AI Final Writing

Cole Thompson, Thomas Rooney

May 2021

Abstract

Google's DeepMind has developed programs that regularly beat the best humans in the world at these skill-based games and they did it with the help of an algorithm called Monte Carlo Tree Search. Monte Carlo Tree Search will choose the most promising move based on any given game state. In our research, we found that all of the games Monte Carlo Tree Search was implemented on were skill-based games like the previous games mentioned above, so we wanted to test Monte Carlo Tree Search on a mostly random game and see if it could regularly beat random move bots. The main goal of this research is to test if Monte Carlo Tree Search can reliably beat humans in random games just as it does playing skill/strategy-based games. We implemented a Sorry! simulation that plays out a specified amount of games where the players are either a random move AI or a Monte Carlo Tree Search AI. We then ran a multitude of simulations that tested the Monte Carlo Tree Search AI against one, two, or three random move AI's. We also experimented with tweaking the parameters c, d, and s in order to test how they affected the results. The results of these simulations provided adequate information that with the optimal values of c, d, and s that the Monte Carlo Tree Search AI reliably beat the random move AI's about 70 percent of the time. These findings are very significant as we were not expecting the Monte Carlo Tree Search AI to perform so highly, as we initially expected it to not make a difference because the game of Sorry! is so random.

Problem Description

Monte Carlo Tree Search has been successfully applied to many multiplayer board games. We have observed that the majority of these games are strategy-oriented where, if played by humans, the most skilled player has the highest probability of winning. Examples of these games are chess, go, bridge, poker, and many others. In these games, Monte Carlo Tree Search will almost always win against humans. Knowing all of this, we wanted to test Monte Carlo Tree Search's performance on a multiplayer board game that relies heavily on randomness, thus we decided on the game Sorry!.

Sorry! is a multiplayer board game usually played by four players, but can be played with as little as two. Each player has four pawns that start in each players respective starting position. The objective of the game is to then get all the pawns into your "home" by moving the pieces around a rectangular track on the outside of the board before any other player does. Players take turns drawing a card and then making a move by taking direction from that card. Some cards move forward, some backward, and there is a special Sorry card that allows you to overtake another player's pawn on the board and send their pawn back to that player's starting position. This drawing a card and making a move based on what you drew makes this game mostly random as you can only make moves based on what the card says. The choices a player does make is what pawn to move and what pawn to overtake if a sorry card is drawn. There are a few more nuances to the game so we will go into more depth about the mechanics of the game in a later section of this paper.

Testing Monte Carlo Tree Search on a mostly random game like Sorry! will be very interesting because we haven't been able to find any research on a similar random game. Most of the research is directed on the most complex strategy games that one would think only advanced humans could master. Thus, what we are trying to do is somewhat of a regression into less complex random games, which is still an interesting problem to solve as many natural game-like scenarios in life are mostly random.

Background

Monte Carlo Tree Search is a best-first search algorithm that is guided by the results of the previous runs of Monte Carlo simulations. This algorithm has been used over the years for many different situations, but most commonly is used in games to help guide an AI to predict the different moves that should be taken to win. In this literature review, Tom will be doing an overview of Monte Carlo Tree Search, while Cole will go over modifications that can be made to help improve its speed and accuracy for different situations.

With the emergence of many different complex games that humans have created, computer scientists have been looking for different methods and algorithms to increase the probability of winning those games. One such algorithm that has been popular in playing board games is Monte Carlo Tree Search. Guided by Monte Carlo simulations, Monte Carlo Tree Search is a best-first search method that combines the precision of tree search with the generality of random sampling. [1]

Monte Carlo Tree Search consists of four main phases respectfully termed selection, expansion, playout, and backpropagation. In the first phase of selection, the tree is traversed from the root until it reaches a node that doesn't end the game in a win or loss state and that has children that have not been added to the tree yet. At this point, the child node with the highest score is selected. The score of the child node i relies on the following formula where v_i is the score, n_i and n_p are the number of times that the child and parent node p have been visited, and C is a constant:

$$v_i = \frac{s_i}{n_i} + C * \sqrt{\frac{ln(n_p)}{n_i}}$$ [9]

In different variations of Monte Carlo Tree Search, this equation is modified in order to try and optimize the results of the search. The second phase of Monte Carlo Tree Search, expansion, includes adding the node that was selected from the selection phase to the tree. Next, we have the playout phase. In the playout phase, moves are played from the newly added node, usually by random moves, until a game is finished. The result of the game, which is a 1 for win and a 0 for loss for a single player. If it is a multiplayer game the result is stored as an array that is the size of the number of players participating in the game. The winner receives a 1 at their index and the losers receive a 0. In the final phase, backpropagation, this result is then propagated back along the traversed path up to the root node. This then updates the score of all the previous nodes on that given path.

The four phases of Monte Carlo Tree Search can be thought of as a single iteration which then gets repeated for a certain time period or a predetermined number of iterations. Usually the number of iterations it takes to get a large enough sample size is hundreds, thousands, or more. After the entirety of this process is completed, the method returns the child of the root node with the highest win percentage, signifying what the algorithm thinks is the best next action to take.

Now we take a look at how one might modify the Monte Carlo Tree Search. A popular version of the MC algorithm is the one used for the game Go, with the first Monte Carlo Go algorithm appearing in the early 90s, and has been continuously modified over the years. One such modification is made in Wang and Gelly's version of Monte-Carlo Go, where each Go board situation is treated as a bandit problem, the bandit problem being "a simple machine learning problem based on an analogy with a traditional slot machine (one-armed bandit) but with more than one arm. When played, each arm provides a reward drawn from a distribution associated to that specific arm. The objective of the gambler is to maximize the collected reward sum through iterative plays." [13] where each legal move that can be made is an arm in the bandit problem. Their MoGo algorithm then uses an extension of minimax tree search called UCT, which "continues playing one sequence each time, ... after each sequence, the value of played arm of each bandit is updated iteratively from the father-node of the leaf to the root." [13] While good, the MoGo algorithm could still be further improved upon by introducing pruning by using a heuristic(s) to help reduce the large size of the tree to make exploration faster.

In another article, Gelly and Silver discuss two more enhancements that can be made to further improve Monte Carlo. The first being their Rapid Action Value Estimation (RAVE), which shares each value of all the actions across each subtree. And the second being a heuristic Monte Carlo Tree Search, which initializes the values of new positions in the tree. [5] One of the main problems with Monte Carlo Tree Search is having to estimate the cost of each node multiple times for each different search tree. With Gelly and Silver's introduction of RAVE, which uses the All Moves As First heuristic, they have a simple way to share information between related nodes in the tree. This significantly improves the performance of the algorithm. Gelly explains "The basic idea of RAVE is to generalize over subtrees. The assumption is that the value of action a in state s will be similar from all states within subtree T(s). Thus, the value of a is estimated from all simulations starting from s, regardless of exactly when a is played." [5] Using the RAVE is good for games where nearby changes don't affect the value of moves next to them, because of how quickly it is able to learn. However, this is rarely the case in most games. To combat this issue, they combine both their RAVE algorithm with Monte Carlo. This then gives both the accuracy of Monte Carlo while also providing the speed from RAVE.

Next, we move to Gelly and Silver's second modification. Since we know that Monte Carlo Tree Search branches exponentially, a vast majority of the nodes in the tree are rarely visited, especially near the leaf nodes. To help against this uncertainty, a heuristic is used when initializing new nodes to more accurately reflect their value. Using a heuristic to help initialize the nodes, they found their winning percentage when playing Go against a MoGo algorithm increased from 60% to 69%.

The Monte Carlo Tree Search algorithm has been used in several different applications over the years, primarily in different strategy games. Monte Carlo Tree Search is carried out by a four-phase process that essentially runs random simulations that then build on each other by identifying an underlying pattern of which path gives you the highest probability of winning in the given game. The process consists of the following phases of selection, expansion, playout, and backpropagation, which are repeated for a fixed number of times or for a certain time period. After all of this, the method then returns the child of the root node that has the highest winning rate, signifying what the algorithm thinks is the best move to take next. Monte Carlo is also an algorithm that can have modifications made to create new more efficient algorithms. Whether it be from introducing new algorithms to combine with it, or introducing a heuristic to help more accurately predict the value of each node or to prune the tree. Whatever the case may be, Monte Carlo is an invaluable algorithm.

Approach to Solution, Representation, and Algorithms

To first solve this problem, we need a working version of the game Sorry!. After looking online for public code for the game, we decided that it would be easier if we wrote the game ourselves. This also allows us to write the code in a convenient way for the Monte Carlo algorithm to use. To start, we first need to go over the rules of sorry that we need to follow.

- There can be a max of 4 players at a time.

- The board consists of 60 spaces around the board, 1 space for each player's start, and 6 spaces for each player's home.
- Each player has a total of 4 tokens that they move around the board.
- Tokens move clockwise around the board, unless a card instructs them otherwise.
- There are a total of 45 cards in the deck, which consists of 5 1's, and 4 of every other kind 2, 3, 4, 5, 7, 8, 10, 11, 12 and 4 sorry cards.
  - 1: Move one token forward one space, or move one token out of home
  - 2: Move one token forward two spaces, or move one token out of home. Draw again.
  - 3: Move one token forward three spaces.
  - 4: Move one token backwards four spaces.
  - 5: Move one token forward five spaces.
  - 7: Move one token forward seven spaces, or split the seven spaces between two tokens. If you split between two tokens, you must be able to move all seven spaces.
  - 8: Move one token forward eight spaces.
  - 10: Move one token forward ten spaces, or move one token backward one space.
  - 11: Move one token forward eleven spaces, or swap one of your tokens with an opponent's token. Both tokens must be outside of the homes and start.
  - 12: Move one token forward twelve spaces.
  - Sorry: Move one of your tokens from your start area to take the place of an opponent's token, which is then sent back to their own start area. Opponents token must be outside of the homes and start.
- When moving you cannot land on the same space as your other tokens, except in your start and at the end of your home.
- If you land on the same space as an opponent's token, you send their token back its start.
- You may move past both your own and your opponent's tokens, counting it as one space.
- There are two slides of each color around the board, landing on your matching color with a token slides that token forward to the end of the slide. If there are any tokens on the slide at the time, including your own, send them back to their starts.

- When moving past your tokens home with a card, you may enter the home, given that you don't land on one of your tokens, unless at the end of home, and that you don't move too many spaces past home.
    - This isn't true if moving backwards with a 4 or a 10 card.
- To get into the end of your home, you must move your token by exact count. Once there, you may not move that token again.
- If you cannot make any moves, you forfeit your turn. If you drew a 2, draw again.
- A player wins when all of their tokens have made it to the end of the home.

To represent the positions of the tokens on the board, we used a 2D array, the first index being the player, the second the token, and the int value being the position on the board. Since all of the starts and homes are unique spaces for each player, we can represent being at the start as a 0, and being in home with negative numbers, with the end of home being -6. We have a State class which represents the current state of the game, containing the positions of all the tokens, the current deck, how many people are playing the game, and who's current turn it is. The deck is represented as an int array of length 13, with the index of the array being the card number, 0 for sorry cards, and the int value at each index being how many of that card is left in the deck. After 45 cards are drawn, the deck is reset to a final array also housed in the State class. Every turn, a player draws a card and the game engine is run. The engine takes in the card and the current state of the game. It then looks at who drew the card and, using the position of their tokens, calculates every possible move that can be made. It then returns an ArrayList of all the moves. A player is then given this array and can pick what move they want to perform. The chosen move is then played out in the state of the game, updating the position of any necessary token.

Design of Experiment and Results

Now that we have a working version of the game, we needed something to play it. To do this we created an abstract Player class with one method called "play" that would take in the current state of the game and the card that was just drawn and then would return the move to be made. We decided to then first create the simplest and easiest to write AI that would be good as a means to test our MC algorithm. We created a random AI that implements the play method by calling the getMoves method to get an ArrayList of all the possible moves that can be made with the state and card. It then sees whether or not the list is empty, and if not it then picks one move to make at random.

To test our AI, we put four random AI's up against each other. After running a few tens of thousands of games, we noticed that the player who drew first was losing significantly more games than the others, which seemed very odd to us. After another few hours of bug fixes, we were confident that we had the game correct. In a game with four random AI's, each one won around 25% of the games. Now, all we had to do was write a Monte Carlo Tree Search.

When designing our Monte Carlo algorithm, there was one major issue we faced. Sorry! is a game that has a lot of randomness involved in the form of drawing cards. This meant we couldn't write a traditional Monte Carlo search tree. A typical Monte Carlo algorithm works as follows: Define a domain of all the possible inputs. Generate inputs randomly from the domain using the probability of each move. Determine which moves led to wins. Aggregate all of the results. It then picks the best move determined and plays it. The algorithm then runs again after the opponent(s) have made their move.

Our algorithm works following these steps, with a few things done differently. First, we'll define what values each node in our tree holds. First is a state object. Starting at the root is a clone of the original state of the game, and as we move down the tree to each new node, a new state is held originating from the original. Next, we have the amount of time a node has been visited, and the number of times a node was a part of a path that led to a victory. Finally, we have an ArrayList of all the possible moves that can be made at each node. The size of the ArrayList will be equal to the max amount of children a node can have.

Now for how our algorithm works. Like mentioned before, our algorithm works similarly to a typical Monte Carlo Tree Search. First, we define our domain of all possible inputs. This is the list of all possible moves that the AI can make obtained from the getMoves method. Next is where things start to deviate a bit. Since Sorry! is a game with imperfect information, it's hard to pick the best possible move that can be made, since almost all of them have a positive effect. To combat this, our algorithm runs the Monte Carlo algorithm several times, all with different determinizations. This technique lets us make decisions by sampling instances of the game where we do have perfect information. Each determinization has a random seed that then plays out multiple games of Sorry!. As a game progresses, new child nodes are created for each move that can be made. We then have a method to select what child node we should continue to search down. This selection incorporates two factors. One, how many times a move has led to a victory. And two, how many times a move has been fully explored. This helps with the issue of leaving nodes that have maybe only had a couple of games played out, but by chance, both led to a loss. Even though it may be the best move to make, the algorithm would just ignore it because it sees that other moves have higher win rates. Once a move, or child node, is selected, it is then expanded, drawing a new random card, and creating new child nodes based on the moves that can be made. Every iteration, the algorithm checks to see if any path has led to a victory. If so the algorithm then back-propagates up through the path increasing the win rate value in each node by 1.

After a determinization has finished running, it returns an array of all the win rates, or the number of paths that led to a victory, that each child node had, divided by the number of times that child node was visited. Each array of win rates from every determinization is then added together. The algorithm then picks which move had the highest win rate and plays it out. This process repeats until the game is over.

When running the simulation for the first time, there were a couple of things that were notable. The MC AI won more games than the random AI, 7 - 3, and that it took a lot longer to finish running compared to the random AI. When timed, it takes around 38 seconds to finish one game between one random AI and one MCAI. Since we wanted to gain useful information, but also didn't want to wait hours before receiving any meaningful information, we decided to run multiple threads at once. All simulations were run on a Core i5 8600k processor with 6 cores/threads, each running at 4.57 GHz. By using all 6 threads we're able to run the simulations 6 times faster than before.

When testing our MC AI, there are three variables passed into the constructor we can alter that affect how it is run. First, we have c, which changes the UCB value. The UCB, or Upper Confidence Bounds, is a formula used in the node selection process that balances the known win rates of nodes with the number of times they have been explored. The formula used to calculate the UCB was mentioned above in our literature review, where we used how many times a node won as the score. The next variable it takes in is d, which controls how many determinizations are to be run. And the last variable is s, which controls the depth, or the number of iterations, each determinization runs.

Now that we are able to run tests in a decent amount of time, we can begin to experiment the use of our MC AI with different c, d, and s values. We can also predict how long a simulation will take based on how many games are being played, and by multiplying d and s together, as those two values control the number of iterations run. The following table shows our results for two-player games, including games played, wins and losses, and what values were used for c, d, and s. Tests with the same c, d, and s have their number of games played and the wins and losses added together.

Games Played	Win-Loss	c	d	s
1000	720-280	1.4	25	100
100	53-47	0.2	5	100
100	60-40	1.4	10	100
100	67-33	1.4	5	200
100	66-34	1.4	15	200
100	66-34	1.4	50	100
100	71-29	1.4	15	120
100	71-29	1	15	120
100	67-33	0.8	15	120
100	62-38	0.5	15	120
100	68-32	1.2	20	120
100	58-42	1.8	25	100
100	69-31	1.6	25	100
100	65-35	2	25	100
100	71-29	2.4	25	100

We also wanted to test the MC AI against more than just one opponent just to see how that would affect its win rate, but we didn't realize how much longer it would take to finish a single game. Wherewith just two players, one game took around 38 seconds on one thread, one game with three players took a few minutes on one thread. Because of this, we were only able to test a couple of variations of the MC AI. Due to time constraints, and taking even longer for four-player games, we didn't test the MC AI against three opponents. Below is our results for the three-player games. The wins between the two random AI are added together.

Games Played	Win-Loss	c	d	s
100	53-47	1.4	25	100
70	34-36	1.4	25	120

Analysis of Results

While testing we needed to create a baseline for what values to use as our c, d, and s. From our literature review and other readings, we knew that the theoretical best value for c to calculate the UCB value was $\sqrt{2}$, so we decided to use 1.4 as a close estimate. Similarly, from our literature review, we found that as you increase the number of determinations ran, the value for each determinization added diminishes as you reach higher numbers, so we chose 25 as our d. And finally, for our s value. From the way our MC AI was coded, we couldn't have a smaller s value than the maximum amount of moves that can be made in a single turn. This number came out to be 76 and happened when a 7 card was drawn with all four of your tokens in play. So, we decided to use 100 as our base. Mathematically, we can expect a higher win rate when using higher d and s values, but with the simulation taking much longer as each value increases. After running 1000 games with our baseline values, we see that our MC AI wins around 72% of the time. We believe this to be substantially better than the random AI, especially in a game that leaves a lot up to chance. Now that we have a baseline, we can look at what happens using different values.

First, we'll start with what we found using different c values. Choosing to lower the c affects the UCB formula to value nodes that have a lower exploration factor more. Using a higher c affects the UCB formula to value nodes that have a higher win rate more. Keeping the other factors constant, we see that using a lower c value has an on average lower win rate compared to 1.4. Using a higher c value still has an on average lower win rate, but not as bad as decreasing the value. When tested using a lower value for d, we can see there is an on average lower win rate, which is to be expected. The same thing applies to a lower value for s. The tradeoff, however, was a drastically increased runtime. Increasing the value of s has similar win rates to when it's 100. Overall, we see that our MC AI has over a 50% win rate in every case, and over a 60% win rate in most cases. When looking at the tests ran for three-player games, we see that the MC AI has above a 40% win rate, however this is in significantly less tests than two-player games, so further testing is needed to confirm our results, however we do suspect the MC AI to continue to out perform random AI in all cases. Overall, we see that our MC AI is significantly better than a random AI in all cases. The challenge we have to try and figure out is what values for c, d, and s to use.

Conclusion and Future Considerations

In conclusion, the problem we took on was to test if a Monte Carlo Tree Search AI could reliably win in a mostly random game. Based on our research a Monte Carlo Tree Search AI will reliably win in strategic games like chess and go, so we thought it would be interesting to test on a game that doesn't rely on that much skill. Thus the game we picked was Sorry! and simulated a multitude of games played between a Monte Carlo Tree Search AI and one to three random move AI's. The results were varied based on the parameters c, d, and s, but when these parameters were optimized the win rate was about 70 percent for the Monte Carlo Tree Search AI. This result was surprisingly great for what is considered to be mostly a random game of luck. Further research and statistical tests will need to be conducted to collect a larger database of simulation results and prove statistical significance of results. We did not have the hardware to efficiently run a substantial amount of these simulations because they take a lot of computational power and time, so if we were to continue with this research upgrading our simulation hardware would be a necessity. Also, because we didn't have the capacity to collect enough data, we did not run any statistical tests which is also something we would need to further this research.

References

[1] C. B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: `10.1109/TCIAIG.2012.2186810`.

[2] Clayton Burger, Mathys C. du Plessis, and Charmain B. Cilliers. "Design and Parametric Considerations for Artificial Neural Network Pruning in UCT Game Playing". In: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. SAICSIT '13. East London, South Africa: Association for Computing Machinery, 2013, pp. 209–217. ISBN: 9781450321129. DOI: `10.1145/2513456.2513477`. URL: `https://doi.org/10.1145/2513456.2513477`.

[3] Adrien Couëtoux et al. "Continuous Upper Confidence Trees". In: *Learning and Intelligent Optimization*. Ed. by Carlos A. Coello Coello. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 433–445.

[4] S. Gelly et al. "Modification of UCT with Patterns in Monte-Carlo Go". In: (2006).

[5] Sylvain Gelly and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go". In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1875. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/j.artint.2011.03.007`. URL: `https://www.sciencedirect.com/science/article/pii/S000437021100052X`.

[6] S. He et al. "Game Player Strategy Pattern Recognition and How UCT Algorithms Apply Pre-knowledge of Player's Strategy to Improve Opponent AI". In: *2008 International Conference on Computational Intelligence for Modelling Control Automation*. 2008, pp. 1177–1181. DOI: `10.1109/CIMCA.2008.82`.

[7] J. Huang et al. "Pruning in UCT Algorithm". In: *2010 International Conference on Technologies and Applications of Artificial Intelligence*. 2010, pp. 177–181. DOI: `10.1109/TAAI.2010.38`.

[8] Thomas Keller and Patrick Eyerich. "PROST: Probabilistic Planning Based on UCT". In: 22 (May 2012). URL: `https://ojs.aaai.org/index.php/ICAPS/article/view/13518`.

[9] JAM Nijssen and Mark HM Winands. "Enhancements for Multi-Player Monte-Carlo Tree Search. Computers and Games (CG 2010)(eds. HJ van den Herik, H. Iida, and A. Plaat), Vol. 6515 of LNCS". In: Springer-Verlag, Heidelberg, Germany.[10], 2011.

[10] A. Ouessai, M. Salem, and A. M. Mora. "Improving the Performance of MCTS-Based µRTS Agents Through Move Pruning". In: *2020 IEEE Conference on Games (CoG)*. 2020, pp. 708–715. DOI: `10.1109/CoG47356.2020.9231715`.

[11] Edward J Powley, Daniel Whitehouse, and Peter I Cowling. "Determinization in Monte-Carlo tree search for the card game Dou Di Zhu". In: *Proc. Artif. Intell. Simul. Behav* (2011), pp. 17–24.

[12] Jan Schafer. *The UCT Algorithm Applied to Games with Imperfect Information.* 2008.

[13] Y. Wang and S. Gelly. "Modifications of UCT and sequence-like simulations for Monte-Carlo Go". In: *2007 IEEE Symposium on Computational Intelligence and Games*. 2007, pp. 175–182. DOI: `10.1109/CIG.2007.368095`.

1 Contributions

Cole contributed by coding and running the simulation, background research, writing part of the background, approach to solution, representation, and algorithms, design of experiments and results, and analysis of results.

Tom contributed by background research, writing the abstract, problem description, part of the background, and the conclusion and future considerations.