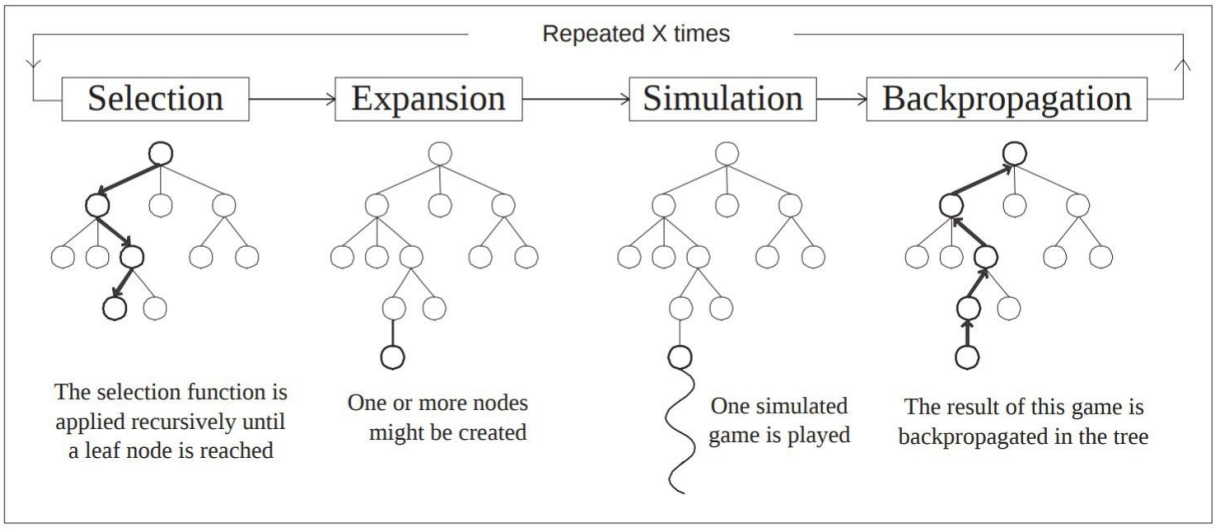
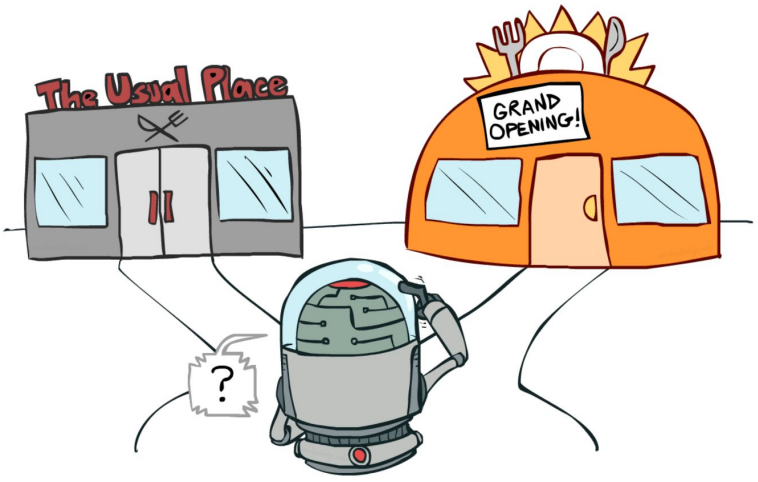
For my project, I implemented Monte Carlo Tree Search via UCB and Thompson Sampling to find optimal moves in a game of tic-tac-toe and programed a light show to demonstrate how MCTS works. MCTS essentially follows these steps:



I created two versions: the first uses the Upper Confidence Bound formula to select which paths to go down and the other uses Thompson Sampling to determine paths. Both of these methods are a way of addressing the exploitation-exploration dilemma when we don't know the payout of each choice. This is commonly known as the multi-bandit problem which gets its name from imagining a gambler choosing which slot machines to play. In our case we don't have slot machines, but instead have paths with unknown results. We want to simultaneously exploit the move with the highest expected payoff that we know



of thus far, as well as explore other moves to get information about their expected payoffs. (I like to think of it as going to as restaurant and deciding whether to order the same dish you've had previous 100 times or trying something new.) Both of these methods tackle the exploration-exploitation dilemma in different ways. The UCB function which I used looks like this:

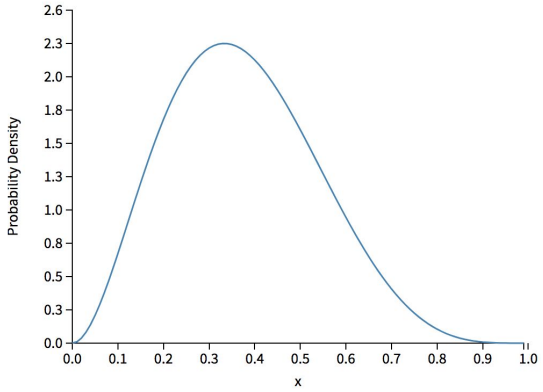
$$\frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

When selecting the next move the move with the node with the highest UCB value is selected. The term on the right represents exploitation and gets larger as the number of wins/simulation gets greater. The term on the right represents exploration and increases when a node is selected less often. The "c" variable can be tweaked to favor exploration or exploitation more.

The other method I implemented in my tic-tac-toe game was Thompson Sampling, essentially drawing from a beta distribution (X ~ Beta(a, b)) to select each move. These beta distributions represent our perception of the world and what we think the expected value of the returns are stemming from that node. Here, X would be the probability of winning the game from that specific node and a & b are the number of wins and failures from simulations stemming from this node. After the selection and expansion, I move onto simulation where I repeatedly draw from the uniform distribution Uni(0,1) to determine the next move till the game is played out. Then I update each visited node in the path. This can be thought as updating the Betas!

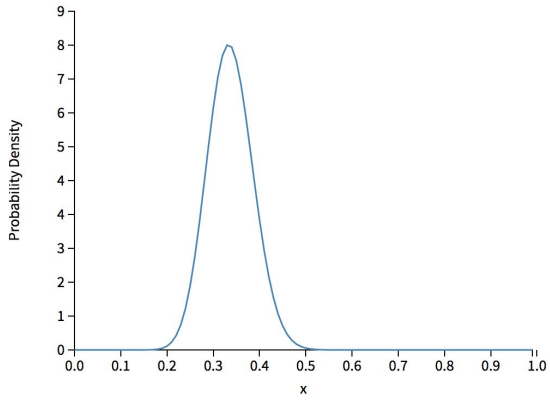
As more trials are conducted (during the simulation phase) the mean of the Beta will shift closer to the true probability of winning and the variance will get smaller and smaller. An intuitive way to visualize how this deals with the exploitation-exploration dilemma is by looking at how the distribution changes as more trails are run.

Below I have graphed the beta distribution: X ~ Beta(3,5). The meaning behind this is that from this node there have have been 6 trials, which resulted in 2 successes and 4 failures.



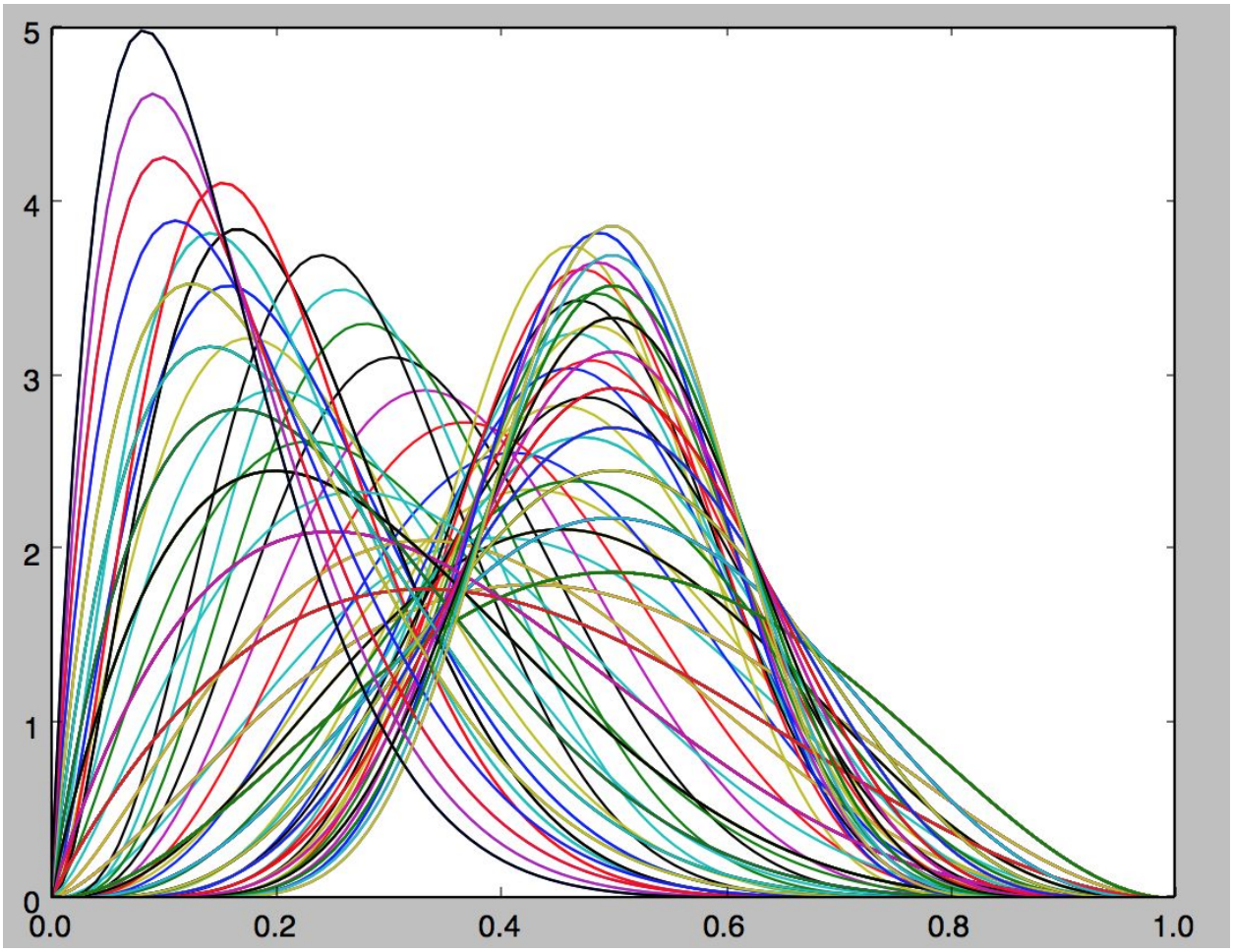
If we were to sample from this distribution there is a significant likelihood that would get a value that deviates from the mean. For example, we could get 0.7 as our value for X. So when this node has not been selected many times before, it's likely we sample a value that deviates from its mean, thus allowing us to explore. Similarly, once many trials have been conducted the beta gets much skinnier, less variation:

X~Beta(31, 61) looks like this:

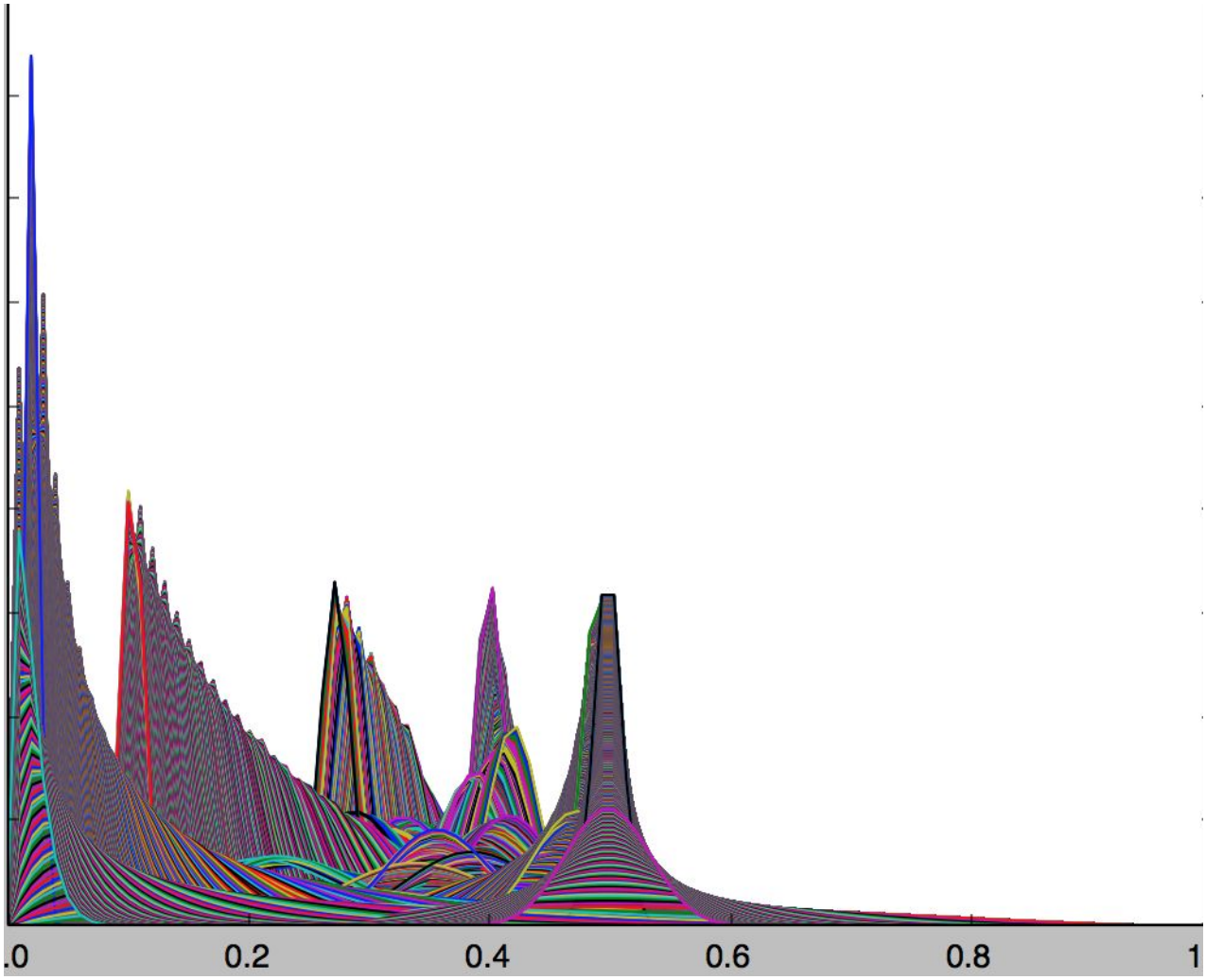


With more trails, when we sample from this beta we are much less likely to get a value that differs very much from the E[X] of the distribution, and this is where exploitation is accounted for. Essentially, by sampling from the Betas, we get to explore. If we simply chose the E[X] of the distribution everytime, our program would never look at other potentially promising paths.

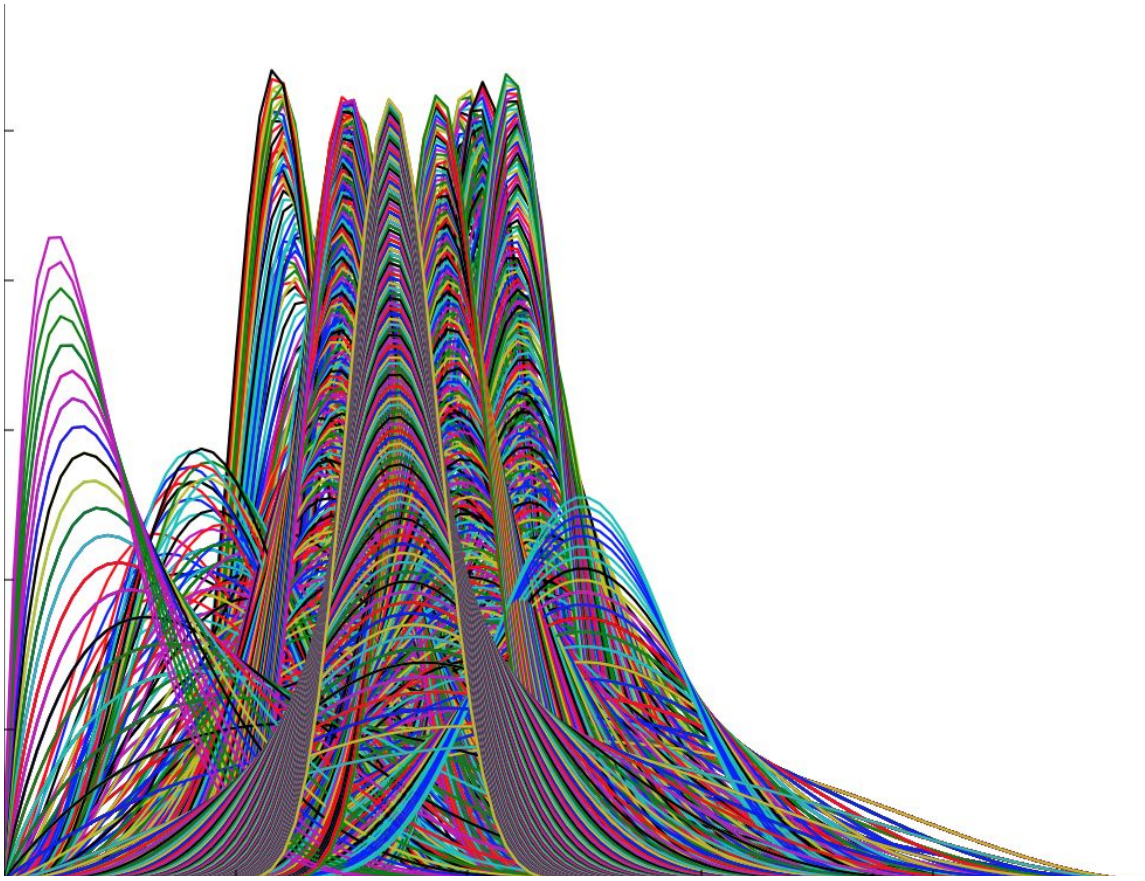
To take this to the next level and visualize this, I graphed the the beta distribution corresponding to each node/state in the tic-tac-toe program I wrote and plotted how the distribution of the probability of winning from that node shifted as more iterations took place. Here you can see how the beta distribution shifted and become skinnier as more trials were conducted. IT'S ACTUALLY SO COOL TO LOOK AT!



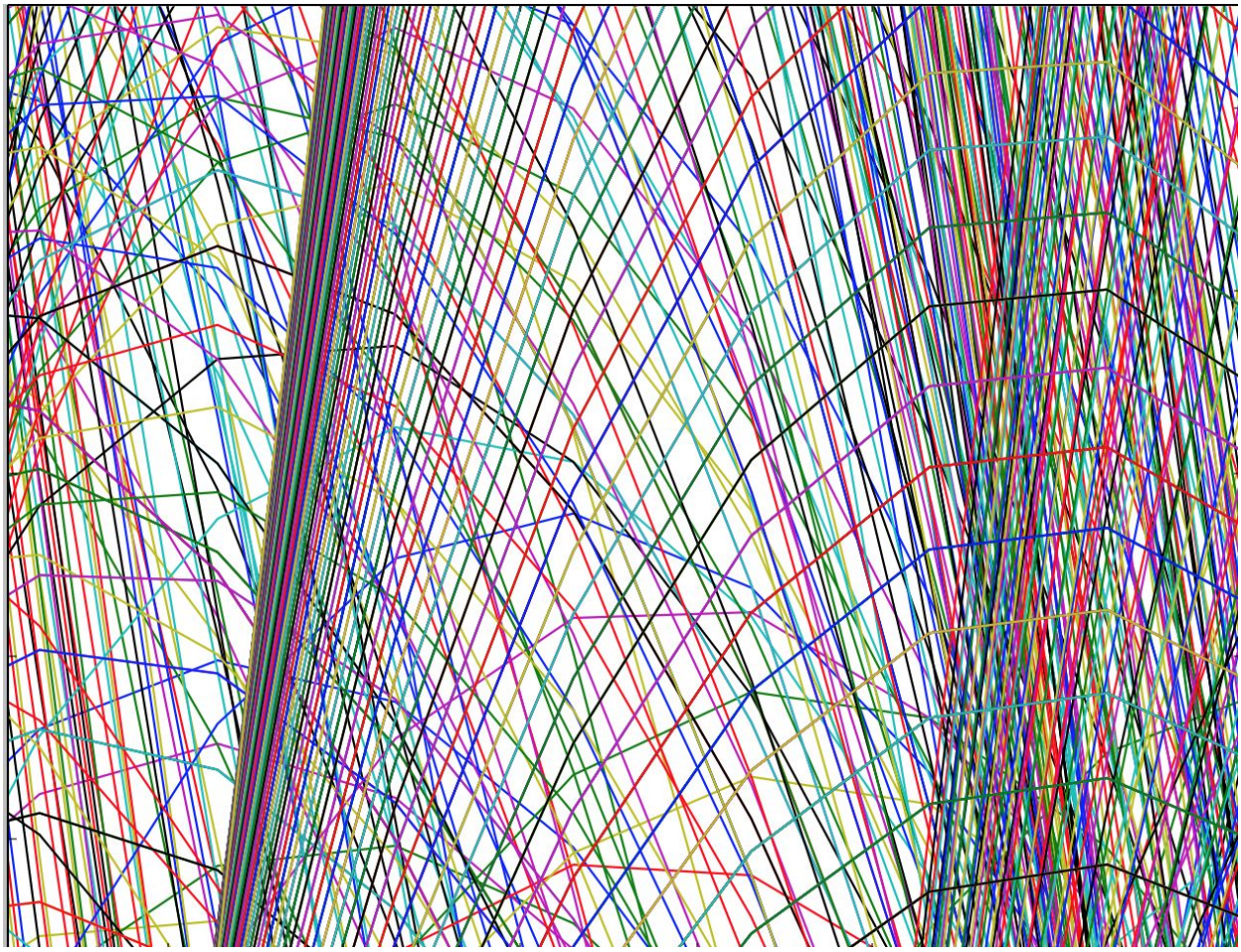
This graph was created with ten iterations for selection of a move, as you can see, the distributions corresponding to each move are not super skinny, since we did not update/run as many trials. You can also see how the distributions shifted as more trials were run.



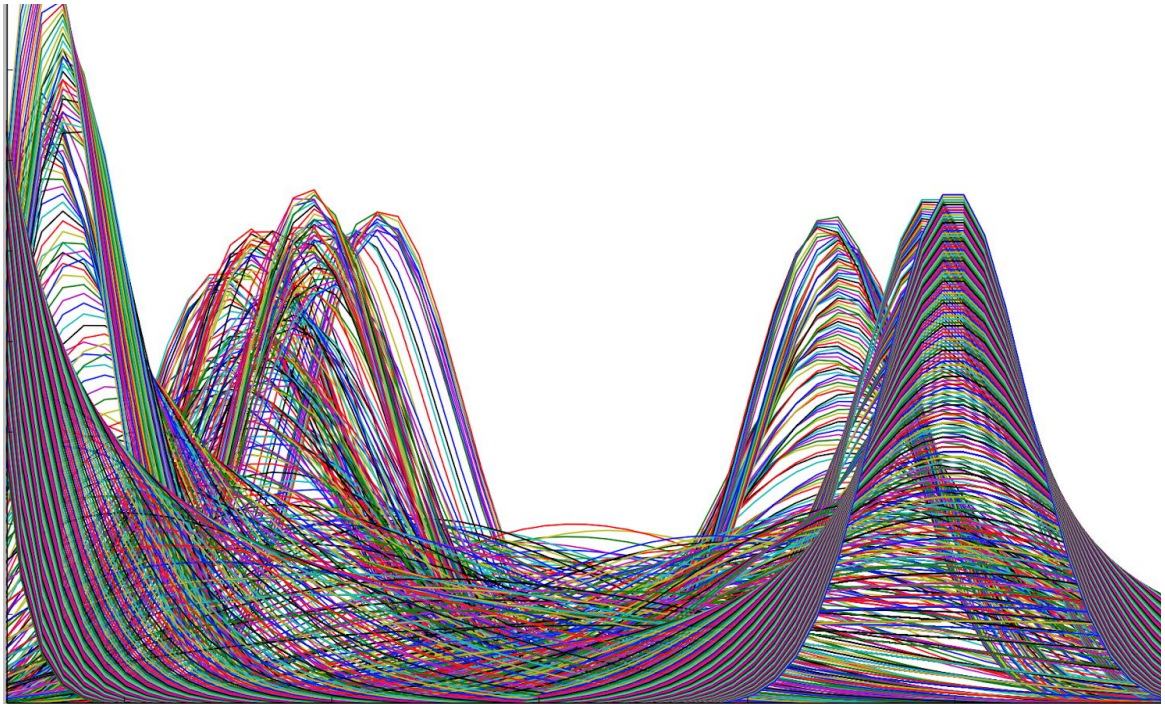
This was after running 1000 iterations each selection. You can really see how the distributions are getting thinner and thinner as they approach their true probability. You can almost derive the true probability of winning from each state since so many iterations were run. SO COOL.

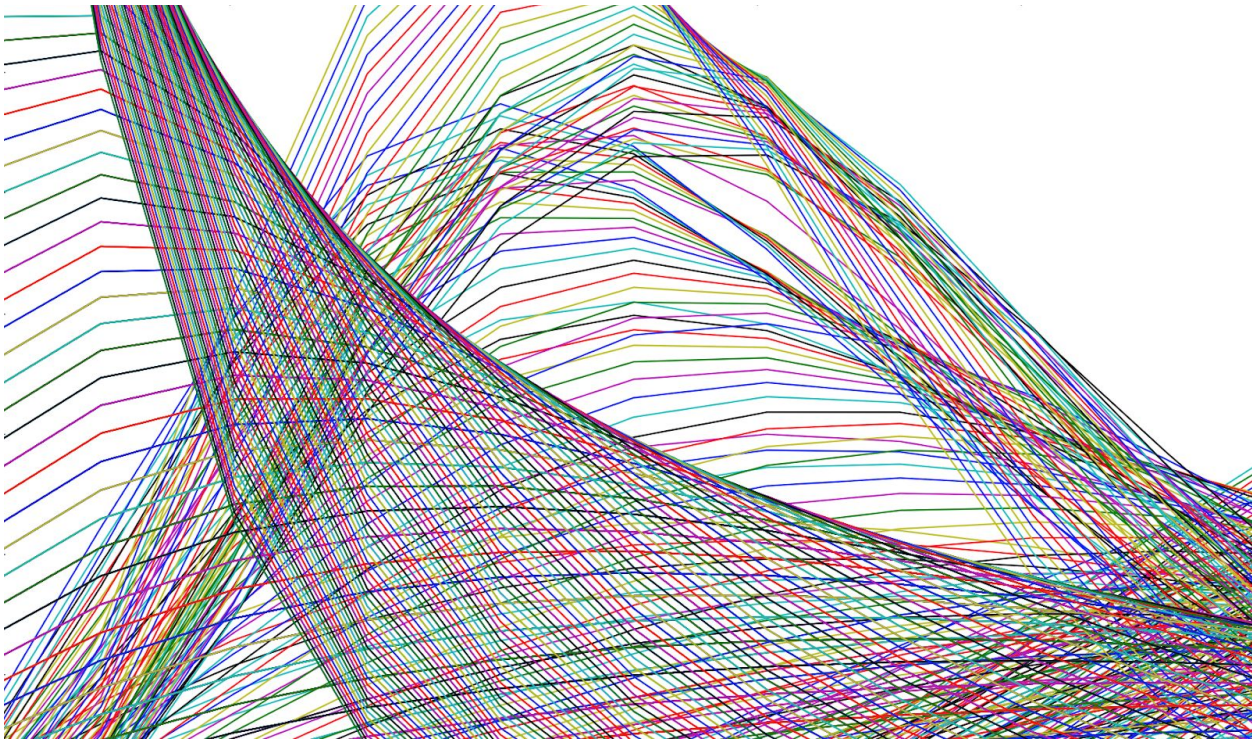


This is for 100 iterations, and after I changed the "c" variable in my UCB function to be 100, making the program value exploration much more heavily.



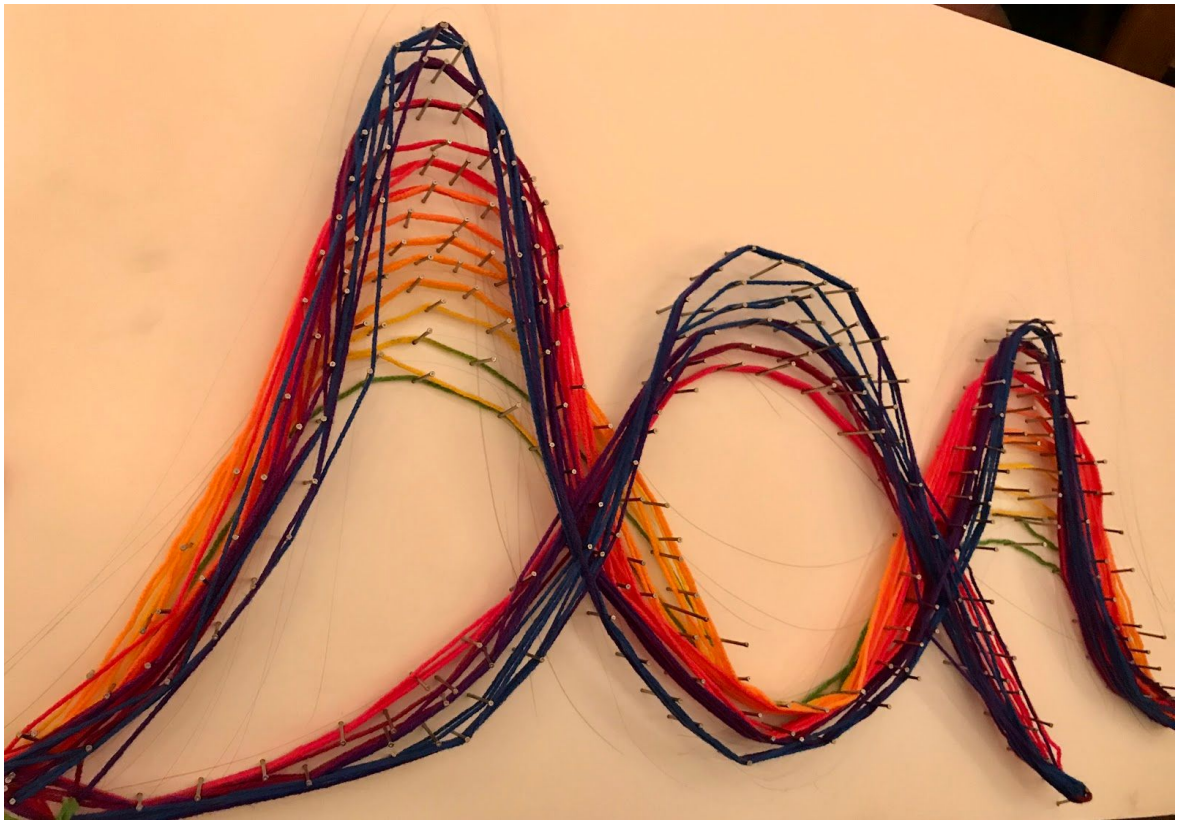
The closer you look, the more you can appreciate how beautiful these distributions are!





Another zoomed in look. I like this one because you can really see how the distribution gets updated every iteration. DISTRIBUTIONS ARE BEAUTIFUL.

I was so inspired by these conjugate distributions, that I wanted to replicate this into tangible art using thread and needles. Here is my finished work below:



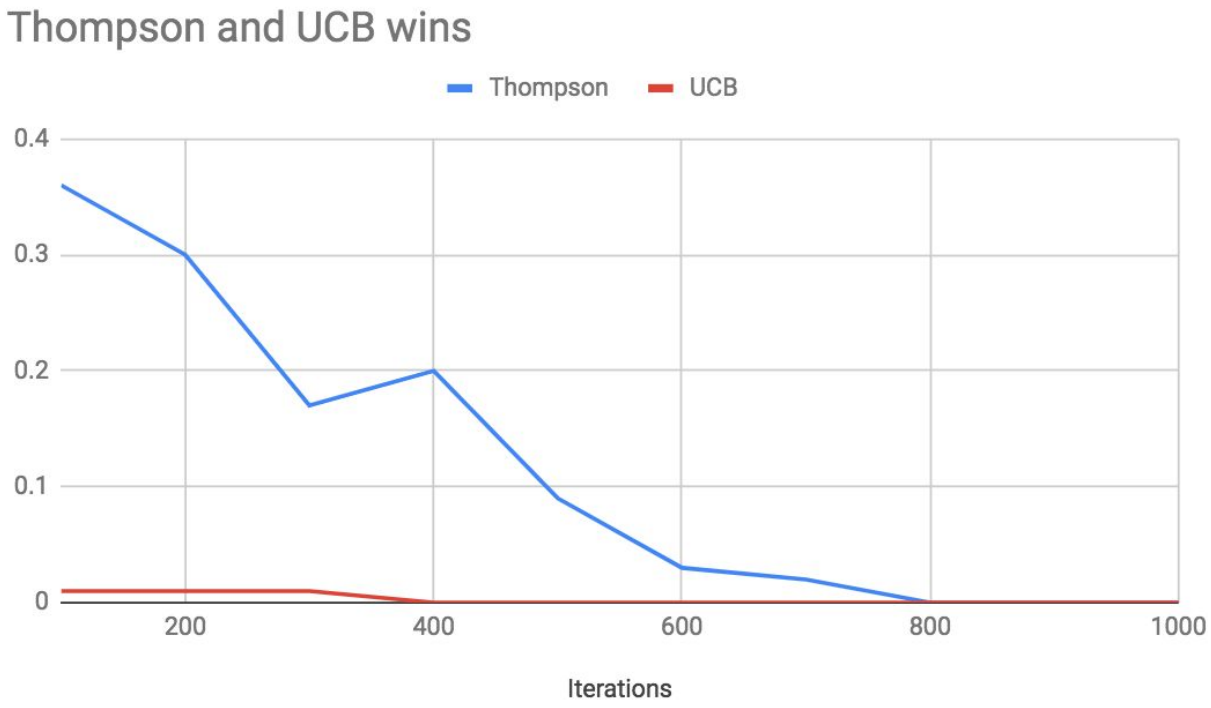
Some differences between the two is that UCB is deterministic whereas Thompson Sampling is probabilistic. When UCB is given the same information, it will always choose the same next move, whereas Thompson Sampling involves drawing a value from the distribution and thus could result in a totally new value being drawn leading to a different path being selected. This key difference has some implications. Firstly, UCB requires us to update at every round, meaning that after we get some new

information we need to backpropagate that new information up the visited nodes before we can make a new decision. On the other hand, Thompson Sampling doesn't need immediate feedback and can still work with delayed feedback. For example, if we could only update our beta's after 500 iterations, we would still be able to use Thompson Sampling for those 499 iterations in between to select new paths by sampling from the betas. This is called batch updating and is less computationally expensive.

While Thompson Sampling is one of the oldest heuristics for dealing with the multi-armed bandit problem, it wasn't researched heavily till recently. After reading research papers by mathematicians like Shipra Agrawal and Emilie Kaufmann, I learned that Thompson Sampling has been found to result in lower amounts of cumulative regret than UCB and other state-of-the-art methods (regret being what is suffered when you make a non-optimal choice instead of the optimal choice, and the longer you explore the non-optimal method the higher regret, essentially opportunity cost). This is because Thompson Sampling takes the whole distribution of each arm into consideration instead of just the upper confidence bound. I wanted to put this to the test in my own program and so I decided to play the Thompson Sampling implementation against my UCB implementation. Both had 500 iterations before they had to select their next move, meaning that they both started on equal playing ground and had equal opportunity to gather information. In tic-tac-toe, the only way for a player to win is to continuously pick the optimal move and wait for the opponent to pick a non-optimal move. If both players only pick optimal moves, then the game will always result in a draw. Knowing this, I ran the UCB and Thompson Sampling programs against each other to see which

was more consistent in selecting the optimal path. After playing 300 games, I found that UCB won 0% of the games, Thompson Sampling won 7% of the games, and there was a draw 93% of the time. Seeing these results led me to believe that Thompson Sampling is slightly more effective in determining the optimal path (at least when there is a sizable amount of information available). Wanting to test how the number of iterations (aka how much information can be gathered) influences how much better/worse UCB vs Thompson perform, I ran 100 games and decreased the number of iterations to 100 for each method. I found that UCB won 7% of games, Thompson won 45%, and there was a draw 48% of the time. I found this super insightful because it suggests that Thompson has a significant advantage at least when information is scarce. Lastly, I simulated 300 games with 1000 iterations for each method and found that when there is A TON of information (1000 iterations is way more than what is needed to explore the small game of tic-tac-toe) Thompson and UCB perform around the same with draws occuring 99% of the time. After reading some more research papers by Bruno Bouzy (specifically "*Old-fashioned Computer Go vs Monte-Carlo Go")*, I found that with enough iterations, MCTS converges to minimax, although very slowly. This would explain the near perfect plays from both methods as the iterations grew.

Here is a graph I made showing how results changed as the number of iterations varied:



Because the holidays are coming up, I thought an especially interesting way to visualize how MCTS works would be through a light show! So I borrowed some programmable

lights and wrote a script which changes the colors of the lights to visualize what is going on in MCTS. You can watch it in my demo video!

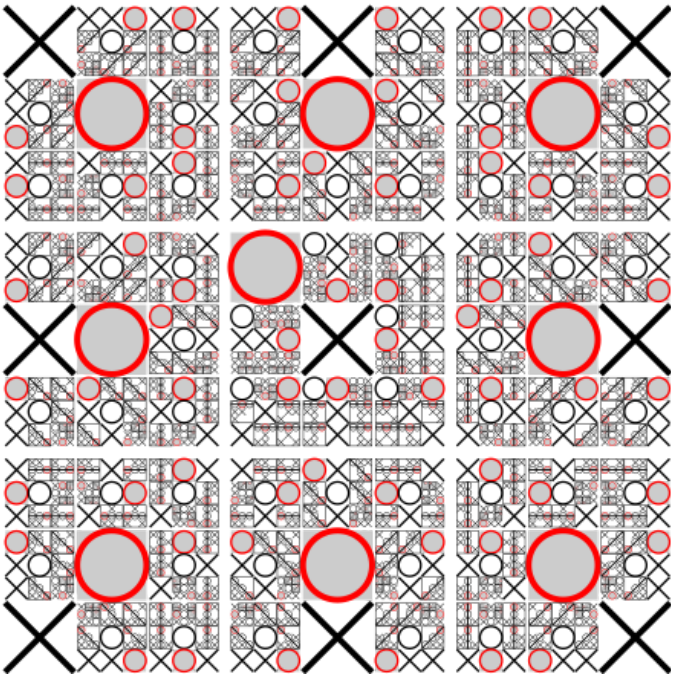
MCTS + Known Tic-tac-toe strategies

Like I mentioned before, something I learned about Tic-tac-toe through researching this project is that the only way to winthe game is to always pick the optimal move and wait for your opponent to make a mistake. If both only make optimal moves, then it will always result in a draw. To see this in action, I ran my MCTS program in computer vs computer mode so that it simulated two players against each other. Just like expected, it resulted in a draw every time!

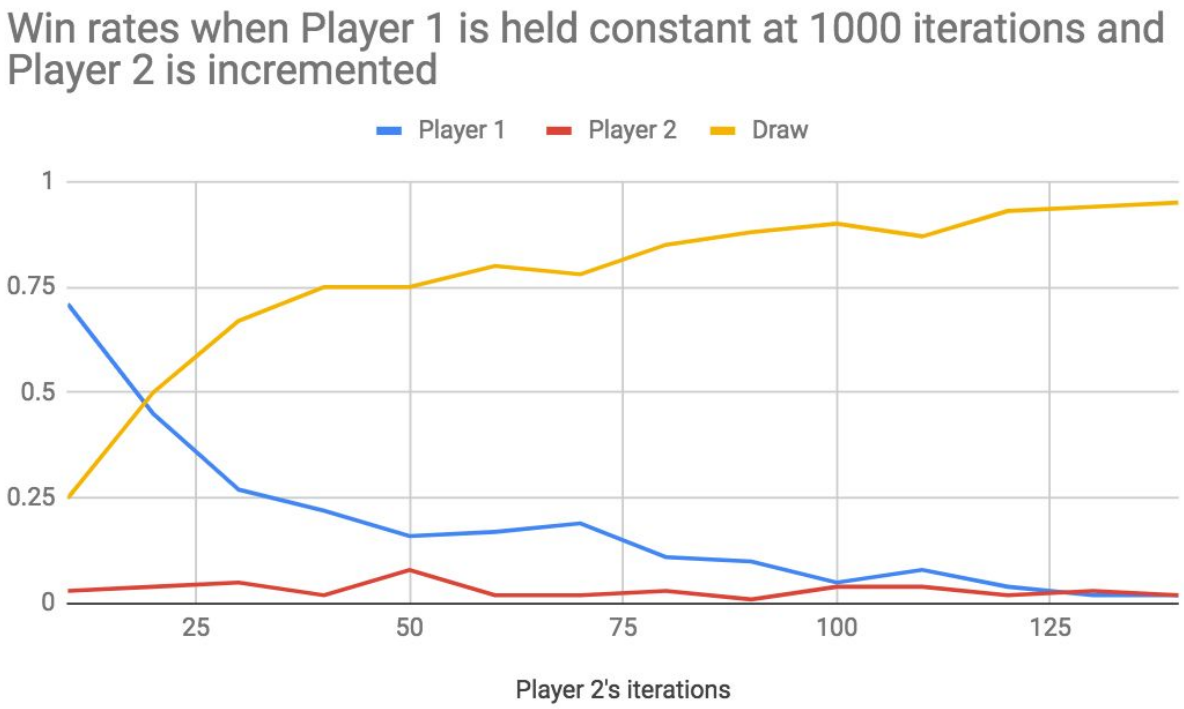
But what really interested me was what would happen if I lowered the number of iterations of MCTS one of the players could go through (thus giving that player less

information before having to make decision) keeping the number of iterations for the other player constant. I decided to put this to the test and had my program simulate 1000 games of tic-tac-toe where player1 had 1000 iterations and player 2 had 250. I was surprised when I saw that that 971 of the games resulted in a draw, and that player1 only won 24 games compared to player2's 5. Since player2 only had ¼ of the iterations of player1, I expected player1 to have a significant advantage. When I thought about this some more, I realized this is likely because the game space of tic-tac-toe is so small that 250 iterations must be above/near the threshold of iterations needed for the program to still make a reasonable decision most of the time. While player 2 did lose 24 times, it still outputted the optimal move 976 times which was super impressive.

This made me curious about what the lowest threshold for iterations should be. After doing some research, I found a book titled *How To Win at Tic Tac Toe* by Ryan Aycock which precisely tells us how to do just that! One of the most interesting things I learned is that if you are the second player, and the only way for you to force a win or a draw is by starting in the center. To make this clearer, here's an illustration:



As you can see, the red O's first play must be in the center (except for the case where X moved there first). To see how few iterations I could give my algorithm before it started outputting non-optimal moves, I decided to play my algorithm at tic-tac-toe and decreasing the number of iterations before it starting selecting a first move which was not in the center. It wasn't until around 70 iterations before it started having trouble. To put test my observations, I decided to run an experiment where I held player1's iterations constant at 1000 whereas I incremented player2's iterations from 10-160 to see how the win rates shifted. I ran 100 games of each setting.



As you can see the rate of wins for both players converges around the 100 mark, suggesting that my implementation of MCTS only needs 100 iterations to determine the optimal move each time before it starts playing as well as the program running on 1000 iterations.

That's all! Thanks so much for reading, and I hope you learned something new! :)

Near-optimal Regret Bound for Thompson sampling