CMPT 310 Final Project Report

Reversi with Mote-Carlo Tree Search

Paige Tuttosi

This report my also be found in the README file, with clear rendered markdown.

1 Introduction

This is a program to play Reversi against the computer. The AI player uses variations of Monte-Carlo Tree Search(MCTS), with varying heuristics to allow the player to play at different difficulty levels.

The motivation for this program is whether Julia can be used to write a better, "smarter" AI player by performing more playouts over the allotted natural play time of 5 seconds. Pure Monte Carlo Tree Search is compared with Monte Carlo Tree Search with tactics and Alpha Beta Minimax search over several levels of playouts in order to compare their intelligence.

The expected outcome is to see Julia perform several more playouts per second than Python is able to resulting in a smarter Pure Monte Carlo Tree Search AI player. With few playouts we should see both Alpha Beta and Monte Carlo Tree Search with tactics easily outperform the Pure Monte Carlo method. However, once a large number of playouts are performed we should see Pure Montre Carlo become a strong competitor for both Monte Carlo with tactics and Alpha Beta Search.

2 Implementation

An animated example of the UI can be found in the README.

2.1 Python library

The GUI for this code was forked from this git repository. My own modifications can be found in my GitHub repository. An animation of an example diff can be found in the README. I have also commented directly in the code with very clear hash marks where my code is residing (figure 1.).

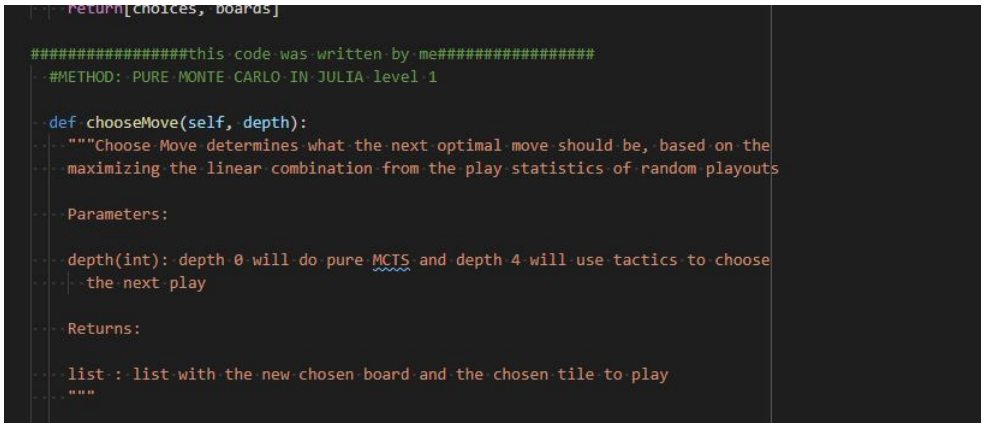

Figure 1: Example of commenting for my own code.

2.2 My Python

PyJulia has been used in order to interface the Julia environment through Python. In order to be sure that the code is correctly dropping to the Julia environment rather than simply leveraging Julia code within the Python environment a test was run. Both the **@** and **dot()** method of matrix multiplication were run within Python, following a call to the Julia function ***.** All were timed. As expected if we are correctly running within the Julia environment, the Julia function has much smaller run times (figure 2.).

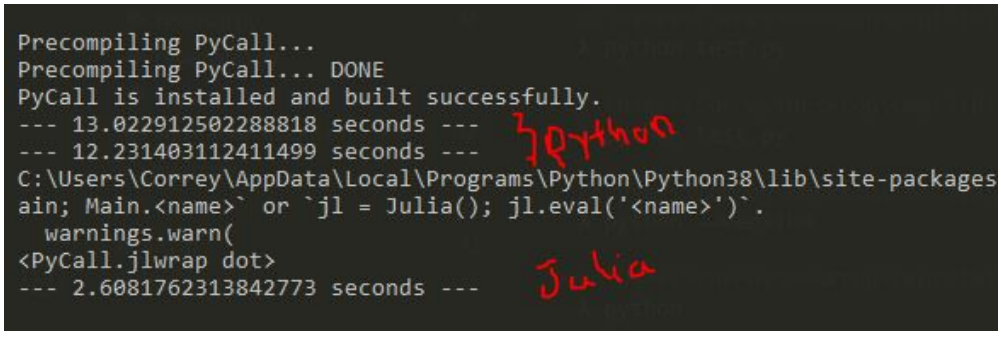

Figure 2: Timing for matrix multiplication in Python vs. Julia.

2.3 Julia

Julia is generally believed to be much faster than Python, and in some cases faster than C or C++.

"[Julia] was designed and developed for speed, as the founders wanted something 'fast'. Julia is not interpreted...it is also compiled at Just-In-Time or run time using the LLVM framework."
Reference

Other resources:
Toward Data Science: 5 ways Julia is better than Python
Info World: Julia vs. Python, which is best for data Science

On top of this Julia is truly gaining a name for itself in the world of AI. The community is thriving and as more and more people join to work on the code and documentation we may one day soon see Julia surpass Python, with its prominent AI libraries such as jax, pytorch, and tensorflow, as "the" AI language Reference.

3 Speed Comparison

In order to validate the claims that Julia is, in fact, faster than Python, the program was run in AI vs AI playouts in both languages and the average run times of each of the models was recorded (for more on which models were tested see section 4. Models).

This was done for 1, 5, 10, 100, 150, 175 and 200 playouts, and a depth of 2, 5, 6 and 7 for Alpha Beta. Except in cases where the testing was cut off early when the games were clearly taking longer than was feasible for normal game play at the given depth or number of playouts.

The comparison can be seen in figures 3-7 and the average of 50 games was taken.

3.1 Playout timing results

The "plays" plots (figures 3,5, and 7) use a log time scale and are the total time it takes for the AI to choose a move and update the board, we want to keep this under 5 seconds for smooth game play.

From these plots we can see that not only is Julia faster than Python, but the speed decreases at a slower rate as the playouts increase. The "playouts" plots (figure 4 and 6) show the time it takes to make a single playout, this helps us get the number of playouts per second for each language. We can see, as is expected, that the time to make a playout is relatively stable independent of the number of playouts. This measure is also independent of the number of empty valid place spaces unlike the timing for the full play.

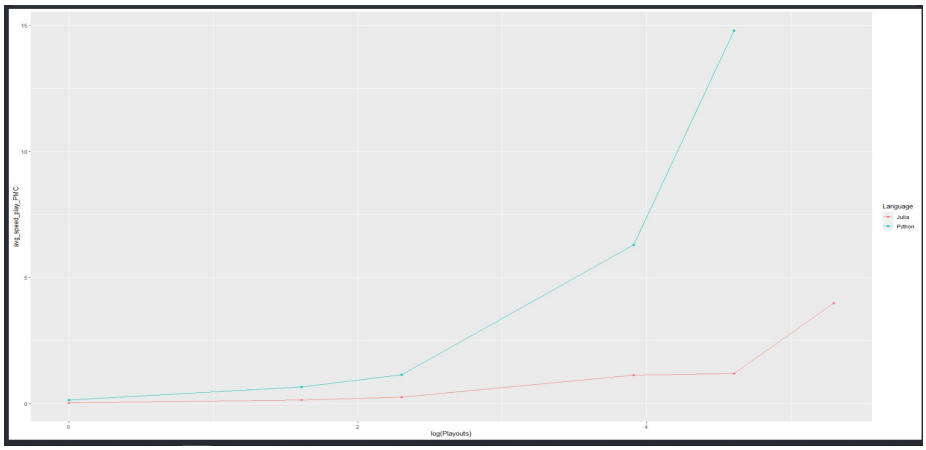

Figure 3: Pure Monte Carlo Tree Search time per play in seconds by log of the number of playouts, colour by language.

We can see that a single Pure Monte-Carlo Tree Search(PMCTS) playout in Julia takes approximately 7.46×10^{-5} seconds, or it can make 13404 playouts per second. Python on the other hand takes 4.56×10^{-4} seconds to complete a playout, or 2188 playouts per second. We see similar results for MCTS with tactics with Julia taking 7.27×10^{-5} seconds to complete a playout (13755 playouts per second) and Python requiring 6.44×10^{-4} seconds (1552 playouts per second). There appears to be a drop in time per playout for MCTS using tactics in Julia, which is surprising as several heuristics need to be run (see section 5. Heuristics) and may speak to the speed of the built in function to choose a random integer compared to hand written heuristics. However, after completing a two sample T-test on the difference in means it was found that the means were not significantly different at a level of $\alpha = 0.05$ (figure 8.). Whereas when completing a two sample T-Test on the difference of means between MCTS with tactics and PMCTS in Python we see that there is a significant difference in means at a level of $\alpha = 0.05$ with p $= 2.2 \times 10^{-16}$, similarly the confidence interval does not include 0 (figure 9.).

This suggests that the calculation of the heuristics in Julia takes a near negligible amount of additional time compared to a random choice. The increase in playouts per second is likely due to differences in the number of empty variables to check during each playout, as well as general background processes that make slight variations in timing every run. Whereas this is not the case in Python where we see a significant decrease in playouts per second when adding tactics and it is clearly and addition of code that is causing the slowdown.

From figure 5. we see that Julia can run 200 playouts in under our allotted 5 second play time, whereas Python can only run about 45 playouts in 5 seconds using both MCTS with tactics PMCTS. These numbers of playouts have been

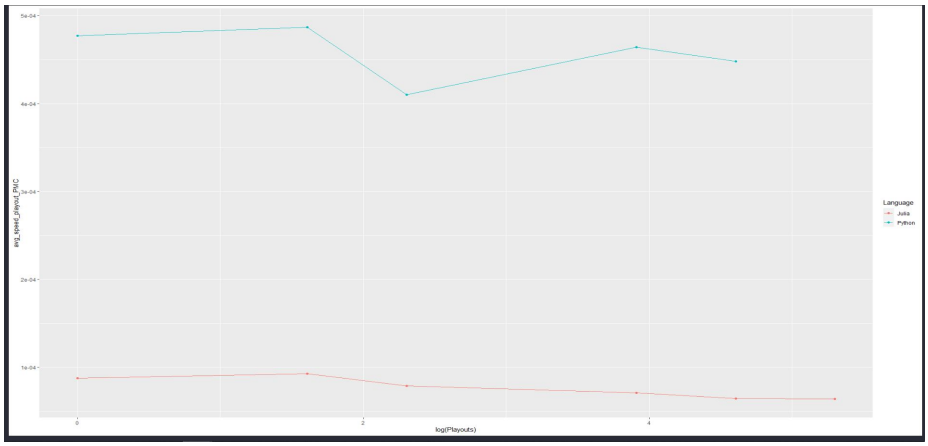

Figure 4: Pure Monte Carlo Tree Search time per playout in seconds by log of the number of playouts, colour by language.

chosen for the main program implementation.

Alpha Beta search is a slightly of a different case. Alpha Beta search does not rely on playouts, instead it is a recursive tree search and relies on the search depth (see section 4.3 Alpha Beta). This is why in figure 7 we plot playtime in seconds against depth rather than the log of playouts. Because of Alpha Beta's recursive nature timing was only done for the entire time it takes to choose the next move, rather than one single search, or the search of one node in the tree. This lead to a large range of play choice times at larger depths. When there are few valid play spaces the search is much quicker than if there are several, as it needs to complete the full tree search at the given depth belonging to every one of the nodes. This is why we see what appears to be a linear increase in figure 7. rather than exponential as we would expect over an increased number of experiments. Once we reach 50 playouts we can see a range in the time to make a play choice anywhere from 0.09 seconds per play to over 100 seconds per play in Julia and similar ranges in Python, although the fastest in Python in over 1 second.

As a future improvement to the Reversi AI it could be beneficial to create a cut off time, or conditional where if the number of valid play spaces is large, the depth is reduced. As such we would be able to increase the base depth in Julia beyond 7. Running tests at a consistent depth of 9 some games were played quickly with playtime's of close to one second, however, it was easy for the game to be caught up in a large search tree and take over 500 seconds, which is unfeasible for our game. A depth of 5 was chosen for our main implementation, since at this depth plays remain consistently under the 5 second limit.

In order to make efficient use of time these tests were done by running the AI against itself. The outcomes of these games were used to monitor that the

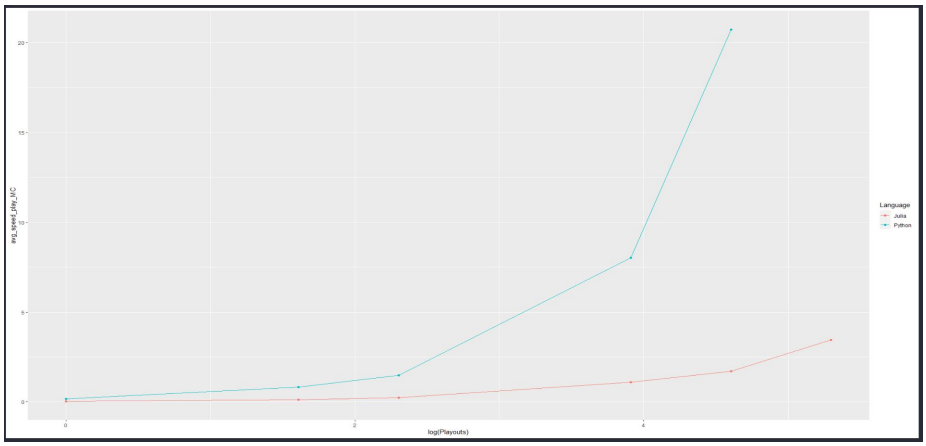

Figure 5: Monte Carlo Tree Search with tactics time per play in seconds by log of the number of playouts, colour by language.

"smarter AI" were, in fact, winning more games (see section 6 AI vs AI).

3.2 Note on Julia optimization

In general programmers tend to be most comfortable with programming in row major fashion as they begin learning in languages such as C and C++, Python and Java. Python, however, is a special case where **numpy** can handle can adjust storage to both row and column major variants, but is most often coded for row major efficiency.

An example of row major iteration:

```
import numpy as np
my_array = np.array([[1,2,3],[4,5,6],[7,8,9]])
rows,cols = my_array.shape
for i in range(rows):
  for j in range(cols):
    print(my_array[i,j])
```

Julia, however, is a column major language. This means that consecutive elements of a column are positioned next to each other in memory, i.e., they are sequentially stored in memory across the columns. Therefore by programming array access as follows, you can see a significant boost in processing speed by sequentially accessing the memory locations.

```
my_array = [1 2 3
            4 5 6
```

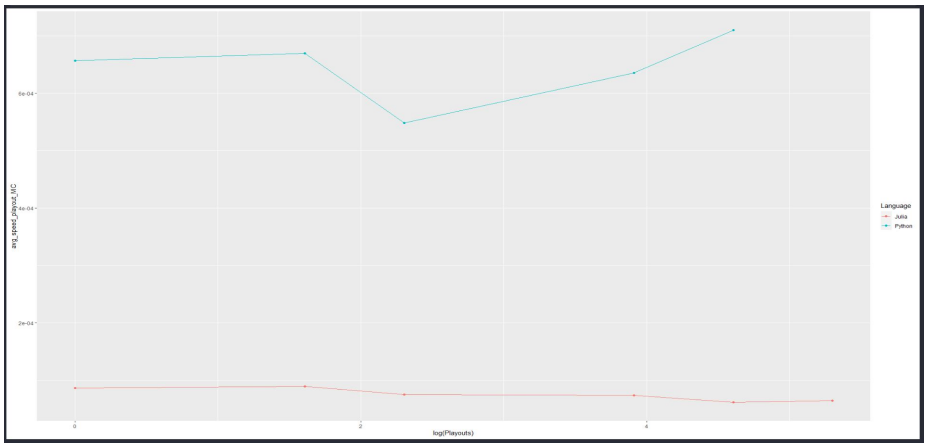

Figure 6: Monte Carlo Tree Search with tactics time per playout in seconds by log of the number of playouts, colour by language.

```
            7 8 9]
rows, cols = size(my_array)

for j in 1:cols
  for i in 1:rows
    print(my_array[i,j])
  end
end
```

Column major programming was used in Julia to increase the speed for this project.

Reference

4 Models

The models are written in both Python and Julia, however, the working implementation uses Julia.

When selecting the difficulty to begin the game, you may select between 1 and 3 stars. These stars correspond to increasing intelligence in the level of the AI opponent.

4.1 PMCTS

One star implements the PMCTS through the function **chooseMove**. This algorithm does K (**playouts**) random playouts for each of the valid moves as

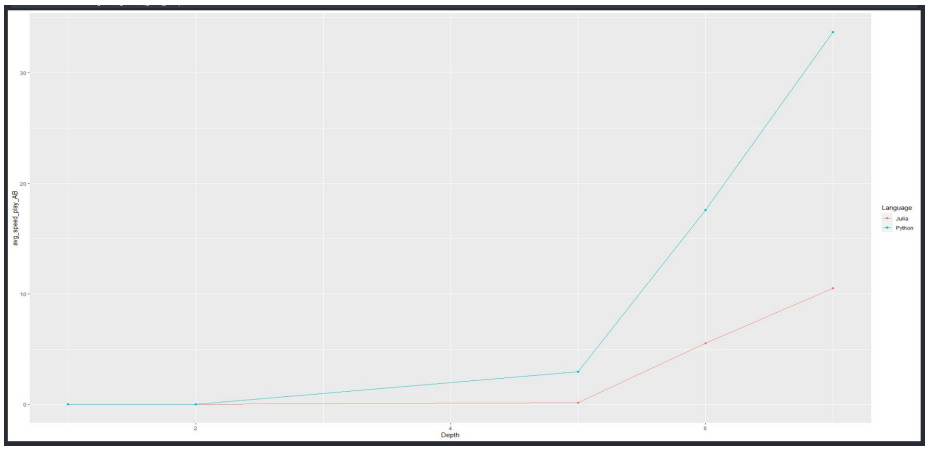

Figure 7: Alpha Beta Search time per play by depth, colour by language.

```
data:  julia$PMCTS.Playout and julia$MCTS.Playout
t = 1.6228, df = 767.96, p-value = 0.105
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.985556e-07  4.200096e-06
sample estimates:
   mean of x    mean of y
7.463771e-05 7.273694e-05
```

Figure 8: Two sided, two sample t-test on difference of means for PMCTS vs MCTS in Julia.

determined by **getPlays** and **valid**. During a playout a random, valid, position is chosen, then this new board is passed on to the function once more continuing until the game has come to an end, i.e.

```
(self.won == True)
```

. The number of wins, losses and draws for each of the *K* playouts is saved and a score is calculated as follows:

$$score = wins + draws * 2 - losses * 5$$

This score is stored in a dictionary with the index of the resulting board as the key. Once all of the playouts are complete the maximum of these scores is found, and this index becomes the chosen board.

```
data:  python$PMCTS.Playout and python$MCTS.Playout
t = -13.289, df = 397.06, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.0002165730 -0.0001607508
sample estimates:
   mean of x    mean of y
0.0004555436 0.0006442056
```

Figure 9: Two sided, two sample t-test on difference of means for PMCTS vs MCTS in Python.

4.2 MCTS With Tactics

Two stars implement MCTS, again, through the function **chooseMove**. This algorithm does K (**playouts**) playouts for each of the valid moves as determined by **getPlays** and **valid**. Rather than selecting a random play the next play is determined by the maximum of the **finalHeuristic** score. More info about the heuristic scores can be found in section 5. Heuristics. Essentially, this score is calculated based on known tactics, which increase the player's chances of winning a game of Reversi.

The rest of the function continues in the same way as section 4.1 PMCTS.

4.3 Alpha Beta

The Alpha Beta pruning on a MiniMax tree is an extension of the MiniMax search algorithm.

The MiniMax algorithm attempts to minimize the possible loss and maximize the gain(ie. wins). MiniMax looks to find the highest value that that player can achieve, without any knowledge of what actions the other player will take. In a way this simultaneously forces the other player to receive the lowest value possible.

Essentially, while traversing through a tree each level will represent alternating players, with one player maximizing their value and the other minimizing their opponent's value. The algorithm will always choose the move that results in the best move for that player. The algorithm never wants to choose a node that may result in a loss.

Once the entire tree is traversed, the branch which results in the most likely win scenarios, given both players maximizing their plays, is chosen as the next move.

Interesting Note : Because, as we traverse the tree and each player is assumed to be maximizing their chance of winning, Non Zero Sum Games are at Nash Equilibrium when MiniMax is followed.

MiniMax resources:

Algorithms Explained: YouTube video
Wikipedia Article
Lecture Notes

Alpha Beta pruning helps the MiniMax algorithm be more efficient. The MiniMax algorithm will tend to explore parts of the tree that it does not need to be looking at. Min and Max bounds are used to limit which branches of the tree are searched to avoid unnecessary computational time and power.

Alpha is the best already explored option along the path to the root for the maximizer, and Beta is the minimum already explored option along the path to the root for the minimizer.

Alpha Beta resources:

Step by Step Alpha Beta Pruning: YouTube
Wikipedia
Geeks for Geeks: Minimax algorithm in game theory
Cornell Lecture on Alpha Beta Search
Lecture Notes

We start assuming the worst case for the maximizer (root) at -infinity for Alpha and infinity for Beta.

Alpha and Beta scores are calculated using the **finalHeuristic** (more on heuristics in section 5. Heuristics). Alpha Beta is recursively called in order to traverse the tree while swapping between minimizing and maximizing. When minimizing the new score value is compared to the current Beta value that has been seen in the tree. If the current value is less than the saved Beta, the branch will continue to be explored, however, if the value is larger than one we have already seen we can prune this branch ($beta <= alpha$).

The same method is followed when maximizing Alpha.

5 Heuristics

There are five score heuristics used in the program.

simpleScore : This simply tallies the number of resulting tiles of the player's colour after they have made a given move against the number of tiles held by the opponent.

slightlyLessSimpleScore : Takes advantage of the most simple tactic by weighting the player owning corner and edge pieces higher as they have strategic advantages.

decentHeuristic : Similarly takes advantage of the corner and edge strategy but penalizes for the tiles next to corners, as these result in the loss of the strategic corner locations.

earlyGame : Early on in the game, while they are still available, the player wants to control the entry points to the corners in order to force the other player into the spaces next to the corners. These are called the power pieces. These positions are weighted the highest in this heuristic.

finalHeuristic : This is the heuristic used in both the MCTS with tactics and the Alpha Beta pruning and is a combination of the aforementioned tactics. This method takes into account the timing of the strategies. For example, we begin by wanting to to control the power pieces, but as these will likely soon be unavailable attention is shifted to the corners while avoiding the positions next to corners. In the end as most positions are taken the strategy becomes to simply control the most pieces on the board.

Reversi strategy resources:

Strategy Guide for Reversi
How to Win Reversi

6 AI vs AI

When completing the speed tests(see section 3. Speed Comparison) the games were being played AI vs AI. We not only want to know comparisons of speed between the methods, we also would like to see an improved performance as new heuristics are added. This is, however, not always the case, as the relationships between the algorithms are complex. However both Python and Julia show a very similar pattern of winning models, suggesting that the winning model is independent of the language.

In order to run these tests you can run the: **runTest.sh** code.

In figure 10. the results of the test runs can be seen.

6.1 PMCTS vs MCTS with tactics

For low numbers of playouts, ie. under 50, MCTS with tactics has a clear advantage over PMCTS. This is as expected, with few random playouts we cannot expect a purely random model to perform well, however, by adding prior knowledge of game play we should expect to see a smarter AI even with lower playouts.

Once we reach 50 playouts we see MCTS with tactics beginning to lose it's edge over PMCTS. Once 150 playouts or more are reached the models appear evenly matched. This, again, is expected as we increase the number of playouts PMCTS is expected to perform just as well, if not better than models written with tactics, while taking less time to compute the optimal play (in the case of Python).

models	Language	Playouts	most_wins
PMC VS MC	Julia	1	MC
PMC VS MC	Python	1	MC
PMC VS MC	Julia	5	MC
PMC VS MC	Python	5	MC
PMC VS MC	Julia	10	MC
PMC VS MC	Python	10	MC
PMC VS MC	Julia	50	Barley MC
PMC VS MC	Python	50	Barley MC
PMC VS MC	Julia	100	Barley PMC
PMC VS MC	Python	100	Barley PMC
PMC VS MC	Julia	150	close to 50/50
PMC VS MC	Python	150	NA
PMC VS MC	Julia	175	close to 50/50
PMC VS MC	Python	175	NA
PMC VS MC	Julia	200	close to 50/50
PMC VS MC	Python	200	NA
PMC VS AB	Julia	1	50/50
PMC VS AB	Python	1	50/50
PMC VS AB	Julia	5	AB
PMC VS AB	Python	5	AB
PMC VS AB	Julia	10	AB
PMC VS AB	Python	10	AB
PMC VS AB	Julia	50	50/50
PMC VS AB	Python	50	AB
PMC VS AB	Julia	100	50/50
PMC VS AB	Python	100	50/50
MC VS AB	Julia	1	50/50
MC VS AB	Python	1	50/50
MC VS AB	Julia	5	AB
MC VS AB	Python	5	AB
MC VS AB	Julia	10	AB
MC VS AB	Python	10	AB
MC VS AB	Julia	50	AB
MC VS AB	Python	50	AB
MC VS AB	Julia	100	AB
MC VS AB	Python	100	50/50

Figure 10: Excel column of the winning model for differing numbers of playouts.

6.2 MCTS vs AB

Both MCTS algorithms perform similarly against Alpha Beta search, so they will be discussed together here.

When the search depth is 1, only the first set of resulting playouts will be searched. This is not particularly informative so it is not surprising it has little advantage over both PMCTS and MCTS with tactics resulting in close to a 50/50 win ratio. Once a depth of 5 is reached, however, a more robust search is performed covering a multitude of possible play scenarios. This results in Alpha Beta winning the vast majority of games against both PMCTS and MCTS with tactics.

Once we reach a depth of 7, we are similarly reaching 100 playouts and we begin to see PMCTS performing well as we did with MCTS, we even have

MCTS coming as a close contender with Alpha Beta at 100 playouts. Once again, this is expected, we should see PMCTS becoming increasingly smarter as more playouts are added.

7 Conclusion

Although Python is an excellent language for many software development projects, when it comes down to needing pure speed from your algorithm Python is not the best choice. Julia is a new up and coming language in the AI field with AI libraries rivaling Python while at the same time having speeds competitive with C++. Julia is an excellent choice for Monte Carlo Models.

Using Julia we were able to achieve 4 times the amount of playouts as we were in Python, and the majority of the time is spent updating the graphical GUI. Had we played solely in command line we would likely have been able to have playouts in the thousands while maintaining a natural play speed of approximately 5 seconds.

Through AI vs AI playouts the "intelligence" of Pure Monte Carlo Tree Search, Monte Carlo Tree Search with game play tactics and Alpha Beta Minimax was compared. As long as the depth is greater than 1 we see Alpha Bets as a particularly strong contender, winning most games consistently against both PMCTS and MCTS with tactics over a range of playouts. However, we confirm that PMCTS may begin to outperform both Alpha Beta and MCTS with tactics as long as enough playouts are performed.

Therefore, if the computational power is available the fastest and least complex model of choice would be PMCTS. It requires none of the prior knowledge of MCTS with tactics and there is no chance of it getting stuck in a deep search tree as with Alpha Beta search. With only this relatively limited testing and computational power it is easy to see how AI giants, such as Alpha Go, have been built on purely random models.