CS 4701: Generalized Engine for AI Game Playing Agents

Haashim Shah (hhs66), Harry Harpel (hah79), Stefan Brechter (scb262)

https://github.com/Ha5hBr0wn/CS4701

Introduction

Game playing has been a cornerstone in Artificial Intelligence throughout its history. Some of the most well-known and notable accomplishment in the field have been in game playing, including Deep Blue defeating reigning chess champion Garry Kasparov in the 90's, and AlphaGo defeating Ke Jie, the number one Go player, in 2017.

While not being the most practical of pursuits, the concepts put forward in developing AI game playing agents extend into all types of other applications. After all, game playing is just an exercise in decision making with either perfect information, in games like chess, or imperfect information, in games like poker. Such concepts easily find their way into being of extreme importance in applications focused on trading, advertising, forecasting, and many more.

One of the most notable algorithms used in game playing is Monte Carlo Tree Search (MCTS), a searching algorithm that utilizes a random approach to understand a game space that is far too large to search completely. In this paper we discuss our approach to creating a generalized engine for producing AI game playing agents that depends on this algorithm.

Monte Carlo Tree Search

We begin our discussion by describing the algorithm that backs the entirety of our work. In order to understand MCTS one must first understand the multi-armed bandit problem.

Imagine a gambler placed in front of k slot machines. Each slot machine draws a reward from a probability distribution that may or may not be the same as the one used for other slot machines. The goal of the gambler is to make as much money as possible with their pulls.

At the heart of this problem is the exploitation vs exploration dilemma. For example, the gambler might pull each slot machine once and then proceed to use the one that gave back the best return.

However, such a strategy is extremely prone to the gambler missing the "best machine" as perhaps they got unlucky the first time they tried it. In this case choosing exploitation over exploration harms the gambler.

In another scenario the gambler may pull all machines many times, in order to get a good understanding of the expected value of each distribution (Law of Large Numbers). From then on they choose to use the one with the best expected value. Again, this strategy is not great as the gambler wastes a lot of pulls on "bad" machines. So in this case choosing exploration over exploitation harms the gambler.

Instead a heuristic that balances exploration and exploitation optimizes the reward for the gambler. UCT is such a heuristic, and is of the form:

$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(N_i)}{n_i}}$$

w_i : total reward from pulling lever i

n_i: number of times lever i has been pulled (only apply UCT after pulling each lever once)

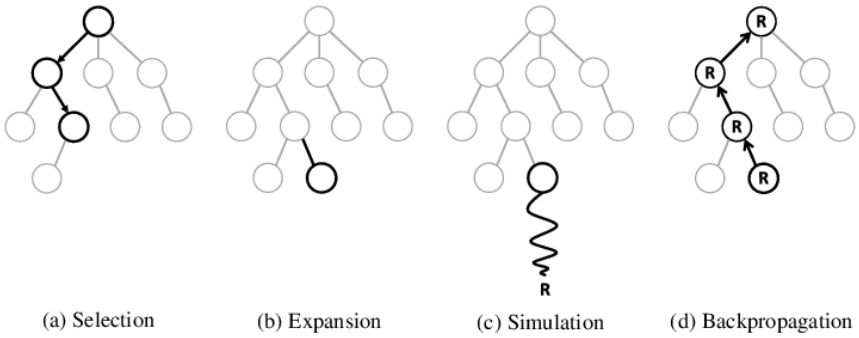
N_i : number of total lever pulls across all machines

c : the exploration parameter (theoretically $\sqrt{2}$ is optimal)

There are many concepts to glean from this formula. First note that every lever is pulled once initially which is a reasonable amount of exploration to do. From that point on the heuristic is applied, and the lever with the highest value is taken. The first term in UCT corresponds to exploitation. Namely levers with higher average returns have a higher first term. The second term in UCT correspond to exploration. Note how levers with a small amount of pulls when there have been a large amount of total pulls will have a higher second term.

Armed with this heuristic we can now describe MCTS. The main idea is to imagine a game tree in which each node corresponds to its own multi-armed bandit problem. The node (game state) is the gambler and each of the neighboring game states corresponds to a slot machine, that returns a reward depending on how good that neighboring game state is for the acting player. The search is adversarial, so we assess the worth of an action for different players at different depths.

There are four main phases in the algorithm as depicted in the figure below:



https://www.researchgate.net/figure/Phases-of-the-Monte-Carlo-tree-search-algorithm-A-search-tree-rooted-at-the-current_fig1_312172859

During selection we search the tree according to the UCT heuristic we described above until we reach a node that doesn't have all of its neighbors expanded (i.e there is a slot machine we haven't tried for this gambler). We then expand the tree by adding a node corresponding to that previously untaken action.

At this point we actually need to determine the reward for that action (slot machine). To do so we need a probability distribution of rewards to draw from. This distribution is created through random simulation of the game from that point on. The simulation can be truly random (light playout) or we can use some knowledge of the game to create a realistic playout (heavy playout). We also can choose to simulate to the end of the game where we have a definite winner, or to some intermediate state where we can apply a heuristic to determine the likely winner.

Regardless of our decision, once we have finished simulating in the way that we chose, we must now backpropagate. In this phase the nodes of the tree along the path that we had taken up through expansion are updated to account for the reward received to be used in the next selection phase when applying UCT.

After running simulations for some set amount of time or count the tree search returns the root level action that it has taken the most. From its perspective that is the best move to take (the slot machine that has given the most reward).

MCTS vs Mini-Max

In this section we compare and contrast MCTS with another well know adversarial tree search based game playing algorithm, Mini-Max. We aim to show that MCTS is the better choice for the algorithm backing a generalized engine for producing game playing agents.

Mini-Max at its core differs from MCTS as it fully explores the game tree up to some pre-specified depth. Once reaching that depth it must apply a heuristic to assess the worth of the encountered game state and backpropagate that up the tree.

Immediately we recognize that Mini-Max requires some prior knowledge of the game as one must have a heuristic to apply to game states in which the game may or may not be over. Furthermore, the performance of a Mini-Max agent entirely depends on the quality of the heuristic. Therefore Mini-Max is a poor choice for a generalized engine.

On the other hand MCTS always has the option to simulate a game to the end, where there is no need to apply a heuristic. In this sense it is much better for generalized game playing as it only needs knowledge of when a game is over, which is specified in the basic rules of any game.

MCTS also has the freedom to run for any prespecified time or simulation count, while Mini-Max must finish the computation that is has started. This makes MCTS much more customizable to a user's needs.

Finally, if a user does have prior knowledge of a game they can introduce that into a MCTS via a heavy playout, or with a heuristic that allows them to cut simulation short. For example, in a game of chess it would be unnecessary to continue simulating once a player is significantly up in material (captures a queen for example). This could be extremely useful in games that take a long time to simulate.

Connect 4 MCTS Implementation

After doing some initial research we began implementing our project in code. Our first goal was to make a concrete implementation of a MCTS agent for Connect 4, a classic game studied in AI. The

goal of this was not only to become more comfortable with the algorithm, but also to see just how good MCTS was against other game playing AI's (Mini-Max).

After finishing our implementation we pulled some Mini-Max implementations off of GitHub. We were unable to merge the codebases due to different data structures or entirely different languages, so we played a few games between the AI's manually to see how our MCTS did. These games can be seen in our repo.

The most notable of these games was game4, in which our MCTS implementation played against a depth 8 Mini-Max implementation. In the interest of space we do not copy the game here for you to see, but you should take a look in the repo linked above.

Our MCTS won the game with the final board looking as follows (MCTS played as 1):

```
0 1 2 1 0 1 0
0 1 1 1 0 2 0
0 1 2 2 0 1 0
1 2 1 1 2 2 0
2 1 1 2 1 1 0
2 2 2 1 2 2 2
```

As you can see the MCTS won along a diagonal starting in the first column. A notable intermediary position just two turns before looked as follows:

```
0 1 2 1 0 1 0
0 1 1 1 0 2 0
0 1 2 2 0 1 0
0 2 1 1 2 2 0
0 1 1 2 1 1 0
2 2 2 1 2 2 2
```

It was Mini-Max's turn to play in this position. There are multiple things to note in this position. First off there is basically no potential for Mini-Max to get 4 in a row at this point (besides straight up column 1 or 7). Further note that Mini-Max loses if it plays in column 1 or 5. So 7 is the only viable action.

Suppose Mini-Max plays 7. MCTS can follow with 5, forcing a 5 in return. At this point MCTS can play 5 again. Now Mini-Max is forced to play up column 7 losing in a total of 8 moves from this position (4 in a row in top row).

Since Mini-Max fully explores to this depth it understands that it is guaranteed to lose in this position. It turns out it did not even bother to prolong its defeat and just plays 1, and immediately loses (some tie breaking heuristic must have come into play). Note the Mini-Max did not see this coming during its previous turn as it is of depth 8, but even then it was already in a position where it was guaranteed to lose given perfect play.

It is also important to note that in this game Mini-Max took significantly longer to compute its move than MCTS (playing 5 seconds per move). This implementation of Mini-Max even utilizes alpha-beta pruning so the fact that MCTS won with less thinking time is quite impressive.

A reasonable explanation for the behavior we see in these games is that Mini-Max is primarily a defensive agent in Connect 4. It can see when it is going to lose 8 moves ahead and prevents it. The heuristic of just maximizing how many pieces it has in a row is not conducive to forcing wins a long way later into the game.

MCTS on the other hand (observed through many human played games against it) forces wins by blocking off columns that the other player can use, as seen in the example game above. This type of play requires seeing much more than 8 moves in advance, as there could be a lot of room left in the other columns that are still available.

The nature of MCTS makes it highly conducive to playing in such a manner, as by blocking a column it will simulate many games in which that column is randomly taken by the opponent in an incorrect manner resulting in a win for MCTS. With a high number of simulations it seems to even understand

the odd-even counting of who has to go in that column first. Overall MCTS, even with relatively little thinking time (5 seconds), is an incredibly intelligent Connect 4 player that is very difficult to defeat.

Refining the Abstract Interface for Using the Engine

Satisfied that MCTS is indeed quite an impressive game playing agent we moved on into refining the interface a user must implement to use our engine. The final interface was simply an abstract class called GameState that a user must provide a concrete implementation for with the game they want to have an AI play.

A screen shot of this interface is included below and should be read carefully as it is the main way to interact with the engine for producing MCTS agents.

```
/** The main trait required for a user to implement in order to utilize the MCTS engine.
 *
 * Note there is no explicit definition of terminal. Meaning the game is not necessarily over in a terminal state.
 * Namely it is just a state at which we can apply a heuristic to deem the worth of previously encountered states
 * during simulation. This would be done in the get_worth function. The simplest heuristic would of course
 * be one that just compares the winner if the terminal state corresponds to one where the game is over. */
trait GameState {

  /** Uses the current GameState to generate a LazyList of the neighboring GameStates that the
   * user wants to explore (technically does not have to be literally all neighbors).
   * The order of appearance on the LazyList should be randomized. */
  def generate_all_neighbors(): LazyList[GameState]

  /** Uses the current GameState to generate a random neighboring GameState used for simulation. The neighbor
   * does not have to be generated uniformly at random if a user wishes to use a heavy playout */
  def generate_random_neighbor(): GameState

  /** Returns true if this game state is terminal, false otherwise */
  def is_terminal(): Boolean

  /** Returns a value in the range [0, 1] determining how much the player who brought about this GameState won
   * given the terminal_state that proceeded it at some point in time. IMPORTANT NOTE: This should NOT evaluate
   * the state for the current acting entity but for the acting entity that brought about this state
   * A 0 corresponds with a complete loss, a 1 with a complete win, and it is the implementers choice to
   * decide if they want any values in between (what a tie corresponds to for example). Also for the root state
   * it is ok to choose an arbitrary value like 0 or 1 as there is no player who brought it about. */
  def get_worth(terminal_state: GameState): Double

}
```

It is important to note just how general the interface for GameState is. The is_terminal and get_worth function leave room for the user to implement heuristics given game specific knowledge if desired. Also, the generate_random_neighbor function leaves room for a user to implement a heavy playout by not generating truly random neighbors. Moreover, the user also has the ability to decide not to consider all moves. For example, in many games, like Gomoku, it really only makes sense to consider moves that are clustered close to where the other pieces are. Finally, there is no restriction on what a Player or an Action is, leaving room for a user to implement team-based games and games in which actions are themselves probabilistic.

Also, the generate_all_neighbors function takes advantage of lazy evaluation which has potential for providing performance boosts during the MCTS, as you only expand one neighbor at a time. It also leaves the potential for performing MCTS on infinite game spaces, although our standard MCTS agent that we provide would not be able to do so as it considers each neighbor at least once. On that note, even though we only have currently provided a single MCTS agent (StandardMCTSAgent), the structure of the project leaves room to add on different variations that a user could instantiate given that they have provided a concrete implementation of GameState.

Overall, the structure and design of this interface provides a very easy way for any user to produce an AI for any game that they have in mind.

Tic-Tac-N

At this point the goal of our project turned to getting a better understanding of how MCTS actually behaves under the hood of all the randomness. In order to do so we devised a game called Tic-Tac-N. The game itself is quite simple, as it is just Tic-Tac-Toe parameterized by the size of the square board (n) and how many pieces you need in a row to win (k). The name is slightly misleading as there are two parameters.

Note we were not trying to understand the game, or to even create a game playing agent that played it well, but rather to use the game as a tool to get a better understanding of MCTS, by varying different parameters of the game and the algorithm itself.

We did implement a nice GUI that a user can play with to interact with the MCTS manually if they pull the repo. In order to get the code working simply create an IntelliJ sbt based Scala project, and copy over the src folder and build.sbt file (and make sure you have Scala installed).

Gathering Data From Simulations

After finishing the Tic-Tac-N implementation, we used our refined MCTS game engine to run simulations of the game with different parameters.

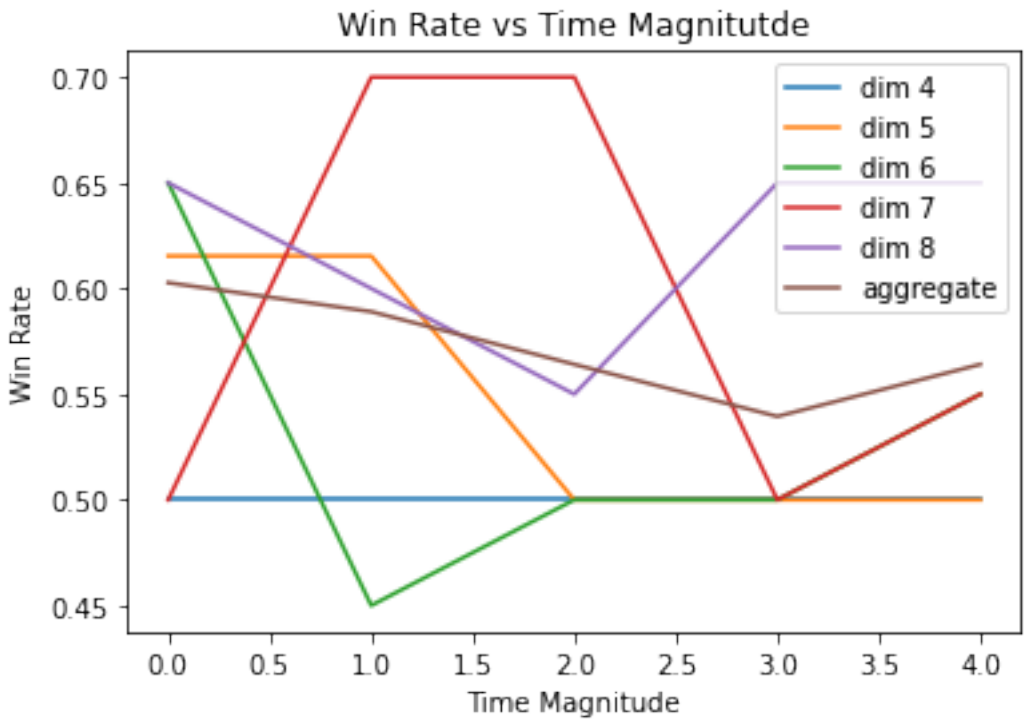
The simulations were organized into two experiments. In the first experiment we simulated different running time agents on different board sizes. The longer running time agent always thought for twice as long. The longer running times tested were 1, 2, 4, 8, and 16 seconds. We ran simulations on board sizes 4…8, and for each longer running time/dimension combo we ran 20 games (each game had k=4, and the player who went first alternated). So in total there were 20x5x5 = 500 games simulated for Experiment 1. The general purpose of this experiment was to see how much better longer running times made the MCTS on different types of game spaces.

In the second experiment we simulated games between equal running time agents. The running times were 2, 4, 6, and 8 seconds. We ran simulations on board sizes 5…12, and for each running time/dimension combo we ran 5 games (each game had k=5). So in total there were 5x8x4=160 games simulated for Experiment 2. The general purpose of this experiment was to capture cleaner data that didn't have varying times for analysis of multiple different aspects of MCTS (see below).

In order to actually gather the data efficiently we ran the games in a highly parallel manner, across multiple computers and threads. We also needed to move the data from Scala to Python for data analysis (to utilize good plotting libraries). In order to do so we had to come up with a serialized format representing the data for a game. We chose JSON due to the highly nested structure of the data we wanted to extract. Hence each simulation can be found in our repo as a JSON file.

Experiment 1

Once we had the data in JSON format we analyzed it in a Jupyter Notebook file that can be found in the repo. The first experiment as specified above yielded the following plot:



The plot is a lot to take in so we analyze it a piece at a time. First note the x-axis corresponds to the *time magnitude* of the simulation. This is the power of 2 that was used in the longer time. Hence points at time magnitude 3 correspond to games between an 8 second vs 4 second MCTS. The y-axis corresponds to *win rate* which is how many times the longer playing agent won over how many total wins occurred (0/0 corresponds to 0.5, if only draws occurred).

First note that nearly all data points are at or above the 0.5 line, meaning that longer thinking agents perform better as expected. Next note that for lower dimensions (4, 5, and 6) the win rate converges to 0.5 as time magnitude increases. This is also reasonable as two people playing Tic-Tac-Toe will likely tie even if one is "smarter" than the other. Finally note the aggregate curve that averages the dimensions together. It shows that the significance of the 2x thinking boost goes down gradually with a final kink up at the 16 vs 8 second agents. This could signify the realization of a strategy that only the 16 second thinking agent could see.

Upon request from Professor Selman, we simulated 100 additional games of dimension 7, with longer thinking time 8 seconds, as it seemed slightly abnormal that the 8 second and 4 second agents were performing equally well.

After running these additional 100 we checked the number of wins the 8 second agent had, the number of wins the 4 second agent had, and the number of ties that occurred. We obtained the following:

```
8 seconds won: 53/100
4 seconds won: 47/100
8 seconds tied 4 seconds: 0/100
```

We then asked ourselves if there was some imbalance due to who goes first that could explain why the agents were so even (both play first 50/100 times). We obtained the following

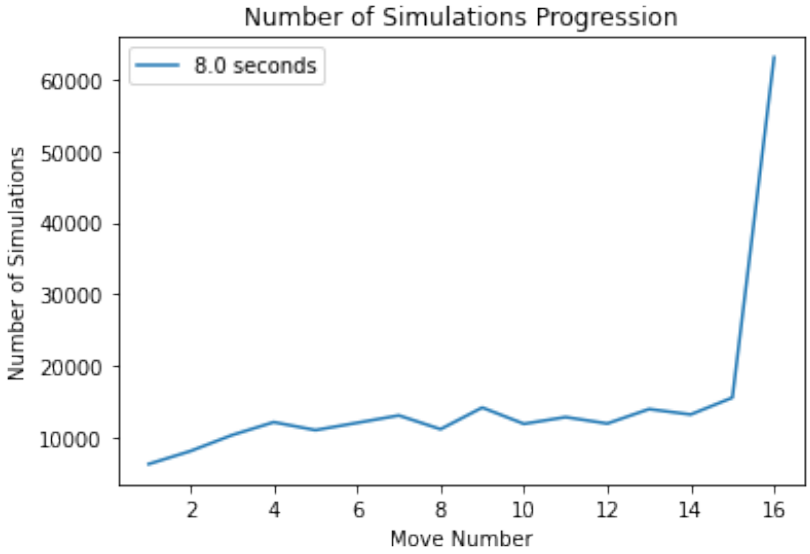
```
p1 wins: 71
p2 wins: 29
```

Clearly just the structure of the game favored the player who went first which explains some of the behavior we observe.

Experiment 2

Recall the second experiment was performed using agents with equal running time. With the time parameter fixed we could study the impact of other parameters as well as some other aspects of MCTS.

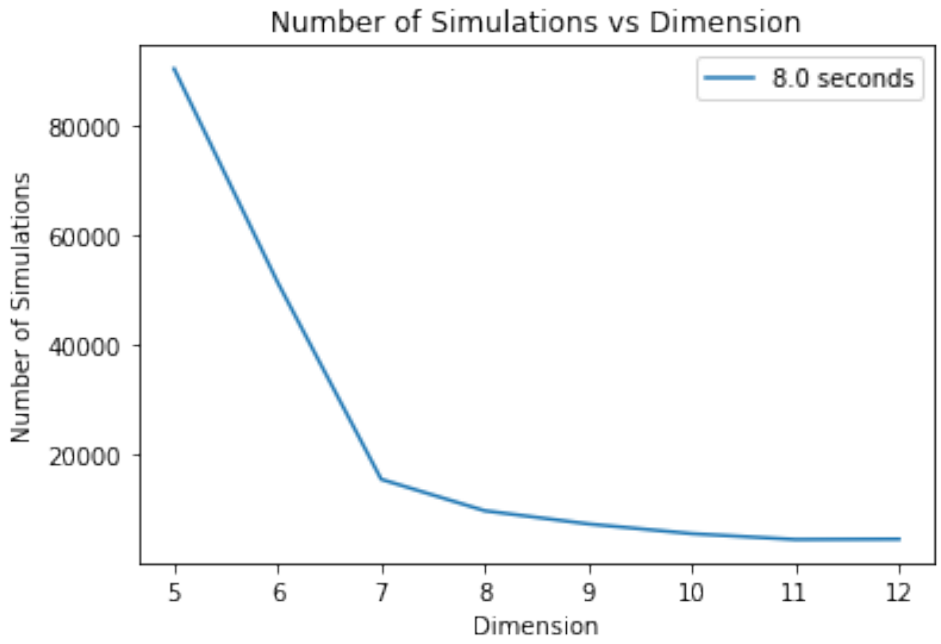
First we plotted how the number of simulations MCTS manages to run changes as the game progresses. We obtained the following plot:



While this plot is just using the data from a single game it shows a general trend found in all games (one can go change the indices in the notebook to see). The number of simulations ran slowly increases as the game approaches the end and then dramatically shoots up for the last move. The same behavior is observed in the Connect 4 games saved in the repo (see game4 where number of simulations was recorded).

This behavior makes sense according to how MCTS works. It should take less time per simulation as the game reaches its end as there is less space to search, also when there is a winning move the algorithm will simulate that one move game the most as it is the best move.

Next we plotted the number of simulations as a function of the dimension of the board. We see the following plot:

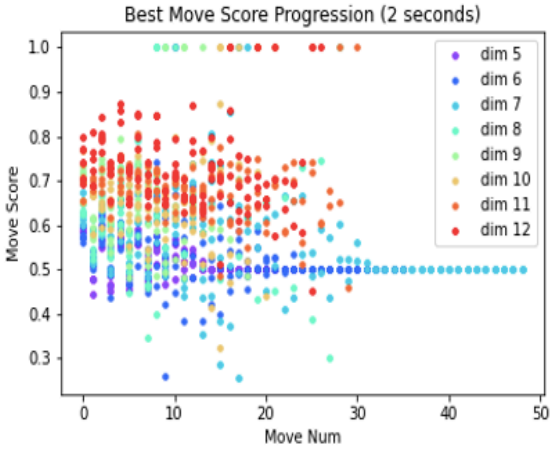


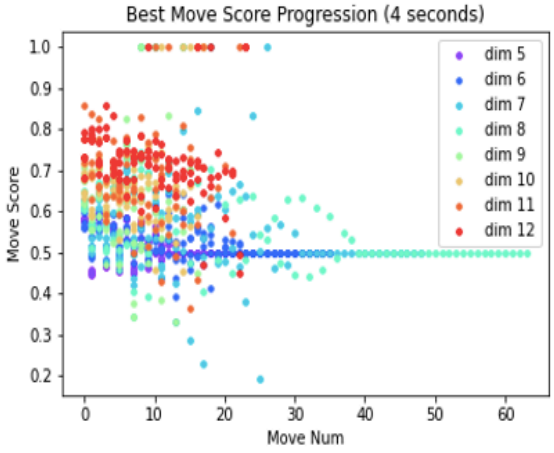
Again we see what we expected, a nice inverse relationship. As the dimension increases the search space increases and each simulation should take longer to randomly come to completion. This rapid drop off shows the importance of bringing in game specific knowledge for more complicated games, as otherwise the MCTS will not perform well.

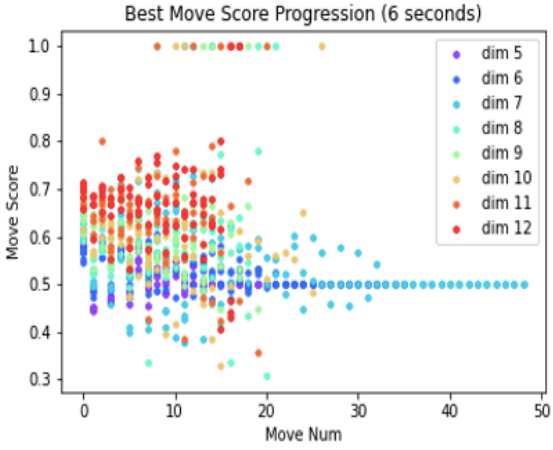
Through human playing with the Tic-Tac-N AI we saw that MCTS was terrible at Gomoku (19x19 requiring 5 in a row). It would achieve only about 2500 simulations in 5 seconds and make moves almost at random, not connecting its own pieces or blocking the opponent's.

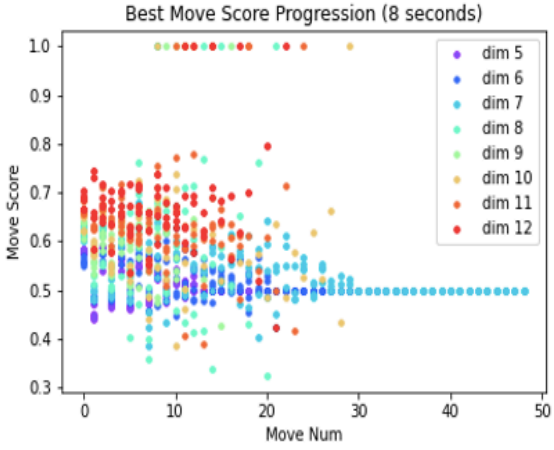
Optimizations to the MCTS on the backend and user implementation of game heuristics would be required to make a good Gomoku game playing agent.

The purpose of the next group of plots we created was to show how confident the MCTS was about its moves as a game proceeded in different dimensions given different thinking times. We made the following 4 scatter plots:







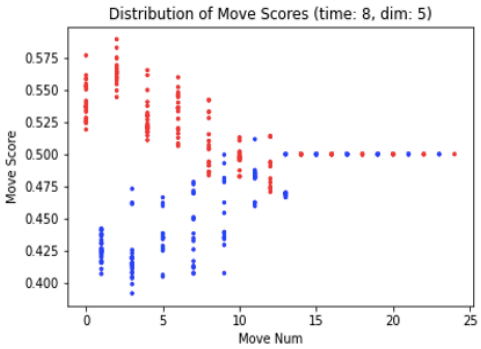


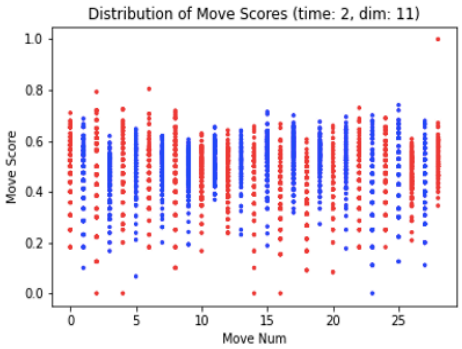
The y-axis on each plot signifies the *move score* of the best move that MCTS encountered during simulation for that specific game on that specific move (5 games for each time/dimension combination). We defined move score by the first term in the UCT heuristic. The x-axis signifies what move in the game we are considering and the colors represent on which dimension the game was played on.

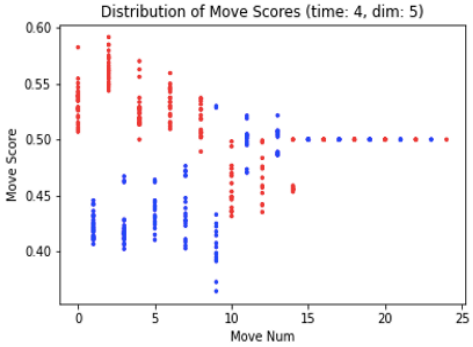
The first interesting thing to notice is just how confident the MCTS is on dimension 12. If you actually play AI on these dimensions manually you will see that it plays quite poorly at all of these running times. Also note that it is even more confident at even shorter times for this high dimension. We call this phenomenon *false confidence*. At this point we postulated that false confidence emerges from the fact that there are so many moves to choose from that the "best move" will randomly be an outlier in the simulation process (and even more so on shorter times as there are fewer simulations to bring it back to its true value). We later see evidence that supports this in a future plots.

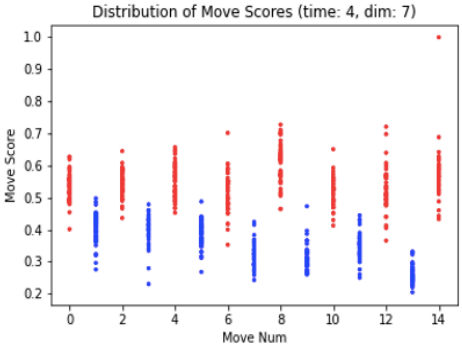
Also take note of the long streak at 0.5 for some games at mid-dimensions. These streaks occur more than 20 moves before the tie actually occurs, potentially promising that MCTS has the ability to see quite a bit ahead. However, this phenomena could also be explained by the nature of the game being prone to having a futile ending on the border of the board if there are no long streaks in the core to take advantage of. In this case future plots supported the second hypothesis.

To glean some final information we made plots of the distribution of move score over all moves considered, not just the best move. This resulted in 25 plots (see the notebook in the repo), one for each time/dimension combination. We choose to just show 4 of the plots here that give some insight into some of the key takeaways. The figures are shown below:









First note the plot on the top right supporting the phenomena of false confidence. We see that with little time and high dimension the moves that MCTS considers are highly clustered at some average with small tails at either end. By taking the best move we artificially skew towards higher move scores even though the data does not necessarily support it as a good move.

Next note that in the two games on the left, towards the end of the game all moves lead to a tie (overlapping at 0.5). This explains the tails at 0.5 in the previous plots as not being the MCTS seeing far ahead, but just being an artifact of the game structure.

The bottom right plot also shows an interesting result that agrees with the strange data point we observed earlier. We saw earlier that player 1 seems to have an advantage in this game we created, and indeed it is shown again. Notice how the first player is consistently confident that it is winning, while the second player is consistently sure that it is losing.

In the bottom left plot observe blue playing on the 9-th move. Up to this point blue had been losing but through simulation found a seemingly "brilliant" move that brought it back in the game, resulting in a tie. This shows the power in random simulation being able to find good moves.

Finally inspect the top left plot. This shows that at low dimension with high times both agents pretty much agree the game is a tie far in advance. The y-axis is a little misleading as it is compressed around 0.5 leading to visually large gaps that are not that big.

Conclusion

To serve as a recap, in this project we first researched different algorithms for game playing including MCTS and Mini-Max. We then created an initial MCTS implementation for Connect 4 and compared it to a Mini-Max game playing agent. Afterwards we refined our MCTS implementation to be accessed through a more general interface, so that a user could use the algorithm for any game that they specified. Finally, we created an implementation of our own game, Tic-Tac-N, to use as a window to gain insight into the MCTS algorithm. We simulated different types of games and gathered the data for plotting to see what interesting phenomenon we could find.

Throughout the project we gained practice with many important AI concepts including game playing, reinforcement learning, simulation, data gathering, data analysis/visualization, and system design.

Sources

https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
https://jonathan-hui.medium.com/monte-carlo-tree-search-mcts-in-alphago-zero-8a403588276a
https://web.engr.oregonstate.edu/~afern/classes/cs533/notes/uct.pdf
https://en.wikipedia.org/wiki/Minimax
https://github.com/marcomelilli/four-in-a-row-js-minimax
https://github.com/sachinparyani/Minimax-Connect-4