A Brief Description of Monte Carlo Tree Search

Douglas C. Whitaker

9 December 2011

1 Description

Monte Carlo Tree Search (MCTS) methods are a relatively new (established in 2006) application of Monte Carlo methods and have seen much use in the field of AI for games (both classic board games and modern computer games). Essentially, this method is used as an alternative to a binary (or n-ary) tree search when the construction and storage of such a tree would be computationally prohibitive. The general algorithm [Chaslot et al., 2006, 2008] is as follows:

1. Identify the current game state that results from each possible move.
2. If any of these game states do not exist in our tree structure (i.e. we have not seen that game state before and thus do not have any information on it), simulate g_i games starting from that state played by random moves.
3. If any games were simulated, update the tree with the results of these g_i games (Backpropagation).
4. Apply a move selection function f to the values, v_i, assigned to each of the possible moves (or, rather, the game states associated with those moves).

In the above description, note that the primary use of the randomness is in randomly filling the tree so that standard MCTS is simply an alternative to starting with a completed tree for an n-ary search algorithm. Moreover, the choice of the move selection function f and the manner in which the v_i are computed can have a tremendous impact on the effectiveness of the algorithm and careful selection of these is necessary.

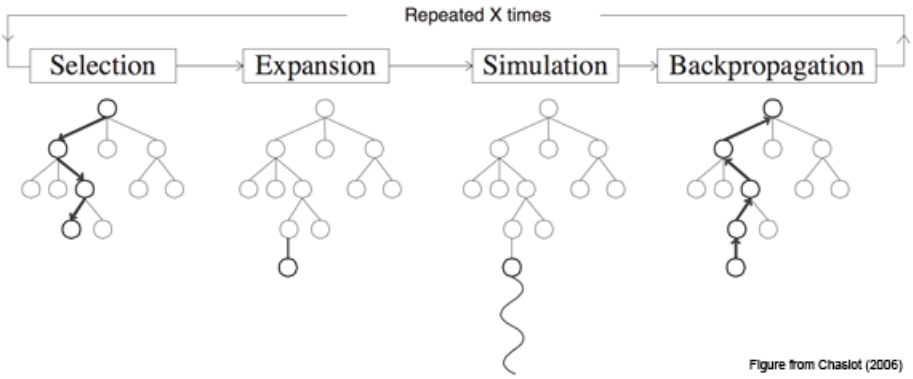

Figure 1: A visual representation of the algorithm [UCT for Games and Beyond, 2010].

1.1 Upper confidence bounds applied to trees

A method commonly employed with MCTS is *upper confidence bounds* (UCB), which essentially is the creation of a one-sided confidence interval for each value v_i for possible moves. These upper confidence bounds (rather than the raw v_i's) can then be analyzed using a selection function f. The general form is

$$\text{Upper Bound } = v_i + C\sqrt{\frac{\log_e N}{n_i}}, \tag{1}$$

where N is the number of times the current state has been visited, n_i is the number of times the potential game state has been visited, and C is a bias parameter which can be changed to allow for different behavior of the AI (and is often tuned through trial and error) [UCT for Games and Beyond, 2010, Winands et al., 2008]. When UCB is employed in conjunction with MCTS, the resulting algorithm is referred to as *upper confidence bounds applied to trees* (UCT).

2 Applications

Much of the research done on MCTS methods is in the field of AI, particularly AI as it relates to games (both classic board games and modern computer games). While effective AI methods have existed for many games for decades (e.g. computers have been able to defeat world-class chess players since 1989), many games have heretofore proved difficult for AI to play adroitly (e.g. a computer was only able to beat a professional level human at the board game Go in 2008). In recent years, numerous games have had AI constructed based on MCTS; among these games are: Go, chess, Scrabble, poker, Settlers of Catan, and Ms. Pac-Man [Chaslot et al., 2008, Szita et al., 2009, Samothrakis et al., 2011].

2.0.1 Go

The reason that Go proved more difficult to develop a skilled AI for as compared to a game like chess are many-fold. Chief among these reasons are

- Go is played on a larger board (19×19) than chess (8×8) resulting in many, many more possible moves,
- the position of individual pieces in Go can have a more subtle, long-term impact on the game than those of chess, and
- there are typically many more valid moves to select from in Go than chess.

3 Example Implementation in R

To help achieve further understanding of MCTS, I decided to implement the UCT algorithm as applied to a game of Connect Four; this game was chosen for the simplicity to implement, and, on a more personal level, I am terrible at it. All of the code written, along with a starter database of approximately 25,000 game states, will be made available at `http://www.stat.ufl.edu./~whitaker/mcts-connect/`. The application of MCTS has been done before, so this is not a novel selection. For computational convenience, the game actually implemented was Connect Four as played on a 4x4 board with a win condition of 3 in-a-row; I'll (unsurprisingly) call this derivative Connect Three. (Note: the code was written with the intention of being able to play Connect k on an $m \times n$ board, $k < \min(m, n)$, but this hasn't been tested yet.)

In all cases, I tested running the MCTS as the player to make a move second (Player 2), i.e. with the assumption that a human will be making the first move. Connect Four is a solved game, and the first player is at a substantial advantage. For this implementation, we assigned values to each game state by

$$v_i = \frac{\text{Number of explored game states resulting in a win for Player 2 that originate at move choice } i}{\text{Number of total game states explored that originate at move choice } i}. \tag{2}$$

Table 1: Win percentages based on 100 simulated games for the various other AI implementations as Player 2 vs. the naïve AI implementation. For MCTS, the value of the bias parameter and whether or not an initial database of game states was available to the AI are listed.

AI	Naïve	MCTS, $C = 2$, No	MCTS, $C = 2$, Yes	MCTS, $C = 0.2$, No	MCTS, $C = 0.2$, Yes
Win %	28%	28%	46%	72%	82%

In other words, v_i is the estimated win proportion based on knowledge of all previous played games that the AI has access to. The move selection function f, coded as `evaluateNextStates()`, was chosen to be the following:

1. Obtain the UCB values for each of the j possible moves.
2. If any $v_i > \text{UCB}_j, \forall i \neq j$, choose move i.
3. If any $\text{UCB}_i < v_j, \forall i \neq j$, prune move i (do not consider it for selection).
4. If a move has not been selected yet, randomly sample from the remaining moves, weighting each move by it's v_i value. When random games were simulated by the MCTS algorithm, $g_i = 10$ was used each time. Unless otherwise specified, the bias parameter used was $C = 0.2$.

It is important again to state that the selection of f and v_i will dramatically influence the performance of the algorithm; in this implementation, these definitions were chosen using intuition and not based on any sort of theory or optimality properties. Moreover, a deterministic component was added to the algorithm: the AI will check to see if any of the possible i moves will result in a win for itself and choose move i with probability 1 if it will result in a win.

For comparison purposes, a naïve AI was also implemented; this AI plays moves at random with equal probability without regard to any previous knowledge. To level the playing field, this AI was also equipped with the deterministic winning move check described above. I ran 100 games of Naïve AI as Player 1 and several AI algorithms as Player 2. Table 1 summarizes the win percentages for Player 2 in these cases. It is clear that for large values of C, the MCTS AI behaves similarly to the naïve AI, and for smaller values it is more able to discern among the moves to select a better move. Also, a starting database of game states can noticeably improve the AI (though the construction of such a database does take time).

Unfortunately, this AI is still woefully inadequate against human players. The MCTS AI seems to adopt an aggressive strategy and would only infrequently block a winning move that an opponent would play, i.e. not blocking worked fine against the random AI, but not against humans that strategize. To remedy this, deterministic block-checking code was added to `evaluateNextStates()`. Unfortunately, it was not feasible to implement a corresponding feature in the naïve AI because the code relied on knowledge of previous states, something which was not provided to the naïve AI. However, anecdotal evidence was collected. Against an inexperienced human player (myself), out of 10 games the AI won 4, making it a reasonable challenge. While the merits of a deterministic block-checking component may be debated (e.g. "If the goal is an AI that reasonably simulates a human opponent, is it reasonable to assume a human will always find a block?" or even "Is it reasonable to have the AI adopt a deterministic strategy which may be exploited when a human may not always see the move that is being blocked?"), it undoubtedly produces a better experience in terms of gameplay.

3.1 Future Improvements

Because of the nature of this project, the implementation of the MCTS AI in R is not perfect. In addition to improvements that can be made undoubtedly be made to the algorithm to improve its performance, there are several minor bugs in the code that should be corrected. Among the minor bugs are:

- The move selection function doesn't gracefully handle a move that results in a full board. The way the v_i's are constructed isn't impacted by this as this is not treated as a win for Player 2, but this would be

a minor issue if the AI were used as Player 1. This is the cause for the warning message that sometimes appears: `In is.na(object) : is.na() applied to non-(list or vector) of type 'NULL'`.

- If the MCTS AI were run as Player 1 instead of Player 2, the v_i values would be incorrect because of the treatment of the tie games (ties would be counted as wins for Player 1). This would be a simple fix but would increase the size of the database file.
- The board is difficult to read. In a late release, the following improvements could be made:
  - Display the matrix upside down from how it currently is (to simulate chips falling in due to gravity).
  - Remove the NA values when displaying the board (they are necessary for the internal logic).
  - Potentially use better symbols than 0 and 1 (though they are necessary for the internal logic).

In terms of algorithmic enhancements, two spring to mind:

- Merge any deterministic scenario-checking into the construction of the v_i's so that the move selection function can avoid special cases.
- Change the move selection function to a more sophisticated approach, such as Simulated Annealing.

4 Conclusion

Despite having no previous experience with game programming or AI techniques, I found MCTS methods simple to understand, intuitive, and above all, effective. MCTS algorithms are very general and can be applied to many situations with little modification. However, merely implementing MCTS (or UCT) is not enough to provide an effective AI - the manner in which v_i's are assigned to each move and the function f which selects a move based on the v_i's are extremely important and depend on the game and the goals of the AI, but these choices are distinct from the research on MCTS and the difficulty in addressing them doesn't, in my mind, detract from the simplicity and effectiveness of MCTS.

References

Guillaume Chaslot, Jahn-Takeshi Saito, Jos W.H.M. Uiterwijk, Bruno Bouzy, and H. Jaap van den Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th Belgian-Dutch Conference on Artificial Intelligence*, pages 83–90, 2006.

Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In C. Darken M. Mateas, editor, *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, Menlo Park, CA, 2008. AAAI Press.

Spyridon Samothrakis, David Robles, and Simon M. Lucas. Fast Approximate Max-N Monte Carlo Tree Search for Ms Pac-Man. To Appear, 2011.

István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-Carlo Tree Search in Settlers of Catan. In *Proceedings of the Twelfth International Advances in Computer Games Conference (ACG'09)*, Pamploma, Spain, May 2009.

UCT for Games and Beyond. Everything Monte Carlo Tree Search. `http://mcts.ai`, 2010.

Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo Tree Search Solver. pages 25–36, 2008.