**Project 3 Report: Isolation**

Ethan Liao and Joon Chan

*Last Modified: 5/5/2019*

**Objective**

The purpose of this project is to implement a two-player time-based game of Isolation. This implementation uses the minimax algorithm with alpha-beta pruning. The program also includes a unique evaluation method to make sure the computer will make proper moves while balancing safety and aggressiveness.

**Overview**

The Minimax Algorithm is a popular method for adversarial search problems in a multiplayer setting. It is rooted in the idea of a evaluation of any state in the game where one player wants to maximize the score and the other wanting to minimize the score away from the first player. The algorithm helps to calculate which move, in any given state, would give the most value in aiding victory. It uses an alternation of max and min steps where player A would try to find the best move expecting that the player B will make the, in theory, “worst” move possible against player A’s goal. This technique allows for evaluation of future predicted states in attempt to bring the game to a more favorable position as the game proceeds. A tree graph can be drawn to mark the various possible paths at any point in the game. Each path is given a score depending on the evaluation function, the higher the more valuable. The alpha-beta pruning method is a technique to allow faster computation of the minimax algorithm. Since there is very large number of paths to take, it would take an extravagant amount of time to check single possibility. By using an alpha and beta value, the algorithm will have a minimum standard to look for. The alpha value is the highest score the algorithm has found for any path for max and the beta value is the highest score along the path for min. If the beta score is lower than alpha, no higher score than beta can be found, and thus there would be no need to search down such path any longer. This way, many unnecessary paths are pruned away so the calculation is quicker. To further shorten the time needed, especially in games with a high branching factor, iterative deepening method is also used. Iterative deepening is a form of searching which limits the depth, or how many moves later, of each path the algorithm will explore. This allows for the algorithm to specify how many moves to foresee instead of endlessly calculating through a path that may not even matter.

**Implementation**

For our evaluation function, at first, we thought of a heuristic that would return the number of valid moves for the MAX (X). However, as we thought about it, it wasn’t going to be a good evaluation function because if it plays versus an aggressive AI, then it will just try to survive until it was killed. If we had a fully aggressive AI, then it would make reckless moves and potentially just make a move that will allow the opponent to trap them. We needed to balance a mixed heuristic between the defensive moves and aggressive moves. Calculating the defensive score was pretty straightforward. we settled on the aggression constant of 27 because the most moves you can possibly make from one position is 27 (if the board is cleared and you are in the middle). Whatever value we obtain from calculating the number of moves the opponent can make is subtracted from the aggression constant. Therefore, we have a score for defense and offense and simply sum it together.

**Debugging**

There were a ton of debugging issues for this project. For starters, when creating and testing a lot of the functions for the Board class, it was often hard to iterate all of the possible moves and it was hard to keep track of the bounds so there was a decent amount of ArraysOutOfBoundsException hit. The same went for generating children nodes of the Board. Implementing the UI the way the project specifies proved to be a challenge as well. If it was just the Board followed vertically downwards with the list of moves, then it would’ve been straightforward. However, it was specified that the list was to be directly to the right of the board, which took a lot longer than expected. There were just so many conditions to keep track of, but eventually it was able to work.

At first, the heuristic we used was not optimal. We were playing against the AI and it seemed to always fall for the same traps (or lose to counting), or even worse, trap itself. However, after we settled on the heuristic (see above) with balancing aggressive and defensive options, then the AI started making more balanced, good moves.

For the alpha-beta pruning algorithm, it seemed impossible to get it to work at first. We needed to make sure the implementation was set first before we touched the algorithm at all (make sure all the methods were bug-free and void of exceptions). After what we thought was a correct implementation, we ran into the first big barrier. There was no way we could tell which path to take, because the algorithm returns an evaluation and not necessarily a specific path. Also, it was very hard for us to be able to return the proper action after we found the best move. We attempted to use various methods such as comparing the best score to each child’s evaluation score or keeping a list of scores for each of the grandchildren. However, none of these methods were successful in solving our problem, after a long period of trial and error, we finally stumbled across a solution using a static board variable to store the desired children board.

Trying to solve the algorithm implemented with a time constraint and iterative deepening proved to be a challenge as well. It was difficult because of the callback of the recursive functions since we can’t stop the recursive function while it is constantly calling back. Only when it is going through and generating children can we stop it with the code. This made the time constraint difficult because we don’t want it to be calling back the terminal states while the time limit is being hit. Overall, we learned so much through this project and it was extremely fun to work on. Thank you!