Peter Schaldenbrand

Artificial Intelligence, Dr. Hwa

Project 2

October 18, 2016

There are obviously tons of different moves each player can make at the beginning of the game. When the board is sparse each queen has a lot of room to move around and shoot arrows. The branching factor is huge too because with each move, there are tons more moves that can each happen after it. My program provides a solution to this by using Alpha Beta pruning. This gets rid of branches that are clearly not the best choice for a player so that less work needs to be done by the minimax algorithm. Alpha Beta pruning really helps with this problem since each state has so many possible successors. With this pruning, we might not have to calculate a value, or visit the successors of many nodes.

In addition to the issue of huge branching factors, there is also the time limit. My program tries to solve both of these constraints by having an iterative deepening search. The program searches all the possible moves that player one can do then figures out the best one. If it has more time, it will move the depth down to two, and now it will figure out all the possible moves that player one can make then player two can make. It then uses minimax with alpha beta pruning to find the best solution path. The depth keeps increasing until it comes near to when the time limit is up. It then returns the best move from the deepest completed search. This provides the most thought out solution to the problem given a time limit. There might be a lot of work wasted since once you finish a certain depth, the previous depth is useless, but since you can’t be sure that a certain depth will complete a search in time, this is the safest way. It’s also not so horrible as far as time wasted since with each depth, the problem grows exponentially, so the time it takes to do minimax on depth=5 makes the time to do depth=4 look tiny.

I tried a few different heuristics in the course of making this program. The heuristics determined how good a board configuration was to the player. The simplest implementation I tried simply figured out how many open spaces were around the player’s queen, then subtracted the number of open squares around the opponent’s queen. This proved to not be a very successful heuristic. The final one I settled on figured out how many possible spaces that the players queens could make, then subtracted the total number of spaces that the opponent’s queens could make. The idea was that you wanted to maximize the number of open spots around you, and minimize the number of spots around your opponent. This is still a pretty naïve heuristic function. The game is complex so it’s difficult to simply say what works and what doesn’t.

My program is decent at this game. It beats me around 75% of the time, although I’m not very good at it. It seems to do better in the beginning of the game than in the end, but I haven’t been able to figure out why.