A study on space search strategies for real-time games

Milon Bhattacharya

E-mail: `milon.20csz0017@iitrpr.ac.in`

Abstract. Real-time multi-player games are characterized by many of players, extremely large search space and partially observable environment. However, due to the requirements of near real-time response, conventionally used search techniques like Alpha-Beta Pruning, which are known to work well for two-player board games are not computationally feasible. Variations to the algorithm has been used to solve this problem, however due to the large number of players such methods would not return satisfactory results. The study is an attempt at implementing a parallel version of the popular Monte Carlo tree Search algorithm. We created a reference implementation of the algorithm for the popular game of Tic-Tac-toe. In comparison to the naive algorithm , our implementation showed showed an twenty percent reduction in execution time, while the corresponding increase in memory requirements was much lesser.

1. Motivation and Problem Definition

With the advent of the Internet, multi-player games have been very popular [1]. A simple Internet search would reveal popular platforms like Second Life [12] which allow players from multiple locations to interact at the same time. Development of such games has been [8] possible only because of emergence of alternate engineering architectures for managing the computation and storage requirements. However, as the games have garnered more audience, the need of having a more *human-like* experience has been felt [6]. The use of conventional artificial intelligence and machine learning methods has been quite successful in meeting the above demand. Popular games like PUBG have automated game-playing agents, which mimic human behaviour very closely. However, as the game attempt better at playing the game and reach human-like performance, such agents either need an exponentially increasing amount of computation or a large number of training examples (if it is based on a deep learning approach) [15]. Hence, there has been a need for finding ways in which the resource requirements of modeling such multi-agent systems can be reduced. This is in spite of a drastic reduction on the cost of of computation and the emergence of cloud based platforms, which support dynamic resource allocation. Thus, there is a dire need to find ways and means which a graceful degradation can be obtained while optimizing the resources and time requirements.

2. Literature Review

The idea of approximating utilities using a stochastic process is not new. A very popular approach for finding the most optimal decision is using the Alpha-Beta Pruning [9] which limits the state space which has to be searched by looking at the payoffs that have been observed till that *ply*. The ease of implementation and understanding has ensured that the algorithm has been extended with many variations [5] [13]. A popular version of it has been extended for multi-player scenarios as well.[10] The problem with the approach is that the search method is *comprehensive* in nature. This can be a source of problem, when the possible actions a can be very large (possibly a among a continues set of options). The Monte Carlo Tree Search [14] attempts to solve this problem by avoiding to search the complete state-space instead, estimate a payoff for a move using a *stochastic* approach. This too has been a very popular approach, and has been shown to given statistically good results for game like Go [11]. The MCTS is able to solve the problem of high branching factor by statistically estimating, rather than comprehensively determining the payoff for a particular action. Also, it meet the criteria of graceful degradation. That is, as the number of played games increase, the recommendations of the MCTS would get better (closer) to those made by searching the state-space exhaustively. The current study is an attempt to solve this problem. Again, this has been comprehensively dealt with in previous attempts [17] [2]. Our aim in this study would be to make certain incremental changes in the approaches.

3. Proposed Solution

Monte-Carlo tree search (MCTS) was described in the study [7] as a best-first, anytime search technique that does not require an evaluation function to determine node values.The algorithm performs simulations in the search space to estimate node values and iteratively grow a search tree while focusing on parts of the tree with the best estimated values Instead of creating a comprehensive tree, the naive MCTS builds a game tree as the search progresses in real-time. This process is guided by simulation results so that the most promising parts of the tree are preferentially expanded, allowing computational resources to be used efficiently by avoiding unnecessarily searching less beneficial subtrees [3]. Each iteration of the MCTS algorithm is called an *playout* and consists of the four following steps:

(i) *Selection*: Starting at the root of the game tree, child nodes are recursively selected until a terminal state or a node that is not yet fully expanded (a node with actions that have not been considered yet) is reached.

(ii) *Expansion*: If the game state represented by the node is not terminal, a new child is added to the node, thereby expanding the tree. If the game state is terminal, the tree is not expanded and the value of the terminal node is backpropagated (see step 4) without the need for simulation.

(iii) *Simulation*: Starting at the newly expanded node, moves are applied to the game state until a terminal state is reached, at which point the value of the state (commonly 1 for a win and 0 for a loss) is obtained.

(iv) *Backpropagation*: The reward value obtained in the simulation phase is used to update the statistics for every node on the path from the newly expanded node to the root of the tree. This normally entails incrementing each node's visit count and updating average node rewards based on the simulation result

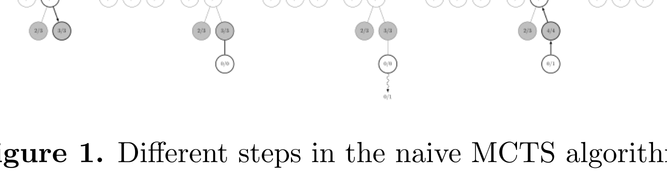

Figure 1. Different steps in the naive MCTS algorithm.

The main difficulty in selecting child nodes is maintaining some balance between the exploitation of deep variants after moves with high average win rate and the exploration of moves with few simulations. The first formula for balancing exploitation and exploration in games, called UCT (Upper Confidence Bound 1 applied to trees), was introduced in [7]. The authors recommend to choose in each node of the game tree the move for which the expression:

$$w_i/n_i + c * \sqrt{logN_i/n_i} \tag{1}$$

has the highest value. In this formula:
w_i stands for the number of wins for the node considered after the i^{th} move
n_i stands for the number of simulations for the node considered after the i^{th} move
N_i stands for the total number of simulations after the i^{th} move run by the parent node of the one considered
c is the exploration parameter—theoretically equal to $\sqrt{2}$; in practice usually chosen empirically.
The first component of the formula above corresponds to exploitation; it is high for moves with high average win ratio. The second component corresponds to exploration; it is high for moves with few simulations.
A possible problem in all versions of the MCTS is the use of synchronization primitives for ensuring that the multiple players don't interfere with each other. We proposed a solution inspired by [2], [6] and [16] in the section 1. It has also been referred to as the tree parallelizing approach to MCTS. The approach maintains a single tree for all agents. All such agents have the same *structure*, as in the possible actions and the payoff for a particular state of the game. Specifically, we follow the principles espoused in [4] which theorizes that simultaneous updates to node statistics happen infrequently, and the data corruption caused by this can be safely ignored. This assumption ensures that the acceptance of occasional write conflicts that might occur when adding a new node can allow us to avoid the latency due to the synchronization primitives.

Algorithm 1: Tree-Parallel MCTS

```
  Input  : A set of player $p_1, p_2 .... p_N$ , iters, and a Game $G$ which encapsulates the rules
           of the game
  Output: A Tree which gives the most suitable set of moves
1 for $i \leftarrow 0$ to $iters$ do
2     $root = MCTS - Node(Initial - state, Game)$
      foreach $p \in \mathcal{P}layers$ do
3         /* Execute this step in parallel                                  */
4         $Selection_p(root)$
          $Expansion_p(root)$
          $Simulation_p(root)$
          $Backpropogation_p(root)$
5     end foreach
6 end for
7 return $root$
```

4. Observations

4.1. Game 1: Tic-tac-toe

The following comparisons show the statistics relating to the two versions of the game tic-tac-toe. In the first game the two player run their own separate version of the MCTS algorithm individually. This was simulated by running both the agents as separate process. This is indicated as the Naive MCTS. In the second case the two players were run as separate threads which access a common MCTS tree. This is what is referred to as the Parallel MCTS in the graphs. The MCTS algorithm improves as the number of interactions (times the game was played) increases. All the experiments described in the section below were done on a cloud instance with 8 GB of RAM and 2 CPUs (a T3 Large instance in AWS terminology). The single thread version is an adaptation from (https://int8.io/monte-carlo-tree-search-beginners-guide), the source code for which is available here `https://github.com/int8/monte-carlo-tree-search`

4.1.1. Execution Time As evident in 2 the parallel version of the algorithm outperforms the naive version of the MCTS when it comes to execution time. When the iterations exceed *30000*, the speedup exceeds *2x*. Which means when both the agents share a MCTS tree, they would take less time to search and update, than they would separately.(specifically, an advantage of 25 percent).

4.1.2. Memory Consumption Further, the other graphs demonstrate the Peak 3 and the Average memory 4 consumption by both the methods.The parallel version of the program shows a linear increase in memory consumption as the number of iterations increase, which is desirable property. This means that the MCTS tree grows linearly

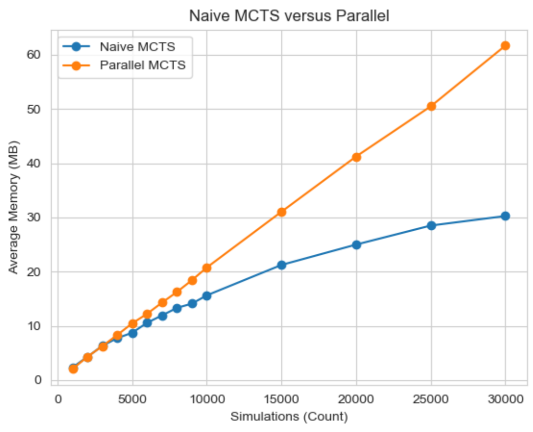

Figure 2. Comparison of Average Memory Requirements for both versions of the algorithm.

4.2. Game 2: Sheep and Caveman

The approach proposed in the study was demonstrated to work as expected for the game of *tic-tac-toe* which is small two-player board game. However, to prove its superiority for real-world game, the same had to be demonstrated on an arcade game. For that, we attempted to simulate a competitive game. This is inspired from one of the assignments which was part of the coursework for the subject. The game runs as following.

- The game has two players, a Sheep and atleast a caveman.
- The arena is a 2D board. For each *turn* of the game, the move of a sheep and the cavemen are single *plys*, but each iteration involves all the players making a move.
- the objective of the Sheep is to evade capture by anyone of the caveman. Each of the caveman wants to capture the sheep first.

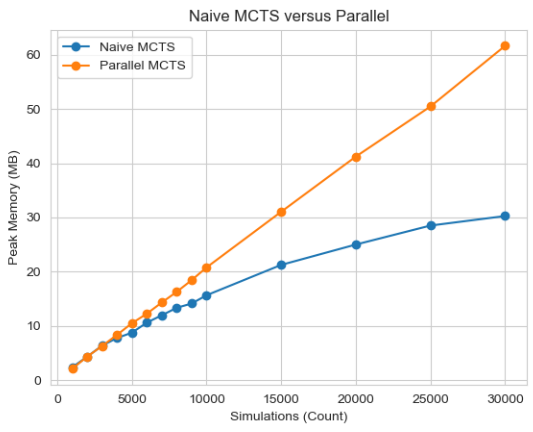

Figure 3. Comparison of Peak Memory Requirements for both versions of the algorithm.

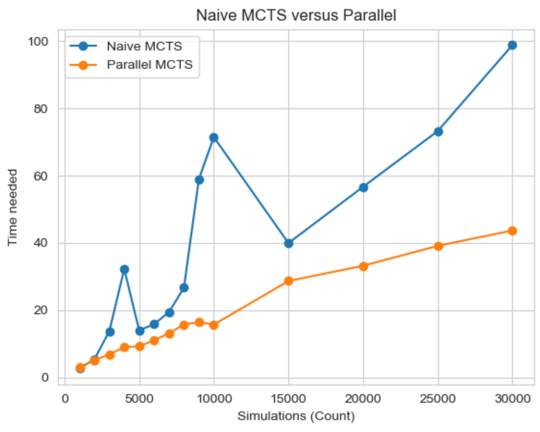

Figure 4. Comparison of execution time for both versions of the algorithm.

- Due to the limited size of the arena, the game is run for a set number of iterations. If anyone of the caveman is able to capture the sheep, that caveman wins and the game is finished. If the sheep is still not captured after the number of iterations are exhausted, then the Sheep wins.
- The caveman and Sheep have a maximum step size, however the sheep's step size is larger than the any of the cavemen.
- The sheep has a fixed field of view and would move as soon as it sees atleast a caveman in the *FOV*.
- The catch, however is if a caveman approaches a sheep from the back (Southern side of the arcade), then the sheep is not able to view the caveman even if the caveman is within the FOV.
- Finally, the arcade itself is not fully manoeuvrable. Obstacles placed at random locations are not usable by the players. This adds a level of difficulty and the players not only have to find the closest direction to move, but also avoid obstacles.

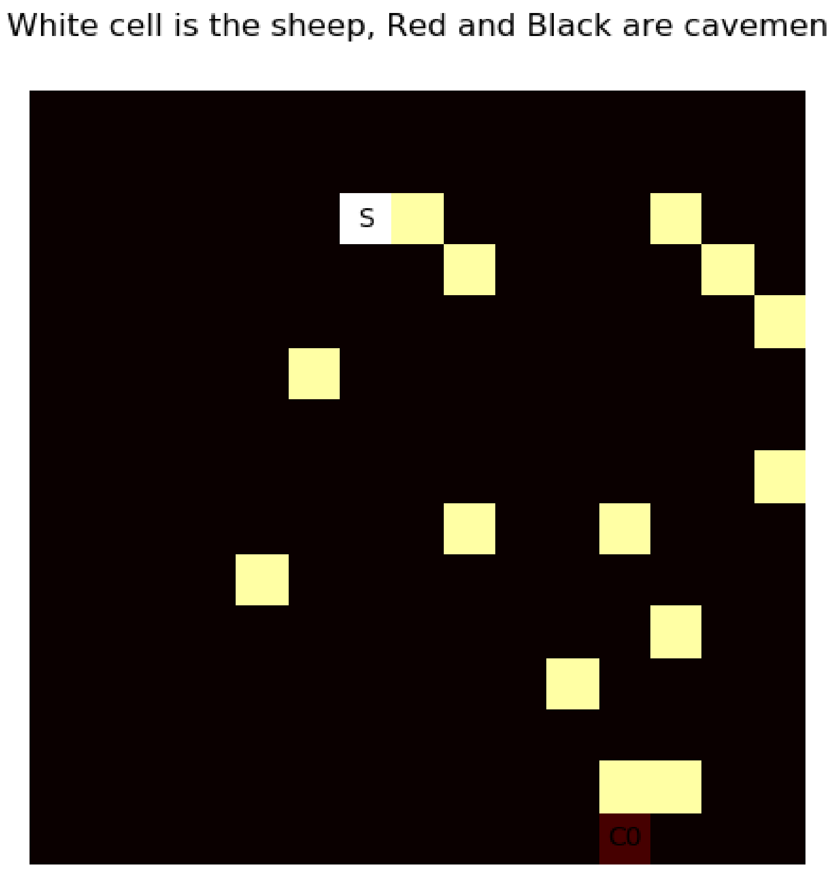

Figure 5. An animation of the game (Click to play)

4.2.1. Execution Time The simulation was performed on a $15X15$ arcade with a single sheep and ten cavemen. 6 shows the relative performance of the two approaches. On an average, the parallel approach outperforms the naive method by 10-30 percent.

4.2.2. Memory Consumption The memory consumption shows an even better trend. It the parallel approach consumes 5-6 percent less memory as compared to the naive approach. This is in contrast to 3 and 4 where the naive version was showed to consume less memory. The possible reason for this could be an experiment error for a game which is as simple as tic-tac-toe. However, for real-world arcade game, parallelism gets rid of the overhead, which results in reduction in memory requirements.
To compare both the approaches, different configurations of the arcade are created (initial locations of sheep,caveman and obstacles). Both approaches are used to find the optimal tree and the time and memory required are recorded.The implementation of the simple game is done in *simple_sheep_nothread.py* The implementation of both the methods discussed in this study and their comparisons can be found at `https://github.com/bmilon/MCTS`

5. Conclusion and future Work

The study is an attempt to make an incremental improvement over the very popular and successful MCTS algorithm. It was experimentally demonstrated that Tree parallelisation approach not only works for simple fully observable-deterministic games, but also can provide

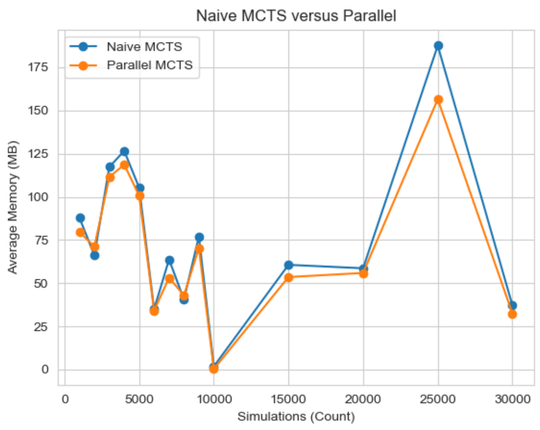

Figure 6. Comparison of Average Memory Requirements for both versions of the algorithm.

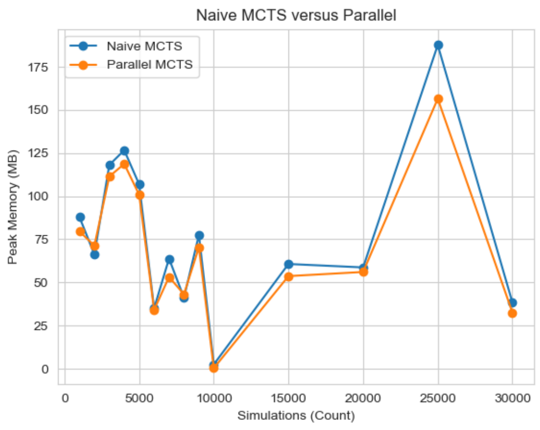

Figure 7. Comparison of Peak Memory Requirements for both versions of the algorithm.

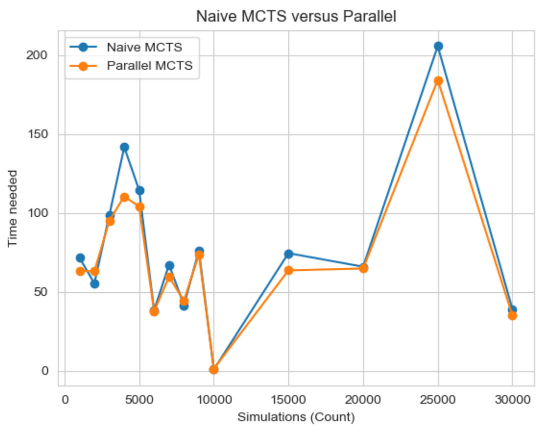

Figure 8. Comparison of execution time for both versions of the algorithm.

faster performance and reduced memory consumption for real-world arcade game. As, a future work this can be extended to even more *complex* games where the arcade is three-dimensional with multiple agents having even higher set of actions. A suitable candidate would be role player games like *Second-Life* which have thousands of humans competing against each other.

References

[1] Elaine Chan. "Massively Multiplayer Online Games". In: *American Psychological Association.* 2006. DOI: 10.1109/CIG.2018.8490403.

[2] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. "Parallel monte-carlo tree search". In: *International Conference on Computers and Games.* Springer. 2008, pp. 60–71.

[3] Marc Christoph. *Parallel Monte-Carlo Tree Search in Distributed Environments.* University of Stellenbosch, 2020.

[4] Markus Enzenberger and Martin Müller. "A lock-free multithreaded Monte-Carlo tree search algorithm". In: *Advances in Computer Games.* Springer. 2009, pp. 14–20.

[5] Chris Ferguson and Richard E Korf. "Distributed Tree Search and Its Application to Alpha-Beta Pruning." In: *AAAI.* Vol. 88. 1988, pp. 128–132.

[6] Stefano Ferretti and Marco Roccetti. "Fast Delivery of Game Events with an Optimistic Synchronization Mechanism in Massive Multiplayer Online Games". In: *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology.* ACE '05. Valencia, Spain: Association for Computing Machinery, 2005, pp. 405–412. ISBN: 1595931104. DOI: 10.1145/1178477.1178570. URL: https://doi.org/10.1145/1178477.1178570.

[7] Sylvain Gelly et al. "The grand challenge of computer Go: Monte Carlo tree search and extensions". In: *Communications of the ACM* 55.3 (2012), pp. 106–113.

[8] Thorsten Hampel, Thomas Bopp, and Robert Hinn. "A Peer-to-Peer Architecture for Massive Multiplayer Online Games". In: *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games.* NetGames '06. Singapore: Association for Computing Machinery, 2006, 48–es. ISBN: 1595935894. DOI: 10.1145/1230040.1230058. URL: https://doi.org/10.1145/1230040.1230058.

[9] Donald E. Knuth and Ronald W. Moore. "An analysis of alpha-beta pruning". In: *Artificial Intelligence* 6.4 (1975), pp. 293–326. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(75)90019-3. URL: https://www.sciencedirect.com/science/article/pii/0004370275900193.

[10] Richard E. Korf. "Multi-player alpha-beta pruning". In: *Artificial Intelligence* 48.1 (1991), pp. 99–111. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(91)90082-U. URL: https://www.sciencedirect.com/science/article/pii/000437029190082U.

[11] Byung-Doo Lee. "The first move in the game of 9 by 9 Go, using non-strategic Monte-Carlo Tree Search". In: *Journal of Korea Game Society* 17.3 (2017), pp. 63–70.

[12] THOMAS M. MALABY. *Making Virtual Worlds: Linden Lab and Second Life.* 1st ed. Cornell University Press, 2009. ISBN: 9780801447464. URL: http://www.jstor.org/stable/10.7591/j.ctt7z825.

[13] Abdallah Saffidine, Hilmar Finnsson, and Michael Buro. "Alpha-beta pruning for games with simultaneous moves". In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 26. 1. 2012.

[14] Mark HM Winands, Yngvi Bjornsson, and Jahn-Takeshi Saito. "Monte-Carlo tree search solver". In: *International Conference on Computers and Games.* Springer. 2008, pp. 25–36.

[15] Jiarong Xu et al. “NGUARD+: An Attention-Based Game Bot Detection Framework via Player Behavior Sequences”. In: *ACM Trans. Knowl. Discov. Data* 14.6 (Sept. 2020). ISSN: 1556-4681. DOI: `10.1145/3399711`. URL: `https://doi.org/10.1145/3399711`.

[16] Xiufeng Yang, Tanuj Kr Aasawat, and Kazuki Yoshizoe. *Practical Massively Parallel Monte-Carlo Tree Search Applied to Molecular Design*. 2021. arXiv: `2006.10504 [cs.AI]`.

[17] Shubu Yoshida et al. “Application of Monte-Carlo tree search in a fighting game AI”. In: *2016 IEEE 5th Global Conference on Consumer Electronics*. IEEE. 2016, pp. 1–2.