COMP 424—Artificial Intelligence

Noah Lackstein—260524490

April 8th, 2016

When deciding how to design my agent, I first considered the characteristics of the game. Hus is a deterministic game where each player has access to identical perfect information. Because the game is deterministic, the agent can determine the exact state the board will be in at the end of each turn. And because we have access to perfect information about the game, it is theoretically possible to determine the best move that can be played during each turn. It’s therefore possible to repeatedly list all of the legal moves that can be played for each turn, what their effects will be, and what moves can be played in response up until one player wins the game.

With this in mind, I decided to implement a minimax tree search algorithm with iterative deepening and alpha-beta pruning to find an optimal move to play given a certain board state. The theory behind minimax is that you can take a board state, play out all of your legal moves, then all of your opponents legal moves, up until the end of the game, then backtrack up the tree. At each level, your opponent chooses the move that minimizes your probability of winning, and based on that you can choose the move that maximizes your probability of winning given what your opponent will do next.

Unfortunately, the branching factor of the game is large enough that it’s not feasible to play out every possible move to the end state in order to determine whether or not playing it will allow you to win. Instead, I needed to cut the tree search off, which is how iterative deepening and alpha-beta pruning allow me to make the best use of the limited computation time available to the agent. The pruning allows me to disregard sections of the search tree by determining which moves are suboptimal and declining to expand them. Iterative deepening allows me to automatically scale the depth to which I grow the tree so that narrower trees can be expanded more deeply while wider trees (such as when there are many legal moves) remain shallower, all while ensuring I have a decision available within the allotted time.

I then need to use a heuristic to determine the relative likelihood of a move at the bottom of my search tree leading to a win versus other moves being considered. At this point, I was left with a decision between using Monte Carlo tree search or an explicit evaluation function. Given that I have perfect information and that the game mechanics are fairly simple, I decided to use an explicit evaluation function.

My evaluation function is pretty straightforward. I decided on three features for each player, for a total of six evaluation criteria: (1) the number of seeds in the outermost row, (2) the number of seeds in the inner row, and (3) the number of seeds that can be stolen from the other player. If at any point a move results in a win for me, the evaluation function will return the largest possible value. The next obstacle was then to determine the weights to assign to each of the six features.

Since I didn’t have any expert knowledge about the strength of various positions nor could I find any research specific to Hus, I elected to use a genetic learning algorithm to try and find a viable weighting of the features. I accomplished this by seeding a population with twenty phenotypes of randomly determined weightings. I then had every phenotype play against every other phenotype in the population, a total of 190 games for a population of 20, and recorded each phenotypes win percentage. To build the next generation, I used a weighted selection, which gave phenotypes with a higher win percentage a greater chance of being selected, to choose parents from the previous generation, and mated those parents to produce a child phenotype. For each feature, the child phenotype randomly selected one of its parents’ weights for that same feature. When producing a child, there was a small probability that a weight would become mutated, accomplished by flipping a bit in the binary representation. I also took the two highest scoring phenotypes from the previous generation and added them unchanged to the next generation. The children would then play against each other, mate, and repeat, for 50 generations. This, along with the weighted selection function, would help carry the best performing phenotypes forward while the poorer performing phenotypes would be unlikely to mate and eventually die out. Ideally, after enough generations, evolution has resulted in a phenotype that maximizes your evaluation function (in this case, percent of games won).

The algorithm and framework that I used for the genetic learning were adapted from code published by [Matt Mazur on his blog](http://mattmazur.com/2013/08/18/a-simple-genetic-algorithm-written-in-ruby/).

After playing nearly 10,000 games over 50 generations, and repeating that over several simulations, my algorithm tended to converge to a similar distribution of weights on one feature versus another. I ultimately submitted my agent with a phenotype that had been the highest ranked for the previous 10 generations in the longest running simulation.

However, even though that phenotype was the strongest amongst the populations I tested, the risk is that much like evolution I might only have a locally optimal solution. The diversity of the phenotypes was primarily limited to the randomness seeded into the initial population. Even though new randomly generated phenotypes were inserted throughout the simulation, their viability would have been limited by the pre-existing stronger phenotypes. Ideally, I would have liked to run the simulations with much larger populations, with each phenotype playing the others more than once, in order to get a higher quality result.

There’s also the risk that my evaluation function is too simple or doesn’t accurately represent the strength or weakness of my position. I’m only making use of a small amount of the information available to me, which I undoubtedly could have improved on. Also, because it’s possible to capture a significant amount of the opposing player’s seeds in a single move, the entire balance of power in the game can shift sharply and abruptly. This probably greatly reduces the power of a static evaluation function. Admittedly, a stochastic process like Monte Carlo tree search might have been better able to capture this information.

It might have also been possible to capture some of that information by selectively expanding the search tree. If it looked like a significant upset might occur within the next few moves, I could have expanded the search tree deeper for just those nodes.

Another option I considered in order to try and search deeper was maintaining a history of previously seen board states and their respective values. I could then check if the current node in the tree was a simple transmutation of previous moves, and if so then I could stop expanding the tree and return the previously seen value. This is a technique that has been recommended when doing tree search over chess moves. However, moves in Hus tend to significantly alter the layout of the board and hence don’t tend to transmute well if at all.

Despite the risks, the agent is certainly able to make better decisions than an uninformed player such as myself, frequently beating me in just a handful of moves. The modifications to the minimax tree search allow it to search six to twelve moves ahead on average. And while it might have only a simple understanding of the game and of the value of different states, it’s able to leverage that in order to make informed decisions that avoid at least the worst subset of moves. I’d consider that a success for such a remarkably simple agent.