# Journal

## Milestone 4

For this milestone, my main focuses were optimization. I decided the best ways was multithreading and making better data structures. I did a large refactoring of the game using my new data structures to cut down on gamestate updating and comparing time. I also added in timed profiling to find out what is slowing down the AI the most. It was by far the simulations that take the most time. When letting it run for 5 seconds, ~4.7 was the simulations. To help with this I decided to thread only the simulation part. The speedup mirrored the number of threads I added. This may have been obvious but speedup doesn’t change how well the AI does, so the AI still loses almost every game to Big Money.

To make the game more playable I also added an extra thread for the rest of the AI. This means the game no longer freezes while running simulations, and a move can be made while simulations are running. This has made the game MUCH more usable.

Up next in my bucket is updating my controls, allowing the exploration parameter to be changed dynamically, testing changes to how best move is calculated, switch between single and multithreaded, and learning how to save off the tree. I believe I’m in a decent spot

## Milestone 3

When testing Monte Carlo no tree search, I tried a few things. I tried purely random moves, assume big money for opponent, and assume big money for both. The best result was assuming big money for both. I thought just doing it for the opponent would find the best move, but the problem is the AI returned 0 wins over a thousand simulations. This shows just how dominant big money is. When only treasure, curse, and victory cards are in the game, Big Money is almost perfect play. The only times it isn’t is when you get a lucky early 8 money (buy a province when you are money isn’t consistently high enough), or a late unlucky 5 or less (buy a silver when your money density is already above 2). Both of these could be easily coded in although I’ll hold off on optimizing the simple AIs for now. MC had a ~50% from my auto simulated games.

My original proposal had me trying out different variations of mcts to handle hidden information. I realized this wasn’t feasible since getting one variation working was hard enough. My approach was instead of just playing a game from the simulation phase, I’d play a game from the current gamestate (normally the first move). If ever I reached a new game state while selecting moves, I’d use this as the gamestate to expand. This idea was the easiest to wrap my head around since I wouldn’t have to algorithmically solve all of the different combinations. Instead I’d just need a way of comparing gamestates without worrying about the order of cards. Currently I’m sorting my vectors and then comparing, but I’ll most likely be changing to an unordered multiset going forward.

I only was able to play one game with the current time interval, but the mcts ai lost 35 to 50 against big money. It was winning for a majority of the game, but lost when it ran out of cheaper victory point cards to buy. The big money strategy doesn’t gain any victory points until it can buy the most expensive ones. It looks like there is a local maximum with only a few thousand simulations. I’d like to see it if I let it run for a few hours.

The AI now is much slower than only simulating over the current moves, so next on my list is to simulate with a time limit.

## Milestone 2

Milestone 2 started on a rocky start. Tic Tac Toe wasn’t finished, so the first part of this milestone was getting that to a completed state. It never was perfect because I realized MCTS doesn’t handle guaranteed wins 7 moves in advance without massive numbers of simulations. I had conversations with Butler, Dr. Clark, and to a lesser extent Squirrel on the problems I faced. Because of dealing with this I started on the real meat of milestone 2 roughly a week late.

The goal of this milestone was to build out the base game and then implement Monte Carlo. I underestimated how long it would take to build these systems. The big issue was the complexity of the game is much higher than Tic Tac Toe, and I made the mistake of trying to implement a little too much right away. Dominion needs Decks, Discard piles, play areas, and hands for each player. It also needs to keep track of what phase ( Action, Buy, or Cleanup ) the game is in. The data structures to house everything needed to be much bigger and more powerful. I was able to get everything working although I definitely want to go back and optimize the code. My Deck class morphed into more of a player board since it housed the discard pile, hand, and play area as well. Naming became a problem, and I ended up with tons of enums to handle everything. For example there are enums for card type, move type, game phase, and cards themselves. I questioned often what exactly the data structures should store. Should the deck be a vector of ints that correspond to an index into an array of Card Data? Ints are small and I was able to get away with storing everything like that in Tic Tac Toe, but I ended up deciding that it should be a vector of pointers to card definitions.. No longer is a turn a single move. I decided on number keys being the buying of cards and Enter to be moving to the next phase. Currently the screen is a bunch of debug placeholder information.

I was not able to work on the implementation of Monte Carlo, so this next milestone I need to do catch up, and get that working. Hopefully this is something I can get done before the weekend, so I can have a clear picture of hitting the difficult problems of MCTS.

## Milestone 1

For milestone 1, the goal was to create Tower of Hanoi and Tic Tac Toe and then build Monte Carlo Tree Search (MCTS) AIs to play them. I was able to build playable versions of both games but I only finished the AI for Tower of Hanoi.

As with DFS, I learned there’s an annoying amount of setup involved even if the projects are simple. I started with Tower of Hanoi first and found the biggest hurdle was writing all of the helper methods needed for MCTS to function. I also wrote this iteration poorly. I first wrote Monte Carlo without the tree search part which surprisingly worked. The AI found winning moves. When I changed to MCTS I made the decision of building my tree structure with all of the methods bake in. This made all of the code tangled together. It became difficult deciding if a method should be in Game (the manages the game), MonteCarlo (manages the AI), and mcts tree node (is and also manages the data structure???). Once implemented, the AI had tons of bugs that were hard to debug because many would happen only hundreds to thousands of simulations in. I even hit a stack overflow from recursively looping over too many nodes. Also, Tower of Hanoi has specific problems to it like finding the quickest solution is the goal instead of just a solution. In the end, the AI performed significantly worse the more rings that were added. This may seem obvious, but it was hard to tell when it was a bug versus when it was a limitation of the implementation.

I also didn’t realize I was missing one of the most important rules of Tower of Hanoi until the last weekend. You can’t put large rings on top of smaller rings. This was really dumb on my part, and when I changed over to the correct implementation, more bugs were revealed.

When starting Tic Tac Toe, I decided to have more structure to what does what. Game will have all methods that deal with the game. For instance, you can ask it what are valid moves, is a move valid, and given a game state + move return the updated game state. It is important that the AI doesn’t have any ways that can break the game that player wouldn’t be able to as well. TreeNode is a data structure only and has no methods on it at all. This may be an overcorrection from Tower of Hanoi but doing it barebones can help with determining what’s actually useful. Currently it contains pointers to its parent, children, and a data struct. The data struct then contains the meta data, game state, and input move to reach this node. MonteCarlo then has the 6 main methods of RunSimulations, GetBestNodeToSelect, ExpandNode, RunSimulationOnNode, BackPropagateResults, and GetUCBValueAtNode. There’s a bunch of helper methods to go along with this, but the main idea is Game will call RunSimulations passing an amount. MonteCarlo then calls the next 4 in a loop with the UCB value being used in GetBestNodeToSelect. The really nice thing about this implementation is there’s very little game knowledge that MonteCarlo holds. This will be extremely useful when I start on Dominion. The AI is not done yet, but it is close. Unfortunately its an all or nothing situation.

I did a little more research on handling hidden information. I’m still not set on my solution, but there’s some interesting ideas to try. One would be choosing between simple state machine AIs to solve parts of a turn like in Hierarchical Portfolio Search. The benefit is a MUCH smaller tree with the drawback being the simple AIs may not contain the exact best move.