Student Name: Robin Farrow-Yonge
Student Number: 160719011

Artificial Intelligence Coursework 1 - Report

The goal for this implementation from the outset was to produce something simple that would play reasonably efficiently with a quickly-executing evaluation function, allowing deeper search. I was not entirely successful in this approach - getting the thing to actually work initially was troublesome enough. As it is, it can safely search to a depth of four. With a little more optimisation it might make five, but in its current form it times out a little more regularly than I'd like.

Code Overview:

MoveScore - I opted to define a subclass to store Move objects and their associated scores, making it easier to pass them back and forth.

chooseMove - The only purpose this function serves is to take note of the player colour and base the call to alphaBeta on that. White always plays first, and so in this implementation if my player is white I'll call alphaBeta with max set to true and vice-versa.

prepareMove - PrepareMove iterates over the current board state, creating MoveScore objects for each index holding a null value. These are returned as an ArrayList representing all the potential moves.

alphaBeta - This function runs based on three conditional statements, these check whether the depth counter has hit zero yet (in which case it evaluates the state and passes the results back up), then whether or not the current player is max or min. After this point, the alphaBeta function runs in an entirely unremarkable manner.

eval - Eval takes a board state, and returns a score for it based on what is returned from scorePatterns and getPatterns. The only interesting thing it does is the addition of a multiplier applied to the opposing player's score. This multiplier is supposed to serve as a way to encourage a more defensive playstyle, after previous versions failed to prioritise blocking the opponent's winning move over setting up their own a couple moves ahead. This problem might have also been solved by checking for terminal game states, but here we are.

getPatterns - This just uses a switch statement to check all the squares around a given square, looking for rows of any length. The length of the rows that are found are added to an array.

scorePatterns - ScorePatterns checks every index in the array of patterns/rows, cubes that value, and adds it to a variable representing the score for that square, which is in turn returned to eval.