NERS 544 Final Project Report: Monte Carlo Tree Search Applied to Simple "Filler" Game

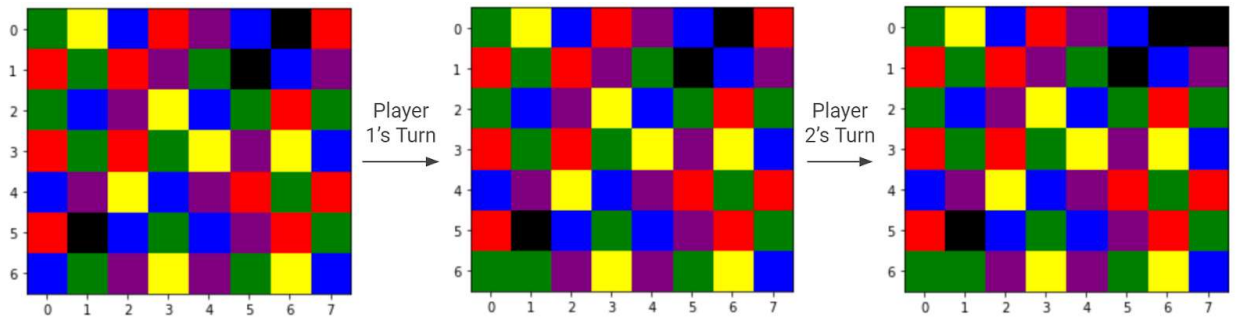
Sumit Basak, Arham Jain, Daniel Manwiller

April 22, 2021

In this project, we applied the Monte Carlo Tree Search (MCTS) algorithm to create an AI for a simple game called Filler. MCTS is a heuristic decision algorithm that uses Monte Carlo random sampling to generate random playouts of sampled game states to build a weighted decision tree. The game of Filler is a relatively simple game, yet it provides a reasonable and even scalable complexity for an MCTS based AI. Details of the game, the MCTS algorithm, our implementation, evaluation of our AI, and our GUI one-player interface are provided in the following sections. All relevant code for our implementation, evaluation, and GUI game player can be found in our GitHub repository [1].

Filler Game:

This game consists of a rectangular board with tiles of different colors as shown in Figure 1. In its standard form, the board size is a 7x8 tiled grid and has 6 distinct colors. This is a 2-player game and the objective of the game is to capture more tiles than your opponent. Player 1 starts with the tile in the bottom left corner and Player 2 starts with the tile in the upper right corner. The players alternate turns and on each turn, the player chooses to change his color to one of the four remaining colors not already occupied by either the player or his opponent on that turn. Each player captures new tiles by changing his color to be the same color as any tiles touching the tiles he has already captured. When a player changes his color, all his captured tiles change color as well. Since neither player is ever able to choose the same color as his opponent, a tile can only be captured once. The strategy in this game is to strategically pick your color to maximize your captured tiles not only in the current move but also in future moves by boxing off regions or expanding toward regions of same-colored tiles. A player might also strategically choose his color to keep his opponent from choosing that color and delay his opponent from capturing a large number of tiles.



a) Initial game board setup b) Player 1 changes his color to green c) Player 2 changes his color to black

Figure 1: Shows the initial game board and one round of gameplay. Note that Player 2 chooses black to capture the tile which sets him up for choosing blue next turn and capturing two tiles. Choosing black also blocks Player 1 from choosing it on his next turn and setting him up on the following turn to capture two purple tiles.

This game provides a simple setup and gameplay while also having a relatively complex strategy given the need to look many turns in the future to determine the optimal move. The complexity of the game can be further increased by increasing the board size from the standard 7x8 grid to be as large as desired. Even with a standard board size, there would already be over a billion possible game states to analyze. The ability to scale the board size to be much larger provides an even more interesting application for the MCTS algorithm where an exhaustive decision tree or other simple decision tree methods would be even less feasible.

MCTS Algorithm:

The MCTS algorithm provides a heuristic search algorithm for some defined decision process and has been applied to many board games such as checkers, chess, and most famously, the game of Go [2]. While there are many different decision tree search algorithms, MCTS leverages the power of random Monte Carlo sampling to efficiently play out simulated games to the end many times to build a weighted decision tree. On each run, a game state is chosen based on the current game state and the outcomes of previous runs. Once a game state is selected, a game is played out with random choices. The decision tree is then updated based on the outcome of each iteration. The algorithm can then be run with a specified number of iterations before the decision with the highest weight is selected. The MCTS algorithm can be broken down into 4 distinct phases as outlined in Figure 2.

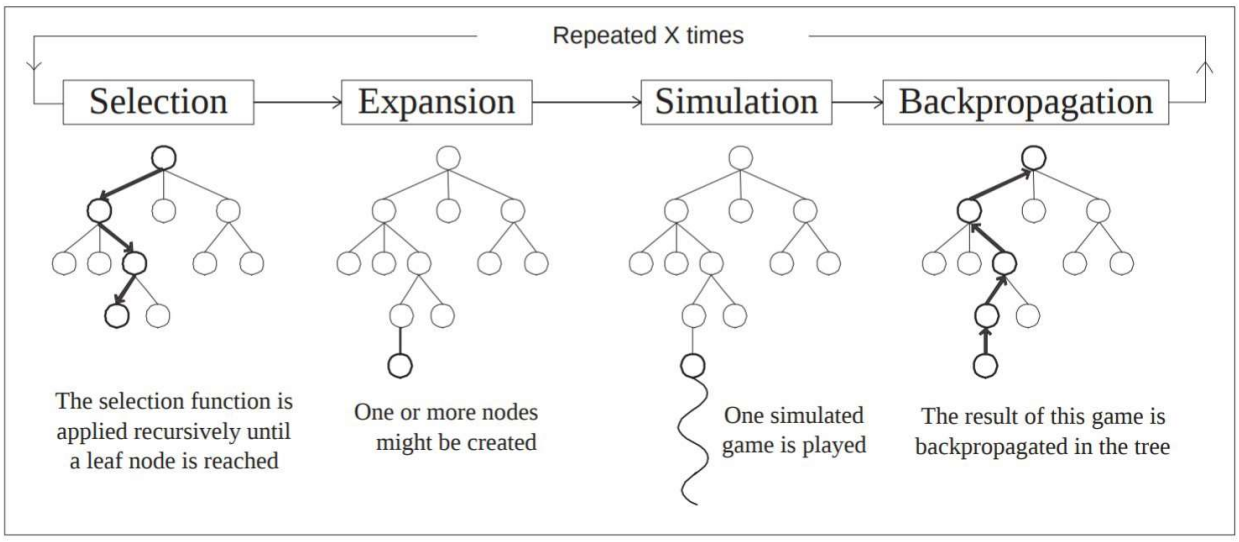


Figure 2: MCTS Algorithm. [3]

Phase 1: Selection

During this initial phase, the game state to explore, by random play-out till a winner is determined, is chosen by traversing the decision tree of previously explored states, where each explored sequential game state is represented as a weighted node. The standard node weighting incorporates the number of wins from previous trials passing through that node and the number of total trials passing through that node. This allows the algorithm to have a tunable parameter for how often it expands on more promising game paths to explore vs. exploring less tested outcomes. The selection for each iteration is based on traversing

down the tree until a leaf node is reached by always choosing the child node with the highest weight. This selection process allows the exploration of game states to favor those that are more likely to be successful. This node weighting and selection essentially functions as a variance reduction method since given a properly tuned exploration parameter and sufficient iterations, the MCTS result should converge to the same answer as an exhaustive tree search but with far less computation. Equation 1 gives the equation for standard node weighting.

$$\frac{w_i}{s_i} + c\sqrt{\frac{ln(s_p)}{s_i}}$$

- w_i : this node's number of simulations that resulted in a win
- s_i : this node's total number of simulations
- s_p : parent node's total number of simulations
- c : exploration parameter

Equation 1: Standard node weighting equation. [4]

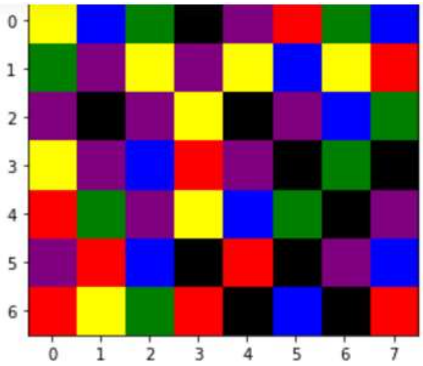
For our algorithm, we modified the node weighting equation slightly to potentially give a further variance reduction. The standard node weighting only considers the percentage of wins of subsequent game states and not by how much it won. This can be helpful in certain applications where it is hard to quantify how much it won by or in games where the amount it won by does not directly influence the chances of a game path being more likely a winning path. However, for the game of Filler, while the only objective is to win in the end, the amount it won by in a given random play-out could be helpful in determining if that is a likely path to victory. So we added a factor to our node weighting equation to account for the average score of a given node's simulations. Equation 2 expresses the node weighting with this added factor.

$$\frac{w_i + r_i}{s_i} + c\sqrt{\frac{ln(s_p)}{s_i}}$$

- w_i : this node's number of simulations that resulted in a win
- r_i : this node's sum of scores of its simulations
- s_i : this node's total number of simulations
- s_p : parent node's total number of simulations
- c : exploration parameter

Equation 2: Our adapted node weighting equation.

We also further modified the selection process to have a factor that accounts for the intelligence of the opponent. The purpose of this intelligence factor is to avoid frequently selecting a game path that appears to be really good but also has a high probability of being blocked by the opponent choosing a color that blocks the AI from taking that path. So, we implemented an intelligence factor as a number between 0 and 1 which corresponds to the probability that a player does a randomly selected move, or an optimal greedy move at some depth. Figure 3 shows an example of how important this factor can be. It is also important to note that setting this factor too high can cause the AI to overfit to the opponent making a certain choice. If the opponent then makes a different choice, the AI's strategy will be completely messed up. Thus, from experimenting with this parameter, we found that a value of around 0.5 gave the best trade-off of weighting what the opponent is more likely to do while not overfitting to a specific outcome.



- Intelligence Parameter = 0.0: AI as player 1 chooses purple.
- Intelligence Parameter = 0.5: AI as player 1 chooses yellow.

Figure 3: Example game board. The optimal greedy strategy for player 1 would be to select purple and then red on the next move, and on from there. However, the opponent's optimal greedy strategy would be to select red, and then green. If player 2 selects red after player 1 selects purple, it blocks player 1 from selecting red on the next turn. Thus, factoring for the intelligence of the opponent is vital in avoiding these situations.

Phase 2: Expansion
This is a fairly straightforward step in which we add all possible children to the leaf node chosen in the selection phase. This equates to selecting a specific game state in the selection phase and then adding all legal moves to this chosen leaf node as new children. These nodes represent specific game states and are what is used in the simulation phase.

Phase 3: Simulation
During this phase, the added game states from the expansion phase are then simulated till the end of the game using random moves by each player. No part of this simulation is stored in the search tree. Only the result of winning or losing and the final score matters for updating the tree in the backpropagation phase.

Phase 4: Backpropagation
The result from each simulation is then backpropagated through the decision tree to update all relevant nodes affected by this result. This involves updating the weights on all nodes that are direct ancestors of the node added during the expansion phase as well as that new node itself. These new weights are then used to select a path to explore in the selection phase during the next iteration.

The MCTS algorithm is run for a set number of iterations and the move with the highest likelihood of success, as determined by the Monte Carlo simulations, is then chosen as the AI's next move. This algorithm provides an efficient way to simulate game outcomes and determine an approximately best move for games where an exhaustive search or other naive decision tree search algorithms aren't feasible or effective. For the game of Filler, an exhaustive decision tree search could possibly be feasible for the standard board setup, although it would require quite a lot of computation time to analyze over a billion possible game states. Other simple decision tree algorithms such as a heuristic minimax algorithm might be a simpler approach to take. But MCTS provides a powerful yet fairly straightforward algorithm for simulating many random play outs of different game states to build a weighted decision tree to find an approximately optimal move in relatively few iterations using node weighting to allow for a form of variance reduction.

Implementation:

We implemented the Filler gameplay and the MCTS algorithm completely from scratch using python code, which is available in our GitHub repository [1]. This involved writing the code to set up the board, update the board based on the chosen move for a given player, and properly display the board at each turn. This also involved writing the code for the MCTS algorithm with its 4 phases and running the algorithm for a given number of iterations to find the best move. The standard MCTS algorithm was actually fairly simple to implement. It required an MCTS class with functions for each of the four phases of the algorithm as well as a function to run through the algorithm with the given number of iterations. The code still took some work to make sure it was working properly but in the end, the implementation was fairly straightforward. However, while the actual MCTS code was fairly straightforward, the code for the Filler gameplay was fairly complex, especially the functionality to find the best greedy move at any given depth. This functionality was required for our extensions to the standard MCTS algorithm like the addition of an intelligence parameter.

We also spent a significant amount of time optimizing our code to run as fast as possible. Python code is typically slower than compiled languages such as C++ but it is often much simpler to implement complex data structures or prototype algorithms in python. Also, if built-in python structures such as lists and sets are used, python code can still be relatively efficient. On each move selection, our AI would spend the most time in the `update_board()` function, which updates the game board with a selected move for a given player. It spends so much time simply updating the board because the MCTS algorithm is based on random playouts of games where a copy of the game board is created and updated with random moves until someone wins. The board updating process is not simple either because it needs to check all neighboring cells of a player's current captured territory and determine which cells it has now captured. This checking over all neighboring cells is not trivial since many of the cells are on the edge of the board and naively checking every neighbor adds lots of unnecessary computation. We were able to speed up this function significantly by using python sets to only add a neighbor to be checked once instead of multiple times if it is bordered by multiple territory cells and to avoid checking any cells that are already captured by either player. We also looked into integrating a compiled version of this function into our code but the functionality in either python compilation tools such as Numba [5] or using a compiled wrapper of Nim code did not appear viable or well suited to our code. We were able to significantly improve our computational efficiency by using built-in python structures like lists and sets wherever possible in our code, and we were able to get our code to run a sufficient MCTS search to determine an approximately optimal move on the order of just several seconds.

We also experimented with parallel processing of our Monte Carlo Tree Search, where we could set some number of MCTS algorithms to run in parallel and then aggregate their results to select the best move. After implementing this parallel processing with the python Multiprocessing library, we tried running the MCTS move selection with 11 python processes running concurrently. This was the number of CPU cores minus one on the laptop running this test. However, with 1000 iterations, this either consistently gave the same move selection every time or chose between two best moves equally often, essentially negating the purpose of aggregating parallel processes. Also, even with the multiprocessing, it still took several times longer to run the 11 processes than it took to run a normal python process. It also took about 50% longer to run with just one process but using the multiprocessing framework compared to running just a normal python process, confirming there is some significant startup time for a multiprocessing approach. This

lead to the conclusion that while multiprocessing could be useful for similar applications and is certainly useful in the field of Monte Carlo in general, it likely would not be useful for our application where a consistently chosen best move can be found in close to the time it would take to start up the multiprocessing framework. We were, however, able to make very good use of python multiprocessing for running our MCTS simulations to compare the best combination of the number of iterations, exploration parameter, and intelligence parameter.

We decided to write our GUI application in Nim, due to one member's familiarity with the language. The GUI is simply a web application that serves an HTML page that can then be opened in the browser and interacted with. See the figure below for a screenshot of the interface.

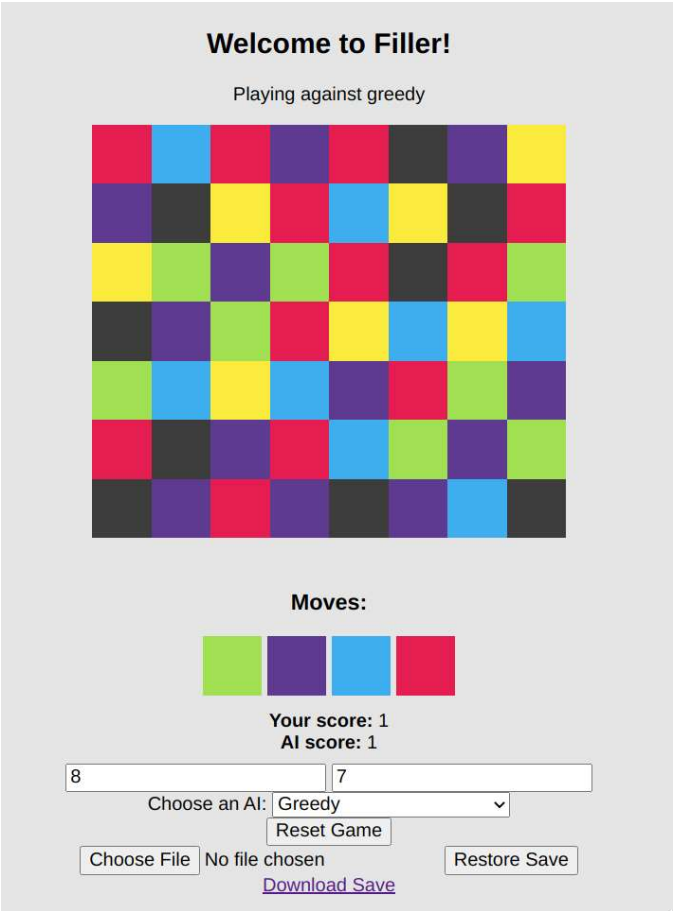


Figure 4: Our Filler GUI interface to play against our AI

In order to interop with the Python code that was running the actual AI and the board, we used a library called Nimpy. The web server code also relies on two packages: Jester to handle the web serving part, and Karax for the HTML templating engine. This GUI parallels the actual Filler game on iOS as closely as possible, where clicking on a square color changes your color appropriately. Then the AI moves and the updated board is shown. We also provided options to change the size of the board, and the type of AI that you play against. We found that it usually took longer when playing against the MCTS version of the AI,

so we limited the number of iterations to 100 in an attempt to make sure that the GUI didn't freeze up unexpectedly, or timeout in the web browser.

One interesting aspect of the GUI is the ability to save and load a board state. This was invaluable when playing against the two different AIs, as the initial board can make a lot of difference in which moves should be used. Downloading the save file uses the Python pickle functionality to dump the board object from memory, then the Nim code serializes it into a text file for download. Restoring a save file uploads a text file that is then parsed using Pickle and reconstructs the board object.

While this GUI is probably a bit overkill for our needs, we found implementing it to be a lot of fun and were happy with the result. It also made it easier to test out certain things with reproducibility.

Evaluation:

To show how well our MCTS based AI was working and to experiment with different model parameters such as the number of iterations or the exploration and intelligence parameters, we tested our AI against a greedy model that selects a move based on an optimally greedy path with some given move depth. This greedy move doesn't look ahead very well if its depth is set fairly low and it cannot account for what the other player does to block it from making certain moves or cut it off following a certain path. So, a greedy model is definitely not an optimal strategy but it is a good way to test how well our AI can perform and to find its optimal set of model parameters.

We ran simulated games of our AI vs. the greedy model with 50, 100, 500, 1,000, 2,000, and 5,000 iterations of the MCTS algorithm. We ran sets of 100 games for each iteration value and recorded the AI's win percentage and average score. Ties are also relatively common occurrences with the standard 7x8 board size but with the object being to win the game, we focus only on win percentage. We ran this simulation with the AI going first and then again with the AI going second because with such a small game board, going first in the game of Filler is a significant advantage. We also ran with a greedy move depth of 1 and with a greedy move depth of 3 to show how well the model can perform against different levels of opponent difficulties. We ran all of this with exploration parameter values of 1, 3, and 5, and with intelligence parameter values of 0.0, 0.25, 0.5, 0.75, and 1.0.

To trim down the total number of simulation variations, we only considered an intelligence parameter of 0.5 when running other simulations since while higher values for this parameter led to better performance against the greedy model, it was actually overfitting to the greedy method and didn't perform as well when tested against a human player. We also ran individual simulations with the best model parameters to test the outcome of playing with a larger board size and to test our AI's performance with and without our adaptation to consider the average score in the Monte Carlo simulations. The figures given below show our results from running these simulations broken into the different parameters or model variations investigated. We show only a representative subset of the total simulation results to best showcase the conclusions about the different model parameters. The figure captions point out some important observations but any further discussion is left to the discussion section of this report.

Number of Iterations:

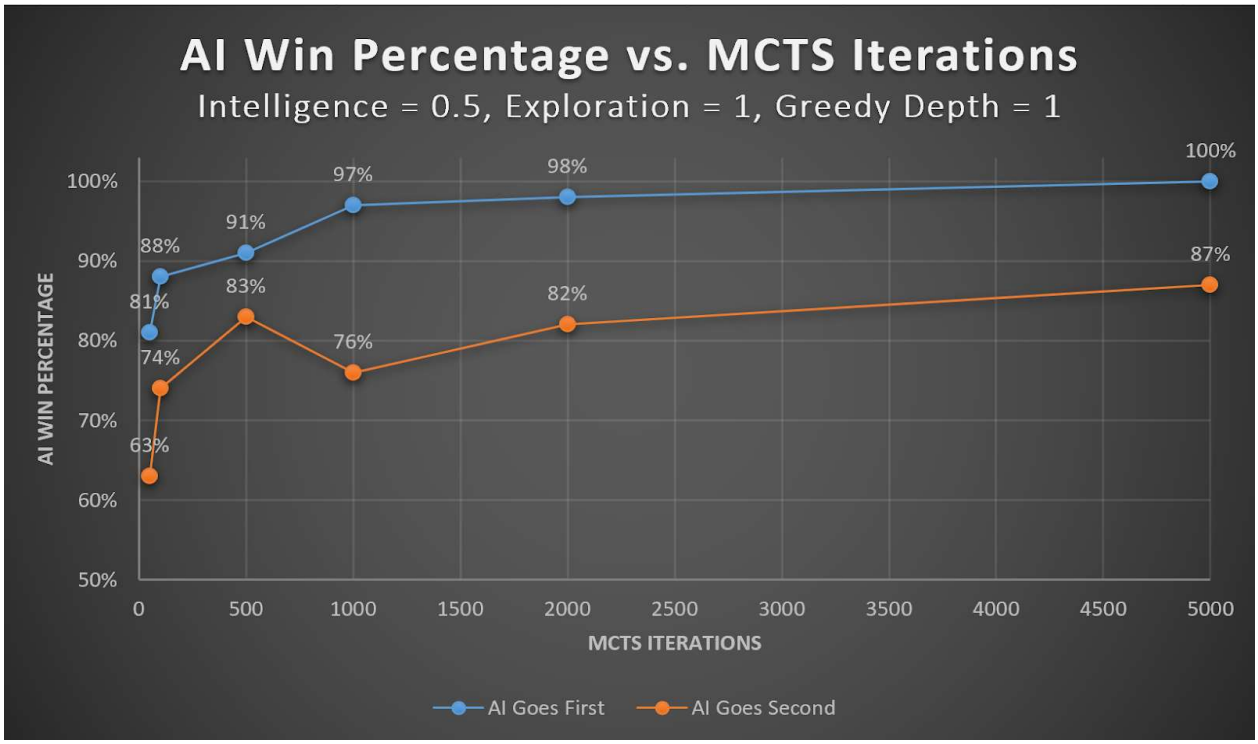


Figure 5: This showcases the AI performance against the greedy model with a greedy depth of 1. Note that the performance plateaus after about 1,000 or 2,000 iterations.

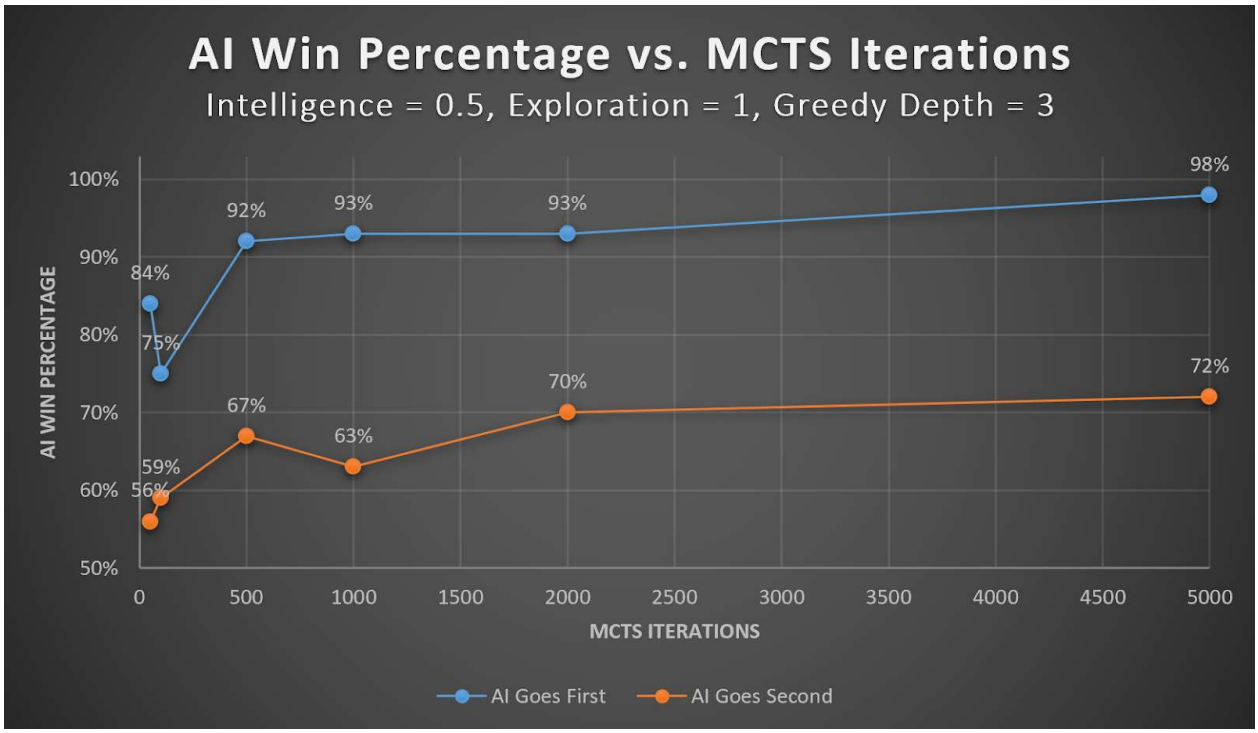


Figure 6: Again showcasing the AI performance against the greedy model but now with a greedy depth of 3. This also shows the performance plateaus but this time just after about 500 iterations.

Exploration Parameter:

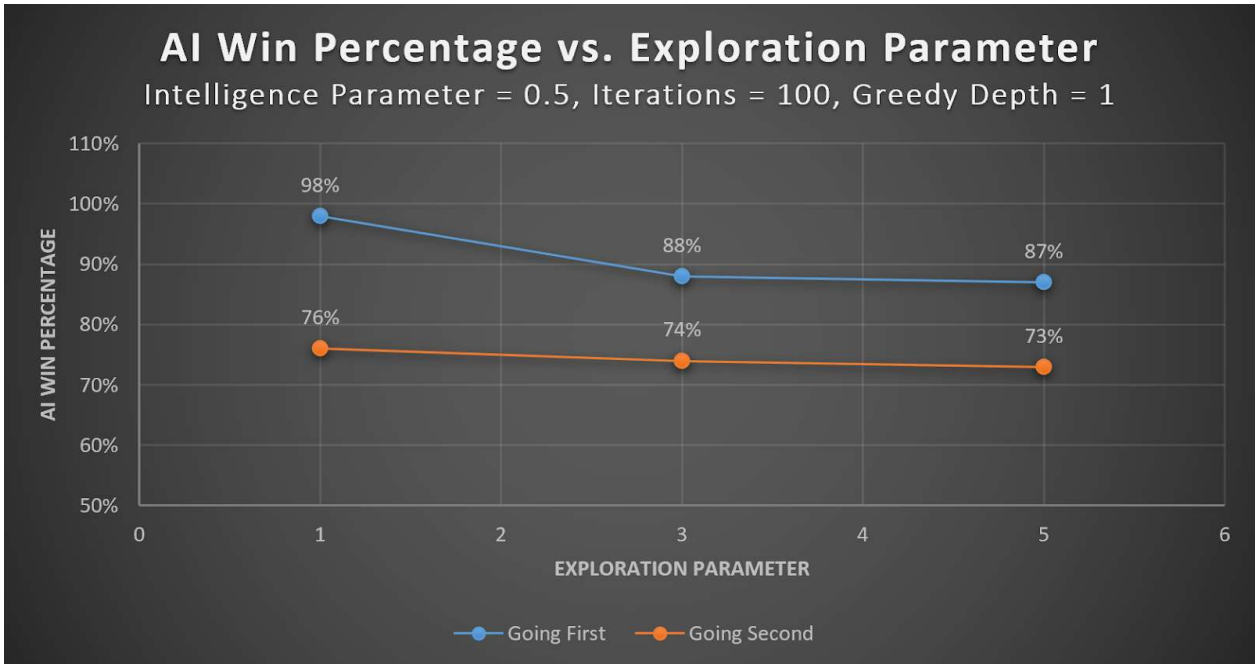


Figure 7: Clearly shows that a larger exploration parameter leads to less variance reduction and worse AI model performance. This was less pronounced with more iterations but was still consistently worse with a larger exploration parameter especially for the AI going second.

Intelligence Parameter:

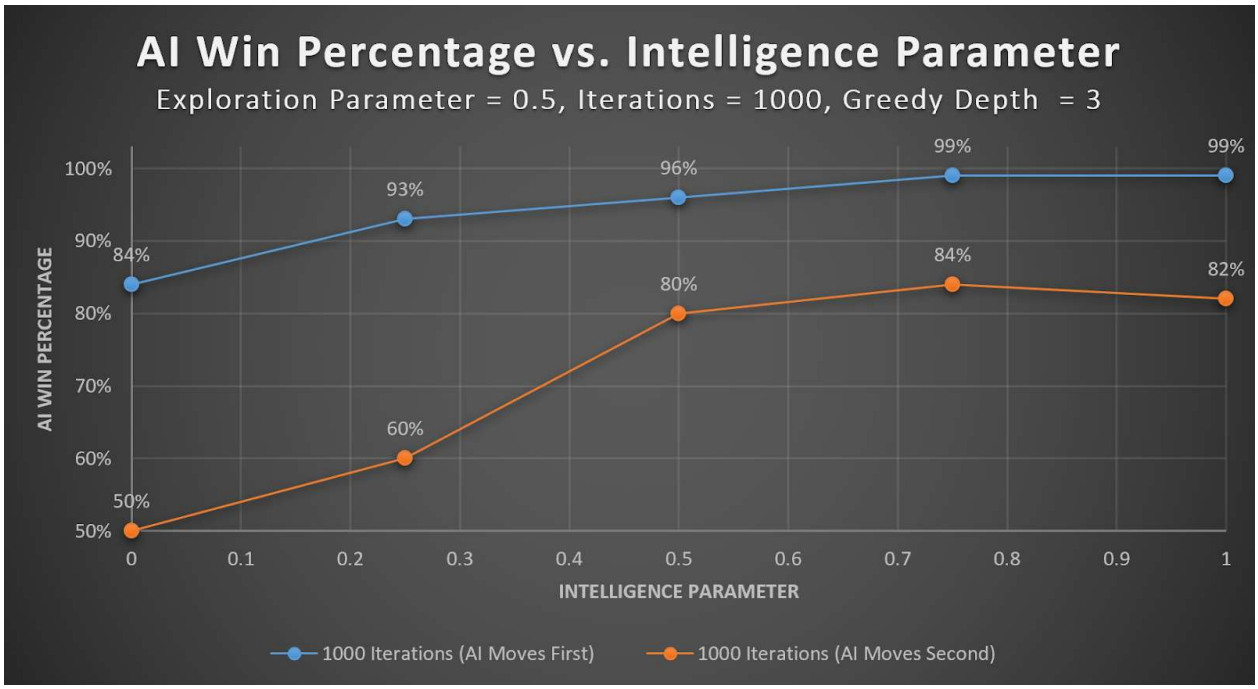


Figure 8: Shows how the addition of the intelligence parameter significantly improves the AI win percentage. Note there is still a point at which this overfits to the greedy method and doesn't actually improve the AI's performance against a human player.

Board Size:

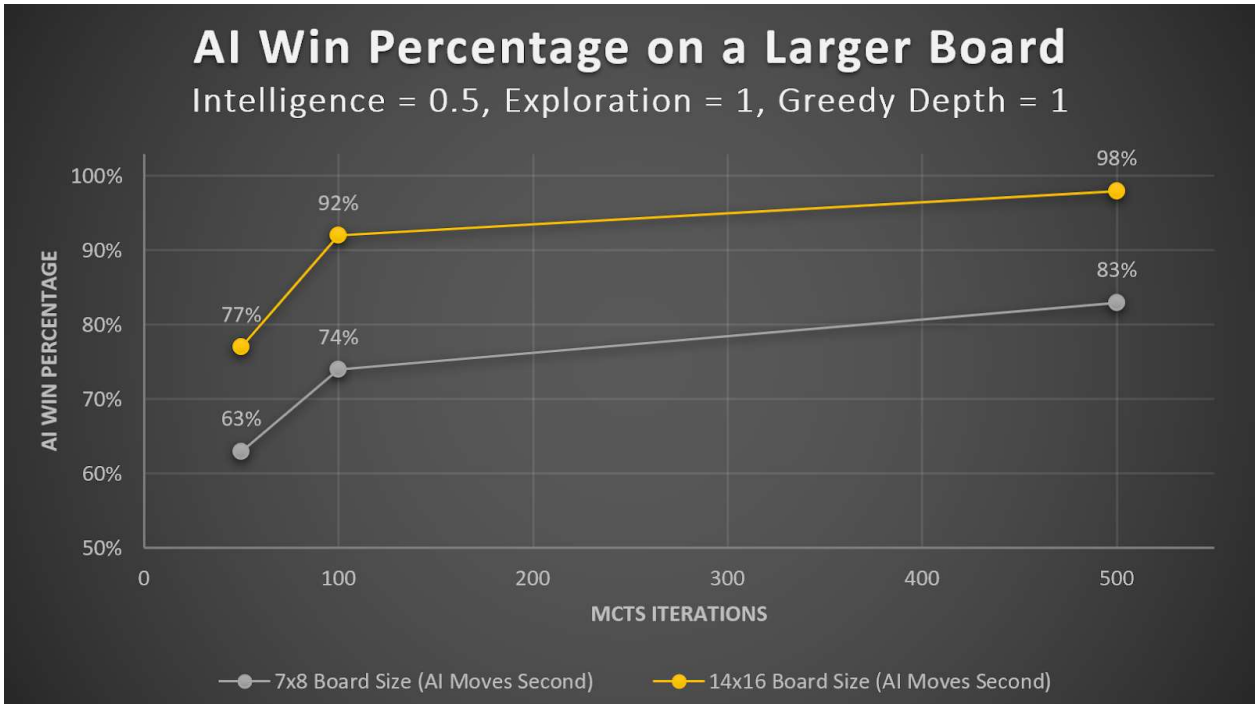


Figure 9: Confirms that the AI can perform much better on a larger board where moving first is less of an advantage.

Average Game Score Consideration:

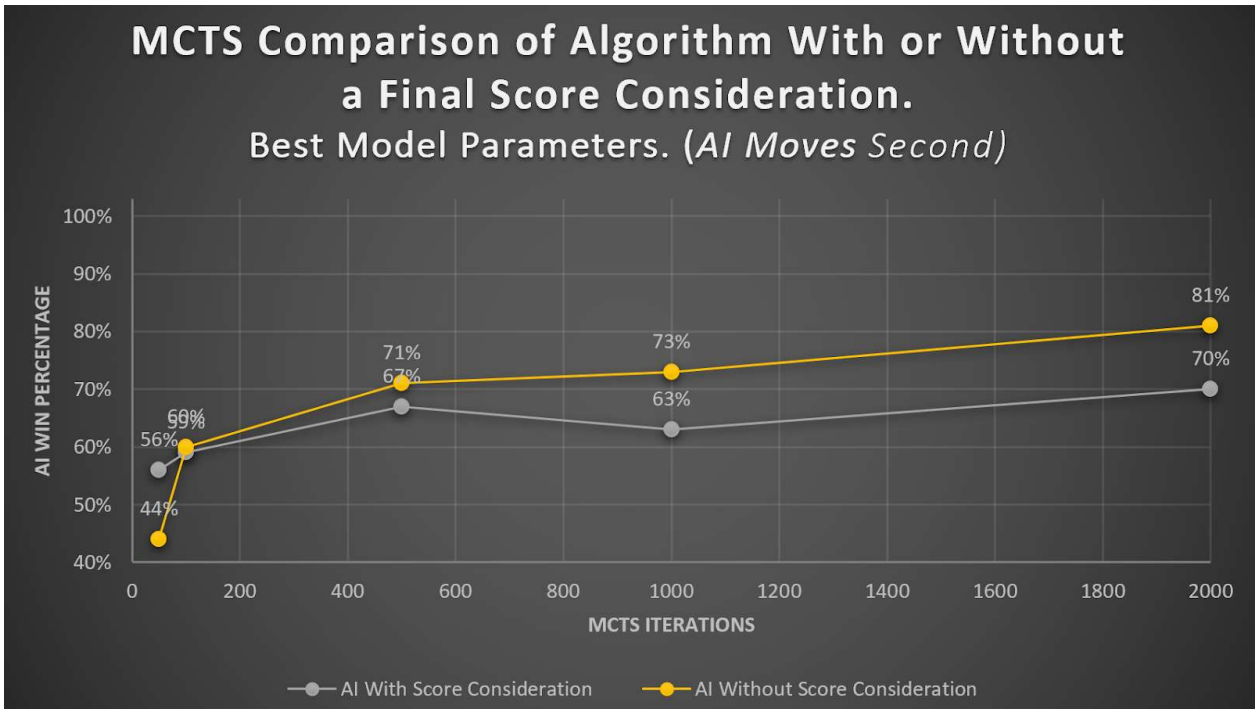


Figure 10: Interestingly, this appears to contradict the hypothesis that adding a consideration of the final game score would improve the AI's win percentage.

AI vs. Human:

	AI Moves First						AI Moves Second				
Player #	Win	Loss	Tie	AI Score	Human Score		Win	Loss	Tie	AI Score	Human Score
1	✓	-	-	30	26		-	✓	-	26	30
2	✓	-	-	31	25		✓	-	-	30	26
3	✓	-	-	33	23		-	✓	-	23	33
4	✓	-	-	29	27		-	✓	-	27	29
5	✓	-	-	31	25		-	✓	-	25	31
6	✓	-	-	32	24		-	-	✓	28	28
7	✓	-	-	31	25		✓	-	-	32	24
8	✓	-	-	29	27		✓	-	-	29	27
9	✓	-	-	30	26		-	✓	-	36	20
10	✓	-	-	31	25		✓	-	-	30	26

Table 1: Results from our AI vs. random people who were familiar with the game of Filler but certainly not expert players. These results were generated by sending an actual GamePigeon filler game, then inputting the data into our AI model to determine each move. This was tested with a parameter selection of exploration = 1, intelligence = 0.5, and iterations = 1,000.

Discussion:

The results shown in figures 5 and 6 clearly demonstrate that our AI can win almost every time against the greedy algorithm given when it goes first and enough iterations of the MCTS algorithm. The results also show the significant difference in win percentage between going first and second. However, the AI is still able to win a significant majority of the time even when going second. This holds true for both greedy depths tested although the win percentages overall were slightly lower with a greedy depth of 3 compared to that with a depth of 1. The win percentages appear to plateau after a certain number of iterations with only very minor improvements in performance with ever-larger numbers of iterations. We selected 1,000 iterations for our final model to test against human players since more iterations requires longer computation times. With 1,000 iterations of the MCTS algorithm per move, the AI was able to select an approximately best move on the order of several seconds, although this is variable depending on the computer hardware.

Figure 7 clearly shows that a larger exploration parameter leads to less variance reduction and worse AI model performance. This was less pronounced with more iterations but was still consistently worse with a larger exploration parameter, especially for the AI going second. We also tested with exploration parameters with values less than 1 but for any value less than 1, the model had the tendency to latch on to a certain move and never explore other paths, causing the AI to choose random moves and never fill in an enclosed but not captured territory. With the addition of the game score consideration, this did not necessarily occur right at the boundary of the parameter being equal to 1 but anything less than 1 would often end up with a game in a group of simulated games where this would happen and the simulation wouldn't complete because the AI was just alternating between unnecessary moves. So, for our final model we chose to use an exploration parameter of 1 as this provided the best variance reduction while not overfocusing on one path and causing undefined behavior.

Figure 8 shows the results from testing different intelligence parameters. An intelligence parameter of 0.0 corresponds to the standard MCTS algorithm but as the parameter increases, we can see how much better the AI performs against the greedy algorithm. This very clearly shows the importance of this parameter for again providing another form of variance reduction using information about what the opponent is likely to do at each move. We must still note that there is still a point at which this overfits to the greedy method and doesn't actually improve the AI's performance against a human player. So, after testing playing against the AI with various intelligence parameters, we chose a value of 0.5 for our final model.

The results in Figure 9 confirm that the AI can perform much better on a larger board where moving first is less of an advantage. With such a small standard size board, the game is often determined simply by who goes first or at least by who the layout of the board tends to favor. As such, there is a limit on how much skill can generate better results at this game. However, with a larger board size, there is much more skill in looking further moves ahead and the game is much less determined by who goes first. Our AI can never maintain a 100% win percentage with the standard board size because any random board may overwhelmingly favor one player or the other, but as the board size increases, the chances that our AI can guarantee a victory increase as well.

Figure 10 shows a very interesting result. It appears that the AI win percentage is actually improved by not considering the average game score in the MCTS selection process. The hypothesis had been that while other applications of the MCTS algorithm were not suitable for such an addition to the algorithm's formulation, for the game of Filler, the win percentage is very closely linked to the final score and the average final score could help to focus the algorithm's search on more likely paths. This again would be a form of variance reduction of the Monte Carlo Tree Search. However, it appears from the data in Figure 10 that the addition of the average game score consideration actually slightly decreased the AI's win percentage. These were not the expected results based on our hypothesis but the results appear to clearly support the original formulation of the MCTS algorithm without any consideration of the average game score. Our final model tested against human players still incorporated the game score consideration partly because we started testing against human players before the results in Figure 10 were finalized and partly because the consideration of game score made the AI play a little more like a human player towards the end of the game where it would still make rational moves to increase its territory even when the outcome of the game was almost certainly determined. However, the results in Figure 10 are interesting in that they demonstrate that the original formulation of the MCTS algorithm to only consider the win/loss outcome

of the Monte Carlo simulations appears to be the better formulation. We would like to point out that AlphaGo, a well-known Go AI that beat the Go world champion, assigned a deep-learning based heuristic score to each possible move. This had the effect of variance reduction as the number of possible moves required to analyze could be decreased using this heuristic. Our attempt to include the average final game score into the MCTS algorithm as an additional heuristic was meant to have the same effect of reducing variance and increasing performance. While this appears not to have been a helpful heuristic for this game, we still believe this idea of an additional heuristic could be helpful for our MCTS based AI.

Table 1 shows the results of our AI vs. human players. These were random people who were familiar with the game of Filler but certainly not expert players. These results were generated by sending an actual GamePigeon filler game, then inputting the data into our AI model to determine each move. This was tested with a parameter selection of exploration = 1, intelligence = 0.5, and iterations = 1,000. The AI won every time when it went first and won a fair number of times when it went second. If both players play perfectly, the outcome is almost completely determined by who gets to go first, as is true in other games like tic tac toe or connect four. However, because of the random nature of the board set up, it may favor one or the other player enough to allow a player who goes second to still win. Thus, our AI wouldn't be able to keep up its perfect streak when going first but for the 10 such games tested against human players, it won every one. These results also showed that if the human player made just one suboptimal move, the AI was often able to pull out a win even when it went second in the game. The results in Table 1 certainly validate that our AI plays very well and with just 1,000 iterations of the MCTS algorithm it can find an approximately optimal move. This compared to the over a billion possible game states required to check what the truly optimal move would be.

Conclusion:

This was a rather fun and interesting project in which we implemented from scratch a Monte Carlo Tree Search based AI to play the game of Filler. We also tested several adaptations to the algorithm to introduce further variance reduction in the Monte Carlo simulations and ran many different simulations to evaluate the best combination of model parameters. We found that the introduction of an intelligence parameter to weight the likely choice of the opponent significantly improved our AI's win percentage against a greedy algorithm and against human opponents but only up to a certain threshold before overfitting to the greedy method. We found that the adaptation to the MCTS algorithm to consider the average game score in the Monte Carlo simulations did allow the AI to play seemingly more rationally but actually appeared to decrease its overall win percentage. We additionally ran simulations to determine the optimal exploration parameter and number of iterations of the MCTS algorithm. We also spent a significant amount of time optimizing our code, testing with parallel processing, and building a GUI interface to play against our AI. Altogether, this project provided a very interesting way to test Monte Carlo methods such as variance reduction and parameter selection. Our introduction of the intelligence parameter appears to have been very helpful and we believe this could be beneficial to other MCTS applications. We were able to show that our MCTS based AI works quite well and our code and instructions for running our one-player GUI are available for anyone who would like to test it out on their own [1].

References:

[1] “Project GitHub Repository”, dmanwill, 2021. [Online]. Available: https://github.com/dmanwill/Monte-Carlo-Tree-Search-Filler-AI. [Accessed: 11- Apr- 2021].

[2] "AlphaGo: The story so far", Deepmind, 2021. [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far. [Accessed: 23- Mar- 2021].

[3] Guillaume Chaslot et al. “Monte-Carlo Tree Search: A New Framework for Game AI.” In:AIIDE.2008 (cit. on p. 12).

[4] M. Liu, "General Game-Playing With Monte Carlo Tree Search", Medium, 2017. [Online]. Available: https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238. [Accessed: 23- Mar- 2021].

[5] “Numba Python Compilation Library”, Numba, 2021. [Online]. Available: https://numba.pydata.org/. [Accessed: 19- Apr- 2021].