulation that MCTS performs. Typically in MCTS only one piece of information would be gained by a simulation. That would be whether or not that specific node resulted in a win or a loss. Now, every move in that simulation provides the MCTS with information on the value of that move. This allows the MCTS algorithm to converge much more quickly on which move to perform next.

3.3 MC RAVE

The RAVE approach is very useful and efficient, but it can sometimes select an incorrect move[?]. When the players have close tactical battles, the sequencing of the moves become very important. In this situation, we cannot treat the moves as AMAF. We still need the contextual dependencies of the MCTS approach.

MC RAVE combines the traditional MCTS algorithm and the RAVE approach into one algorithm. MC RAVE uses a weighted approach that combines the values from the MCTS value and the move values from RAVE. The weighted approach works as a sliding scale that gives more value to the MCTS value the more simulations that started from that node[?]. Inversely, if only a few simulations have been played out, the RAVE value would have more weight.

The MC RAVE approach works because the strength of the RAVE approach is that it provides a lot of information and converges quickly with relatively few simulations. The MCTS approach is given more weight as more simulations are performed because it gives a much more accurate estimation of the contextual dependencies, but it requires more simulations than RAVE.

3.4 Their Results

Computer programs that used more traditional approaches to playing game have had very little success playing Go. The deterministic approaches struggled to defeat even low rank amateurs. Now with new Go programs implementing MCTS, they have achieved a lot of success and the achievements are only growing. The top computer programs can now compete with top professionals in 9x9 Go[?]. Not only that, but those programs can even compete against the top pros in handicap games of 19x19 games of Go. That is an incredible feat taking into consideration the immense complexity of a 19x19 board. Clearly, MCTS has demonstrated its impact on AI approaches to Go.

4. USING MCTS FOR NARRATIVE GENERATION

MCTS has demonstrated that it has applications outside of playing board games. Kartal et al[?] used MCTS for narrative generation. Their algorithm uses a list of actors, items, and places and various actions that let those things interact with each other. The user then specifies the initial setup and overall goal for the story.

4.1 Tree Representation

As stated previously, the nodes of the MCTS tree represents the state of the system at that node. In the narrative generation, the nodes will hold the information of what action is happening, and the current attributes of the actors. Any previous nodes will hold the information about the story that happened up to the current node's point in the story with the root node being the very first action taken in the story.

The attributes of the actors are simply information that is needed to help describe the story. An actor may have an attribute of their current health, their current location, or the current anger level of the actor. Certain actions become more likely or less likely to occur depending on these attributes.

- **Move(A, P):** A moves to place P.
- **Kill(A, B):** B's health to zero(dead).
- **Earthquake(P):** An earthquake strikes at place P. This causes people at P to die (health=0), items to be stuck, and place P to collapse.

Here is a small sample of possible actions that may occur in a story. It would not make much sense if actor A killed actor B if actor B did not interact with actor A. Although if actor A's anger is sufficiently high enough, it may make it more believable for this action to occur[?].

4.2 Narrative Simulation

In Go, the simulation step of MCTS would end in either a win or a loss, but that approach does not really apply to narrative generation. Instead the authors chose to establish a threshold for their simulation policy. When the story in the simulation reaches a certain length, or if the story accomplishes a predefined amount of the goals outlined by the user, the simulation with then stop.

To establish a value for the simulation, the authors developed a believability function. The believability function evaluates the simulation based on how believable the narrative is, and how many goals the narrative accomplished. The believability is based on the order in which the actions occur and how likely they are to occur given prior events.

The believability functions strikes a balance between goal completion and believability because it would not make for a good narrative without either. A narrative could easily complete the goals laid out without being very believable, and a narrative could be very believable while not accomplishing any of the goals. Sometimes this results in one being sacrificed for the other. For example, maybe a long series of actions that are not the most believable are used for the sake of completing the goals of the narrative. While this outcome is not perfect, it is preferable over the two extremes.

In the simulation process the authors decided to use a guided approach. Their algorithm has a table that keeps track of various actions that have occurred in the simulations and their respective evaluation score. They use these scores to bias the random sampling in the simulations towards actions with higher scores.

4.3 Tree Pruning

The trees for MCTS can get very memory intensive, so the authors decided to implement tree pruning into their algorithm to essentially allow for the algorithm to use the memory more efficiently. In tree pruning, an algorithm will selectively cut out pieces of the tree that do not seem very promising.

In the authors' implementation, they decided to plan out the story one step at a time. To accomplish this, they allow the tree to expand for a fixed number of iterations. Then, after those iterations, the next step is selected. All other steps are discarded, and then the process repeats. This process allows for the algorithm to much more efficiently use the memory it is allocated.

The authors do make note that this approach makes the algorithm no longer probabilistically complete. This means that it is possible that one of the branches that is pruned would be preferable to the current path. The current path may just be a local maxima instead of the preferred global optimum. Even with this flaw, the authors still found their approach to perform reasonably well.

4.4 Their Results

The authors compared their results to three different tree search approaches: Breadth-First Search, Depth-First Search, and Best-First Search[**?**]. Each of these algorithms should provide the optimal solution if given enough time and memory. Depth-First Search and Best-First search in particular are capable of finding the solution very early on in the search.

They compared these four algorithms allowing them two different amounts of nodes. The first comparison allowed the algorithms up to 100 thousand nodes, and the second comparison allowed the algorithms up to 3 million nodes. They are then compared on the score each one achieved from the believability function.

In the first comparison, the only algorithm that came close to MCTS was the Breadth-First Search algorithm. Bread-First Search scored on average over three trials of .05, while the MCTS scored a .07. The other two algorithms scored below .01. When the algorithms were allocated 3 million nodes, MCTS far and away outperformed the other algorithms. MCTS scored .09 while the others didn't even get up to .01.

Todo: Talk about comparison of Authors' implementation of MCTS to traditional MCTS

5. USING MCTS TO PLAY MARIO

5.1 Variations in Their MCTS Algorithm

5.2 Their Results

6. CONCLUSIONS

7. ACKNOWLEDGEMENTS

8. REFERENCES

8.1 Citations