Flat Monte-Carlo Tree Search and UCT in the Game of Checkers

Zach Foster
foste448@umn.edu

May 11, 2016

1 Introduction

Monte-Carlo tree search (MCTS) is a best-first search method used to make decisions. MCTS is especially valuable to game playing because it is an anytime algorithm, and it doesn't require a heuristic. In this paper I will examine various methods of tree search, both informed and uninformed, minimax search, the Monte-Carlo method, and various Monte-Carlo tree search methods.

During my examination of all of these methods I noticed a distinct lack of the game of checkers in the evaluation of algorithms presented. Therefore, I have implemented two agents for the game of checkers, each using a different variation of MCTS. The first variation of MCTS I will implement is the most basic, flat MCTS, and the second variation is the Upper Confidence Bounds for Trees (UCT) algorithm. After giving a brief overview of my implementation of the agents I give a brief overview of the experiments I ran in order to compare the agents. Finally I review the results of my experiments and conclusions I have drawn.

2 Related Work

2.1 Tree Search

Tree search is often used to solve both single player problems such as puzzles and pathfinding. Tree search algorithms work by considering various possible action sequences. The initial state of the problem is represented by the root node of the tree. Each branch represents an action, and children nodes represent the state resulting from an action being performed on their parent node's state. Tree search algorithms take advantage of the tree structure to solve single player games by finding a winning state and then applying the sequence of actions taken to the starting state.

2.1.1 Uninformed Tree Search

Uninformed tree search strategies, also known as blind search, only know what is defined in the problem definition[11]. The problem definition defines the requirements of a goal state, and actions that can be taken. With this information uninformed searches traverse the search tree by taking actions to new states, and evaluates whether the current state is a goal state or non-goal state. Some examples of uninformed tree search algorithms are breadth-first search, depth-first search, and iterative deepening depth-first search.

Breadth-first tree search starts by expanding the root node, then it expands all children of the root node, and continues this process until a goal state is found. At a given depth, n, all nodes at depth n are explored before any nodes at depth n + 1. It is easy to see that breadth-first search is complete because it will expand to all possible game states. Breadth-first search also finds optimal solutions if the path cost is a nondecreasing function of the depth of the node[11]. The most common case of this is when all actions have the same cost. While breadth-first search is complete and finds optimal solutions, it can be costly in terms of space and time. The number of nodes expanded is $O(b^d)$[11] where b is the branching factor and d is the depth of the solution. Because breadth-first search doesn't backtrack or reexamine any nodes, this is also the time complexity. The space complexity is the main issue for breadth-first search because CPU clock times have increased at a much quicker rate than memory.

Depth-first tree search expands to the deepest possible node in the search tree. It starts by immediately traversing to a leaf node. The search then backs up and traverses to the next deepest node and continues this process. Depth-first search has a much better space complexity than breadth-first search, O(bm)[11] where m is the height of the complete search tree. Depth-first tree search is not complete because it can get stuck in infinite loops between nodes. Depth-first tree search also has a worst time complexity than breadth-first tree search, $O(b^m)$[11].

A popular variation of depth-first tree search is iterative deepening depth-first search (IDDFS). IDDFS is depth-first search, but the depth it is allowed to traverse is restricted. The allowed depth is then iteratively increased. IDDFS is complete, finds optimal solutions, and has a space complexity of O(bd)[11]. The only disadvantage is the time complexity of $O(b^d)$. Because of its advantages, IDDFS is a good choice for uninformed tree search.

2.1.2 Informed Tree Search

Informed tree search strategies use problem specific knowledge other than the problem definition. By leveraging this extra information, informed tree searches can find solutions more efficiently than uninformed search methods. The problem specific knowledge usually finds its way into the informed tree search algorithm through a heuristic function. A heuristic function gives an estimate of the cheapest path from the current node to a goal node. I will examine A* search because it is one of the simpler, and more popular informed tree search strategies.

A* search evaluates nodes in the search tree by combining the cost to reach the current node from the root node with the heuristic value of the current node. A* keeps track of frontier nodes in a priority queue, and always expands to the lowest cost node[11]. A* is both complete and optimal; however, only if its heuristic function is admissible[11]. A heuristic function is admissible if it never overestimates the cost to reach the goal from a node[11]. Because A* is complete and optimal, it is often used in pathfinding problems and single player puzzles when a heuristic is known. The fact that it requires a heuristic in the first place is its main disadvantage because they can be difficult to find.

2.2 Minimax Search

Another very popular algorithm used in game playing is minimax. Minimax is a search algorithm used in two person, zero-sum games of perfect information. Minimax assigns a value to every node in a game tree by first giving values to terminal nodes that represent the desirability of the position. Minimax recursively gives the non terminal nodes values based on whose turn occurs at

that node. If one players turn, the node is equivalent to the maximum value of the children, if the other players turn, the node is equivalent to the minimum value of the children nodes [2]. Basic minimax is troublesome for more complex games because the search space can become so large. Because of this, many game playing programs implementing minimax use heuristics to estimate the value of nodes when a terminal node isn't within a specified depth. These heuristics can be hard to find and some rely on expert knowledge of the game being played. MCTS is better in the sense that it doesn't require a heuristic. Another advantageous variation to minimax is alpha-beta pruning. Alpha-beta pruning prunes parts of the search tree that are guaranteed not to hold an optimal solution [2]. Minimax is often better than MCTS in games with good heuristics or smaller search trees, but fails to compete with MCTS in more complex games such as Go.

2.3 Monte-Carlo Method

The Monte-Carlo method is the method of performing random samples on a domain, then using the results of the samples to make inferences about the domain. The Monte-Carlo method was originally created by Stanislaw Ulman in 1947, and was used in the Manhattan Project to gain insights on the problem of neutron diffusion in fissionable material[10].

The Monte-Carlo method is useful in situations where no current mathematical method can solve a problem. By running a large number of random samples on an object or system, you can begin to gain insights about the object or system. Another technique is to run sufficiently long random simulations rather than a large number of simulations[6]. At its heart, the Monte-Carlo method relies on the Law of Large Numbers, and leverages randomness as a strength[6].

The Monte-Carlo method is often used to simulate a real life system and gain insights about it. We can gain insights to anything from production lines to telecommunication networks by performing the Monte-Carlo method on them. The Monte-Carlo method can be used as a tool to estimate numerical quantities such as the throughput of a production line[6].

3 Monte-Carlo Tree Search

Monte-Carlo tree search (MCTS) is a best-first search method used to make decisions, and has gained its popularity in game playing. MCTS is especially valuable to game playing because it doesn't require a heuristic, although its performance can often be enhanced with the use of heuristics[7]. MCTS is also an anytime algorithm, it can return a value at anytime of calculation. As the time it is given increases, its solution is expected to be better. I will give a summary of the MCTS algorithm [4] [1], and examine variations of the algorithm [7]. First I will look at the most popular version of MCTS, UCT. The next variation recursively nests MCTS for single player games [3], the third variation alters the MCTS for the single player games [12], and the fourth explores parallel implementations of MCTS for the game Go [5].

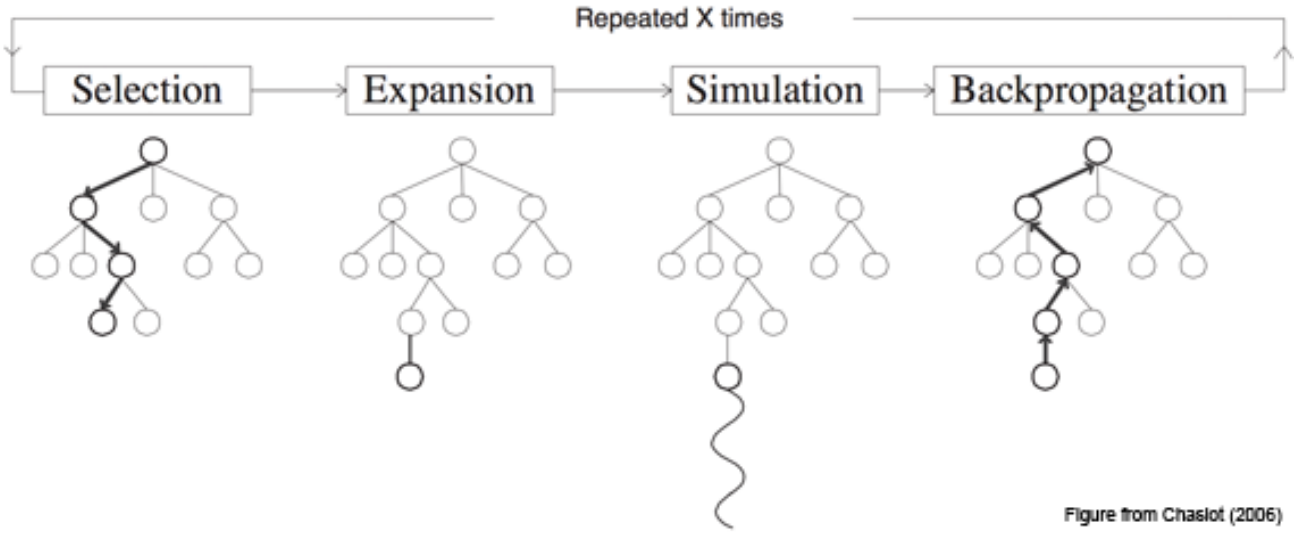

Figure 1: The four stages of Monte-Carlo tree search[4]

MCTS is comprised of four main stages, selection, expansion, simulation, and back-propagation [4]. Each node in a MCTS keeps track of the number of simulations performed on it, and the number of simulations that resulted in a victory.

3.1 Selection

Selection starts at the root (current position in the game) and repetitively selects child nodes until it reaches a leaf node. The node with the largest confidence interval upper bound is chosen. Each node's confidence interval depends on the specific implementation's balance of exploitation and exploration. Exploitation favors well traveled nodes with large rewards and narrow confidence intervals while exploration favors less traveled nodes [1].

3.2 Expansion

Unless the leaf node marks the end of the game, expansion occurs and a new node is added to the Monte-Carlo search tree.

3.3 Simulation

During simulation a play-out is performed at the new node. This play-out uses the Monte-Carlo method to perform a stochastic choice of moves for each player until the game ends.

3.4 Backpropagation

During backpropagation all nodes on the current path from the selection stage have their win and play values updated based on the outcome of the play-out [1]. Some variations of MCTS such as Implicit Minimax Backups in MCTS also update each node with a heuristic value calculated with the minimax backup rule based on children [7]. These values are then taken into consideration

during the expansion stage of the algorithm. Finally, MCTS picks the move with the highest simulation win rate when time runs out [4].

3.5 Variations of MCTS

The most popular variation of MCTS is The Upper Confidence Bounds for Trees (UCT) algorithm. This algorithm uses upper confidence bounds to handle it's balance of exploitation and exploration during selection [9]. UCT is based off of a upper confidence bound used in bandit problems (UCB1)[1]. Both UCT and UCB1 are simple, and guaranteed to be within a constant factor of the best possible bound on the growth of regret. UCT has been successfully used in many other variations of MCTS including Nested-Monte Carlo Search, which I examine next.

Cazenave [3] presents Nested Monte-Carlo Search (NMCS). NMCS combines randomness in play-outs with nested calls in order to guide its search toward better states. Nodes have score values, not win/play ratios because NMCS is applied to single player games. Starting at the root, NMCS randomly tries all possible moves, plays a nested search at the lower level after each move, and memorizes the move associated to the best score of the lower level searches [3]. The number of nested levels is specified. Because of the randomness of the play-outs, improvements in nested searches is not guaranteed. To account for this NMCS memorizes the best sequence found so far. When time runs out the best sequence is returned.

Maarten et al.[12] presents Single-Player Monte-Carlo Tree Search (SP-MCTS). SP-MCTS uses a new selection strategy to calculate a node's confidence interval that takes into consideration the range of score possible in the game being played. SP-MCTS's simulation strategy can be random as it is in basic MCTS, or pseudo-random based on heuristic knowledge of the game it is playing. Because SP-MCTS is for single player games in which the player receives a score, nodes do not keep track of wins or losses. Instead, they keep track of the average score of simulations they were a part of, this is updated during back-propagation. SP-MCTS isn't turn based. Instead of having a set time for each turn and returning a single move, SP-MCTS performs one large search from the initial position and then plays all the moves at once. In order to avoid local maxima, this process is time-limited, and restarted with a different random root. The solution with the best score is then returned as an answer.

Cazenave [3] presents Nested Monte-Carlo Search (NMCS). NMCS combines randomness in play-outs with nested calls in order to guide its search toward better states. Nodes have score values, not win/play ratios because NMCS is applied to single player games. Starting at the root, NMCS randomly tries all possible moves, plays a nested search at the lower level after each move, and memorizes the move associated to the best score of the lower level searches [3]. The number of nested levels is specified. Because of the randomness of the play-outs, improvements in nested searches is not guaranteed. To account for this NMCS memorizes the best sequence found so far. When time runs out the best sequence is returned.

Chaslot et al. [5] presents Parallel Monte-Carlo Tree Search (PMCTS). PMCTS discusses two methods of parallelization for MCTS, leaf parallelization and root parallelization, and introduces a third method, tree parallelization. In leaf parallelization one thread performs selection and expansion, simulation for the leaf nodes takes place on separate threads, then back-propagation occurs on a single thread once all simulations are finished. Root parallelization builds multiple MCTS trees, each one having its own thread. Once time is up, the correlating root's children are combined from the separate trees, and the best move is chosen. Tree parallelization shares one tree across multiple threads, each of which runs a MCTS. Mutexes are used to lock certain parts

of the tree to prevent data corruption. Tree parallelization also uses 'virtual losses" to balance exploitation and exploration in a way that is advantageous to parallelization. When a node is visited by a thread it is assigned a "virtual loss," and it's value is decreased until back-propagation occurs. This keeps multiple threads from exploring uncertain nodes, but doesn't stop multiple threads from exploring promising nodes. Root parallelization was proven to be fastest, followed by tree parallelization, and finally leaf parallelization.

All examined modifications to MCTS can be advantageous when applied to games. UCT is easy to apply, and can be used in conjuction with other modifications such as NMCS. The SP-MCTS and NMCS modifications allow for single player game playing. During their experimentations the SP-MCTS and NMCS articles [3][12] applied their algorithms to SameGame. SameGame is a puzzle that starts with a grid of squares each colored one of five colors. The objective is to match orthogonal groups of same colored squares. Upon matching, the squares disappear and (number of blocks removed -2)^2 points are awarded. Blocks fall downwards to fill gaps, and columns shift left to fill empty columns. SP-MCTS held the highest total score, 73,998, across 20 positions of a test set at its time of publication. NMCS beat it with a score of 77,934 with a level specification of three. NMCS also hold the world record for Morpion Solitaire disjoint version, a NP-hard puzzle. The third article discussed [5] addresses parallelization. The experiment to test the parallelization techniques was not performed in a single player game setting like the other two modifications. It was performed on the two player game Go against the GNU Go [5]. While the method of testing used is different, parallelization itself may be applicable to the other two modifications. For instance, SP-MCTS's method of restarting with a random root to avoid local maxima may be done in parallel to increase the performance of the algorithm.

4 Flat MCTS and UCT in the Game of Checkers

In my experiment I implement and compare two agents that use simple variations of MCTS to play the game Checkers Lite[8]. Checkers Lite is a simplified version of checkers in which there are no kings, and at most a single jump can be made per turn. While this is not a complete implementation of Checkers, I believe it is sufficient to compare the two variations of MCTS.

4.1 Flat MCTS

In my experiment I implement flat Monte-Carlo tree search, which is the most basic implementation of MCTS in the game of Checkers Lite. Flat MCTS implements MCTS as described above, and uses a uniformly random (flat) distribution during the selection stage. This means flat MCTS ignores expansion and exploration by randomly selecting nodes during selection instead of an equation.

4.2 UCT

I also implement the UCT algorithm to play Checkers Lite in my experiment. The UCT algorithm is described above, and the equation used during selection to balance exploration and exploitation during the MCTS is given below. The most desirable benefit of the UCT algorithm is its guarantee to be within a constant factor of the best possible bound on the growth of regret. $\bar{X}_j$ is the $\frac{win}{visit}$ ratio of the node and represents exploitation. The remainder of the equation represents exploration. The constant I use for C_p is $\frac{1}{\sqrt{2}}$ because it was shown to satisfy the Hoeffding inequality with rewards

in the range 0...1[1]. n is the number of times the current node has been visited while n_j is the number of times the child node under consideration has been visited.

$$\text{UCT} = \bar{X}_j + 2C_p\sqrt{\frac{2ln(n)}{n_j}}$$

5 Implementation and Experiment Design

5.1 Implementation

I began the comparison of flat MCTS and UCT by implementing both algorithms in the programming language Swift. All implementations and experiments were completed with the IDE Xcode. The main components of my implementation are a State type to represent the state of the Checkers Lite game, and a Node type to represent a node in the search tree. My State implementation was accomplished with the help of Steven Materra's implementation of Checkers Lite[8] for iOS, and my Node implementation was accomplished with the help of Peter Cowling, Ed Powley & Daniel Whitehouse Python implementation of UCT that was previously available at mcts.ai (no longer hosted).

I also implemented UCT and Flat functions to perform selection, expansion, rollout, and backpropagation for a designated amount of iterations and return the best move choice for the root node. Finally I implemented a main function to play games of Checkers Lite with various agents as players, record the game results, and record average CPU time of the agents' turns. I give descriptions of my types below.

```
class CheckersState: State, CustomStringConvertible {
  var boardData = [[SpaceType]]()
  var currentPlayer = Player.None
  var description: String
  func copy() -> State
  func setupBoard()
  var currentPlayer: Player { get set }
  func getMoves() -> [Move]
  func makeMove(move: Move)
  func gameIsOver() -> Bool
  func getResult(player: Player) -> Result?
}

class Node {
  var player: Player
  var moves: [Move]
  var root: Bool
  var parent: Node?
  var move: Move?
  var children = [Node]()
  var wins = 0.0
  var visits = 0

  init(state: State)
```

```
    init(state: State, move: Move, parent: Node)
    func selectUCT() -> Node?
    func selectFlat() -> Node?
    func expand(move: Move, state: State) -> Node
    func update(result: Result)
}
```

5.2 Experiment Design

In order to compare UCT, flat MCTS, and random move selection agents I played UCT and flat MCTS against random move selection with varying amount of MCTS iterations to develop an idea of how many iterations it takes for the MCTS agent's to gain an advantage over random move selection. I then played games of UCT vs. flat MCTS agents to analyze how they performed against each other at varying number of iterations.

6 Results

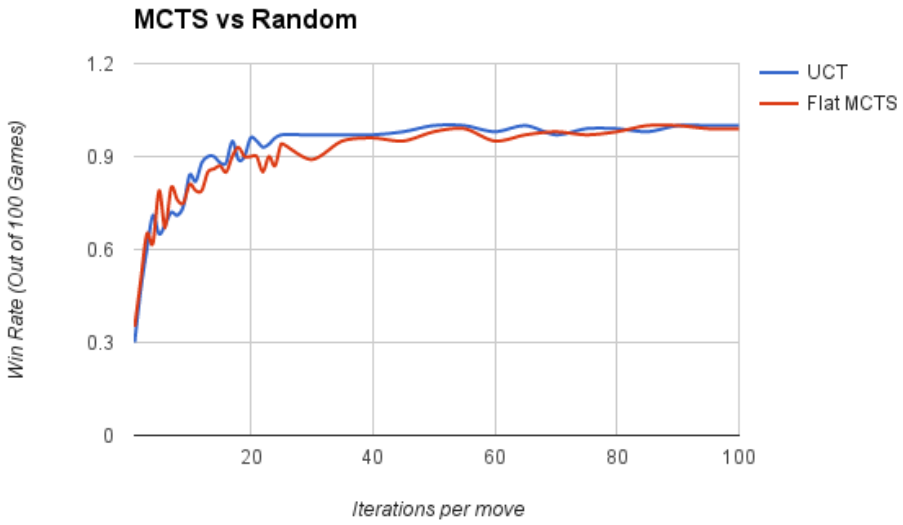

Figure 2: Results of flat MCTS vs Random and UCT vs Random

The results of the MCTS agents vs the random choice agent show that both MCTS agents win at least 95% of the time after 30 iterations/move. Flat MCTS seems to perform better than UCT during the 5-10 iterations/move period, but UCT tends to outperform flat MCTS for the remainder of the measurements. Both MCTS agents seem to immediately dominate the random choice agent and consistently win most of the time at an iteration/move count of only three. It is, of course, expected that the MCTS agents would outperform the random agent, but the low number of iterations/move before they dominate surprises me. After examining these results I decided to

start the comparison of UCT vs flat MCTS at 100 iterations/move because this is where they both clearly dominate the random agent.

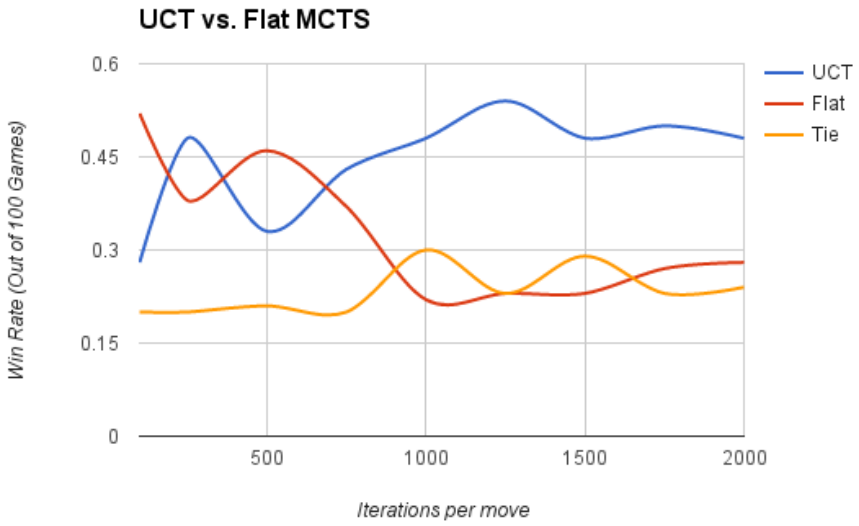

Figure 3: Results of UCT vs Flat MCTS

The results of the UCT vs flat MCTS agents shows that the UCT agent begins to perform better than the flat MCTS agent around 750 iterations/move. Before 750 iterations/move neither agent performs better than the other for a prolonged period. I found it useful to include the amount of ties between the agents in this results chart to visualize the number of ties because it is a significant amount. The number of ties between these two agents is significant and remains above 15% for all data points, and peaks at 30% around 1000 iterations/move. Based on these results I can conclude that the exploration vs expansion method implemented in the UCT agent outperforms the flat MCTS agent in Checkers Lite when iterations/move exceeds 750. After 750 iteration/move the UCT agent outperforms the flat MCTS by an average of 21.8%. This increase in performance is significant, and begins to steady out as iterations/move increases to 2000 iteration/move.

7 Conclusion and Future Work

Throughout this paper I have investigated tree search methods, both informed and uninformed, the Monte-Carlo method, and many variations of MCTS. I then devised and carried out an experiment to compare a UCT agent and a flat MCTS agent in a simplified version of checkers, Checkers Lite. Lastly I drew conclusions from my experiment. I concluded that both UCT and flat MCTS agents easily outperform an agent the chooses its moves completely at random. I also concluded that UCT beats flat MCTS in the game of Checkers Lite by an winning an average of 21.8% more games after 750 iteration/move. In the future I would like to fully implement the game of checkers. I would then like to implement a nested Monte-Carlo search agent and compare its performance in the game of checkers against the UCT and flat MCTS agents. Next, I would like to explore parallel implementations of the agents. I may face some difficulty implementing parallelization in

the agents because the Swift language doesn't currently support posix threads. Apple has a library for synchronization and parallelization known and Grand Central Dispatch, but it doesn't offer as much control over parallelization as threading. Apple does have plans to implement POSIX threads in the Swift language and I may wait until then to implement parallelization. Lastly, I would like to implement other games such as chess and Go, and compare each agent's performance on them. This would take a considerable amount of work to implement the game, but once the games were implemented I would be able to run the agents on them immediately because they aren't domain specific.

References

[1] C. Browne. Monte carlo tree search, 2010.

[2] M. S. Campbell and T. A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[3] T. Cazenave. Nested monte-carlo search. In *IJCAI*, volume 9, pages 456–461, 2009.

[4] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[5] G. M.-B. Chaslot, M. H. Winands, and H. J. van Den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71. Springer, 2008.

[6] T. T. Kroese D P, Brereton T and B. Z. I. Why the monte carlo method is so important today. *WIREs Comp Stat*, 2014.

[7] M. Lanctot, M. H. Winands, T. Pepels, and N. R. Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[8] S. Materra. Checkers lite. `https://gitlab.stevenmattera.com/iOSExamples/CheckersLite/tree/master`. Accessed: 2016-04-28.

[9] J. Méhat and T. Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):271–277, 2010.

[10] N. Metropolis. The beginning of the monte carlo method. *Los Alamos Science*, 15(584):125–130, 1987.

[11] S. Russel and P. Norvig. Artificial intelligence: A modern approach. *EUA: Prentice Hall*, 2010.

[12] M. P. Schadd, M. H. Winands, H. J. Van Den Herik, G. M.-B. Chaslot, and J. W. Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games*, pages 1–12. Springer, 2008.