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[2]. The researchers were able to achieve this by using brute force deterministic tree searching methods combined with human knowledge of chess. The human knowledge allows the AI to evaluate the strategic value of a move much like a grand master would, and then populate a tree to search for the best move. This event 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[2]. Go is a board game about positional board advantage which is something traditional AI struggles with. They struggle because moves in Go tend to have very long dependencies. A single move may have major effects on moves 50 to 100 moves ahead[3]. This makes game trees for Go significantly deeper than chess. Go also has many more moves available to the player at any time than chess. This makes the game trees for Go much wider than in 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[2]. It took twelve years for AI to advance from defeating Garry Kasparov 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.

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

2. BACKGROUND

MCTS combines the random sampling of traditional Monte Carlo methods with tree searching. Monte Carlo methods use repeated random sampling to obtain results. 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 traversal to reliably choose the best move. The traversal is in convergence when the traversal selects the same path on each traversal. MCTS is a heuristic method and as such it will not always find the 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.

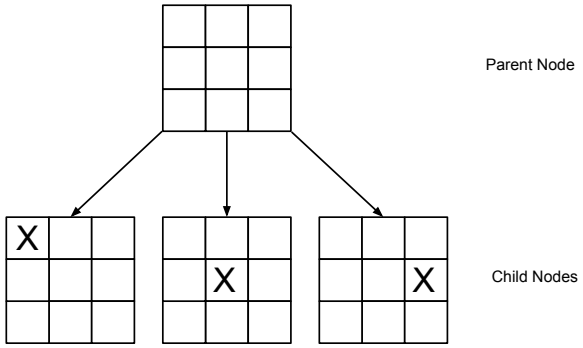

Figure 1: A small portion of what a tree for TicTac-Toe represents

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 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. In the naive implementation of MCTS, each potential child node must be added to the tree before any of those child nodes can be traversed. Other variations of the MCTS algorithm have ways of avoiding this requirement[1]. 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 root node would be that child and every other child node would be discarded.

Along with the game state, each node contains a value that gives an estimate of wins compared to total games in that subtree. The higher the value, the greater proportion of wins in that subtree. The lower the value, the smaller the proportion of wins in that subtree. By choosing the node with the greatest estimated value, the MCTS algorithm is choosing the path with the most number of wins. This means that the MCTS algorithm is maximizing the number of winning moves it can select. This is what MCTS relies on to be effective.

2.2 The Four Steps of MCTS

The process of MCTS is split up into four steps: Selection, Expansion, Simulation, and Backpropagation. These four steps are iteratively applied until a decision from the AI must be made. Typically, there is a set amount of time that the AI has to make its move, so that is when the algorithm will make its decision.

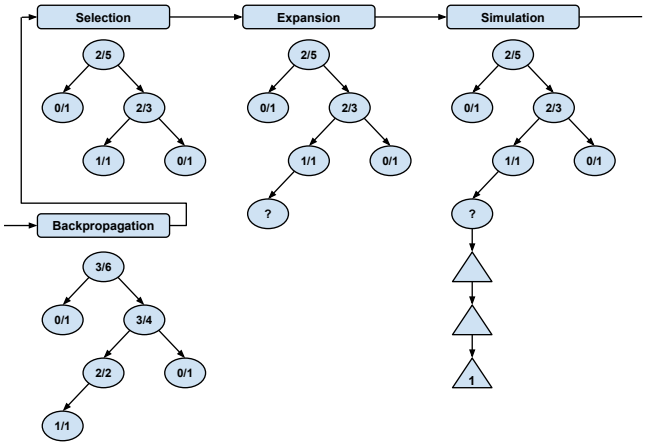

Figure 2: The four steps of MCTS

In figure 2 the first number in each node represents the number of wins in that subtree. The second number is the total number of simulations performed in that subtree which is also the same number of nodes in the subtree. The ratio of these two numbers provide us with the estimated value of each node.

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 greatest estimated value. In figure 2, starting from the root node, the tree policy must make a decision between the 0/1 node and the 2/3 node. Since 2/3 is greater than 0/1, the tree policy will choose the 2/3 node in its traversal. Once at the 2/3 node, the tree policy will then choose the 1/1 node because it is greater than 0/1. So now the algorithm is at the 1/1 node as it transitions into the expansion step.

Expansion - In the expansion step, a new node is added to the tree as a child of the node reached in the selection step. The algorithm is currently at the 1/1 node, so there is a child node added onto that node indicated by the node with the ?. There is only one node added to the tree in each iteration, and it is at this step.

Simulation - In this step, a simulation(also referred to as a playout or rollout) is performed by choosing moves until either an end state or a predefined threshold is reached. In the case of Go or TicTacToe, an end state is reached when the game ends. Then based on the result of the simulation, the value of the newly added node is established. For example, a simulation for a node in Go reaches the end of a game(the end state), and then determines a value based on whether the player won or lost. In figure 2 the simulation ended in a 1. Therefore, the value of the new node is 1/1. One simulation resulted in a win, and one simulation has been performed.

In the simulation process, moves are played out according to the simulation policy[1]. This policy may be either weak or strong. A weak policy uses little to no predetermined strategy. It chooses moves randomly from either a subset of the legal moves or from all of the legal moves. A policy may prefer a certain subsection of moves because those moves might be more favorable. Perhaps in the game of TicTac-Toe the corners are considered to be more favorable. We incorporate this into a simulation policy by having the algorithm randomly choose corner moves until there are no more corner moves left. Then the policy will choose moves at random from the rest of the legal moves. A strong policy uses a more guided approach to choosing moves. A strong policy may make the simulation too deterministic or make it more prone to error[2], so a weak policy is generally preferred.

Backpropagation - Now that the value of the newly added node has been determined, the rest of the tree must be updated. Starting at the new node, the algorithm traverses back to the root node. During the traversal the number of simulations stored in each node is incremented, and if the new node's simulation resulted in a win then the number of wins is also incremented. In figure 2 only the nodes with values 0/1 are not updated since they are not a parent of the newly added node. This step is very important because it ensures that the values of each node accurately reflect simulations performed in the subtrees that they represent.

2.3 Upper Confidence Bound

The upper confidence bound applied to trees(UCT) is used by MCTS as the tree policy in the selection step to traverse the tree. UCT balances the idea of exploration versus exploitation. The exploration approach promotes exploring unexplored areas of the tree. This means that exploration will expand the tree's breadth more than its depth. While this approach is useful to ensure that MCTS is not overlooking any potentially better paths, it can become very inefficient very quickly in games with a large number of moves. To help avoid that, it is balanced out with the exploitation approach. Exploitation will tend to stick to one path that has the greatest estimated value. This approach is greedy and

will extend the tree's depth more than its breadth. UCT balances exploration and exploitation by giving relatively unexplored nodes an exploration bonus.

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

When traversing the tree, the child node that returns the greatest value from equation 1 will be selected [1]. N represents the total number of simulations performed at that node and its descendants. W represents how many of those simulations resulted in a winning state. C represents an exploration constant that is found experimentally. The first part of the UCT takes into consideration the estimated value of the node from the ratio of simulations won to total simulations. This is the exploitation part of the equation. The second part of the UCT is the exploration bonus. This compares the total number of simulations performed at the parent node and its descendants to the total number of simulations performed at the examined node and its descendants. This means that the lower the number of simulations that have been performed at this node, the greater this part of the equation will be.

3. USING MCTS TO PLAY GO

MCTS has been very successful 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[3]. Those programs use the naive MCTS algorithm with the UCT approach, and they apply their own variations that take advantage of certain aspects of Go.

3.1 All Moves as First(AMAF)

All moves as first(AMAF) is a methodology that treats all moves as if they were the next move played. AMAF does not grant any move extra strategic value based on when it is played. Therefore, in AMAF moves have no contextual dependencies on other moves. This is particularly useful when a move played elsewhere on the board has little or no impact on the move being examined, or if a game arrives at the same state regardless of the order in which the moves are played.

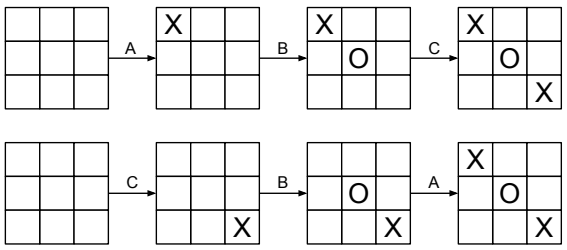

Figure 3: Comparison of two sequences of moves in TicTacToe

In figure 3 are two possible sequences of moves that can be played out in the game TicTacToe. Even though the order of moves A and C are different, it still results in the same game state. AMAF is useful in analyzing the effectiveness of this situation since the order in which the moves are played has no effect strategically. Thus, we can treat playing move A first or move C first as having the same strategic value.

The AMAF methodology is applicable to Go because many of the situations only affect what is happening locally. If a move is made elsewhere on the board, it does not have much of an affect on the strategic value of the move being examined. It is also important to note that in Go a move that repeats a board state is illegal. Therefore, this methodology will not have any inconsistencies with replaying the same move.

3.2 Rapid Action Value Estimate

Rapid action value estimate(RAVE) takes the concept of AMAF and applies it to a tree structure. RAVE can be thought of as assigning values to the edges of the tree which represent moves. The value of these moves come from any simulation performed within the subtree in which that move was played. The value is a ratio of these simulations that resulted in a win to the total number of simulations. This is different from MCTS in that MCTS chooses nodes for the strategic value of the game state represented by that node. RAVE chooses nodes for the strategic value of the move.

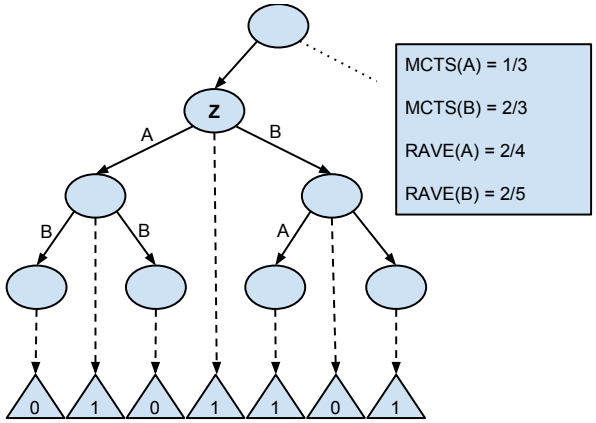

Figure 4: MCTS vs RAVE

Figure 4 is a comparison between moves A and B from node Z. The triangles represent the result of the simulations performed at each node. In MCTS, the value of A is the value of the node A points to. In this case, A has the value 1/3. Likewise, B has the value 2/3. These values come from the three simulations performed in their respective subtrees. In the RAVE approach, the value for move A is determined by any simulation performed by the descendants of node Z in which A was performed. This accounts for the simulation performed in the subtree of B that used move A. Now in RAVE, the value of A is 2/4. The same is true for the RAVE value of B. B was performed in two other simulations in Z's subtree. This makes the RAVE value of B 2/5.

When determining which node to traverse to from node Z, MCTS and RAVE would produce different results. MCTS would choose the node that B is pointing to because the MCTS value of B is greater than the MCTS value of A. RAVE would choose the node A is pointing to because the RAVE value of A is greater than the RAVE value of B.

The RAVE approach is very powerful and allows us to retrieve much more information out of every simulation. MCTS only gains one piece of information from each simu-

lation. That information is only the result of the simulation. In RAVE, every move performed in a simulation provides us with information. The strategic value of a move in RAVE is developed much more quickly as a result. This means that trees generated by RAVE converge more quickly than trees generated by MCTS.

3.3 MC RAVE

The RAVE approach is very useful and efficient, but it can sometimes select an incorrect move[3]. In Go, 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 naive MCTS algorithm and the RAVE approach into one algorithm. MC RAVE stores the values of each node from MCTS and the value of each move from RAVE in the tree structure. MC RAVE takes a weighted average of the two values to determine which node to choose in traversal [3]. When few simulations have been performed, the RAVE values are given more weight. In this case, RAVE is more accurate because the contextual dependencies of moves are less clear. When a lot of simulations have been performed, the MCTS values will be weighted more heavily. The MCTS values are given more weight because the contextual dependencies of the moves are more strongly developed and are more accurate overall.

Todo: I might want to step through the MCTS vs RAVE diagram traversal using MC RAVE here

3.4 Results

AI that use more traditional approaches to playing games have had very little success playing Go. The deterministic approaches struggle to defeat even low rank amateurs. Now with new Go programs implementing MCTS with RAVE, they have achieved a lot of success and the achievements are only growing in number. The top computer programs can now compete with top professionals in 9x9 Go[3]. Not only that, but those programs can even compete against the top pros in handicap games of 19x19 Go. Handicap games let one player start with some number of pieces on the board. 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

Automated narrative generation is an interesting problem with many applications. One such application is in video games. Narrative generation provides the user with a unique experience on each playthrough which can extend the amount of enjoyment a user can receive from a single game. Another application for narrative generation is as a learning tool. Generating an endless number of fun and interesting stories provides plenty of material for reading practice. These are only two examples, but there are many more.

Kartal et al[4] decided to apply MCTS to narrative generation. MCTS is a promising approach to narrative generation because as with Go it shares the same problem of expansive search spaces. With each additional character, item, location, or action, the number of possible narratives grows exponentially. Given the success in Go, with its huge search spaces, it makes sense to apply MCTS to narrative generation.

The researchers' algorithm uses various predefined actors(or characters), items, locations, and actions to generate a narrative. Actions are what are used to have actors, items, and locations interact with one another. Here are a few possible actions:

- **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.

Each action has a name and in the parentheses are the actors, items, or places used by the action. The action move takes a character A and moves them to place P. Actions are important because they are what progress the story.

In the researchers algorithm, the user does not provide any of the actions used by the algorithm. However, the user defines the initial setup and goals for the narrative. An initial setup indicates where actors or items are located. The inspector is in his office, or the lamp is at Becky's house. The narrative goals are what the user wishes to occur in the story. Perhaps the user would like there to be at least two murders and for the murderer to be captured. Given this information, the algorithm will attempt to generate a believable narrative that satisfies the goals set by the user.

4.1 Tree Representation

As stated in section 2.1, the nodes of the MCTS tree represent the current state of the system. In narrative generation, it is a little different. Each node only represents a specific step of the story instead of capturing the entirety of the narrative up to that point. In order to retrieve the current narrative up to that node, we would need to traverse up the tree all the way to the root node. This structuring is similar to the MCTS tree for Go in that to know the sequence of moves leading up to a node, we would need to traverse back to the root. A node by itself does not provide us with the order in which moves are played, only the current state of the game.

In addition to encoding for an action, each node also keeps track of various attributes. Attributes are characteristics of an actor that help keep track of the current state of the story. An attribute of an actor might be that actors current health. The health of an actor may go down if attacked. Another example of an attribute is the location of an actor. It would not make much sense if an actor traveled to a location that they are already at which would be possible if the location of the character is not stored.

4.2 Believability Function

4.3 Narrative Simulation

Todo: I should make clear the two different heuristic methods they introduced, so it is easier to contrast them in the results 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 will then stop and be evaluated.

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 by the user 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 uses a table that keeps track of various actions that have occurred in the simulations and their respective average evaluation score. The algorithm uses these scores to bias the random sampling in the simulations in favor of actions that have produced higher evaluation scores in the past.

4.4 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 it is allocated more efficiently. In tree pruning, an algorithm will selectively cut out pieces of the tree that do not seem very promising.

The authors implemented tree pruning by only allowing the algorithm to plan out the narrative one step at a time. When the next step is being selected, the algorithm will run for a predefined number of iterations. After those iterations have been run, the algorithm will choose the node with the greatest potential value that is a child of the current root node. The chosen node effectively becomes the new root node of the tree while keeping track of the steps that preceded it. When the new node is chosen, all other children from the previous node are discarded. The memory that those nodes were allocated can now be used for future simulations. 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 algorithm to perform reasonably well.

4.5 Results

The authors compared their results to three different tree search approaches: Breadth-First Search, Depth-First Search, and Best-First Search[4]. 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.

The authors compared their playout biasing approach to a MCTS algorithm without it. They found that given a low budget of nodes, the typical MCTS algorithm outperformed their biased approach. Alternatively, they found that when the algorithms were allowed a large number of nodes, the biased approach far outperformed the typical MCTS approach. The authors also compared their tree pruning approach to a typical MCTS algorithm. They found that the typical MCTS algorithm would quickly run out of memory when given a large number of actors, items, and places. The tree pruning approach used memory much more efficiently and found much higher scoring stories than the typical MCTS algorithm.

5. CONCLUSIONS

6. ACKNOWLEDGEMENTS

7. REFERENCES

[1] D. Brand and S. Kroon. Sample evaluation for action selection in monte carlo tree search. In *Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014 on SAICSIT 2014 Empowered by Technology*, SAICSIT '14, pages 314:314–314:322, New York, NY, USA, 2014. ACM.

[2] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Commun. ACM*, 55(3):106–113, Mar. 2012.

[3] S. Gelly and D. Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856 – 1875, 2011.

[4] B. Kartal, J. Koenig, and S. J. Guy. User-driven narrative variation in large story domains using monte carlo tree search. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 69–76, Richland, SC, 2014. International Foundation for Autonomous Agents and Multiagent Systems.