CONNECT-K FINAL REPORT

Partner Names and ID Numbers: Shawn Williams, 79146390 and Alvin Liang, 70314204

Team Name: ColgateAI

1. Describe your heuristic evaluation function, Eval(S). This is where the most "smarts" comes into your AI, so describe this function in more detail than other sections. Did you use a weighted sum of board features? If so, what features? How did you set the weights? Did you simply write a block of code to make a good guess? If so, what did it do? Did you try other heuristics, and how did you decide which to use? Please use a half a page of text or more for your answer to this question.

For AI project, creating a good heuristic function proved to be the most challenging part. Writing the code for the function was trivial, what was hard was finding the right logic to play Connect-K. Initially during the early stages, we implemented a simple function that tried to put K pieces of the same color together. After we implemented alpha beta pruning and iterative deepening search we started doing more research on how to win Connect 4. I came across a paper by Victor Allis titled "A Knowledge-based Approach of Connect-Four" where he shows that Connect 4 can always be won by the player that goes first. I attempted to implement the algorithm and strategy he described in his paper, only to find that it was not applicable to Connect-K. Connect-K is a rather bad name for our project, as when played without gravity it more closely resembles a version of Gomoku. As it would turn out, Victor Allis also authored a paper titled "Go-Moku and Threat-Space Search" where he proved that Gomoku was a solved game. He described a strategy called threat-space search, where the AI would build up threats and win once two threats were built. We attempted to implement this strategy but found that it was beyond the scope of our programing skills. Instead we used Allis' paper for inspiration and implemented a strategy of blocking the opponent from making threats. We have no proper proof of this, but it seems if we block every attempt of the opponent from making threats, we can always force the game to come to a draw. We implemented code that does attempt to win the game, but we found it hard to attempt to both win the game and block the opponent at the same time, so we weighed blocking the opponent more. Our final heuristic function attempts to build up threats of its own pieces, but prioritize preventing the opponent from building up threats. This results in a best case scenario of winning when it built up threats of its own, and a worst case scenario of forcing a draw.

2. Describe how you implemented Alpha-Beta pruning. Please evaluate & discuss how much it helped you, if any; you should be able to turn it off easily (e.g., by commenting out the shortcut returns when alpha >= beta in your recursion functions).

We extended our minimax algorithm to assign a score to each move using our heuristic function and prune moves that did not improve our current score. The parameters taken in by alpha beta pruning included: BoardModel state, int depth, double alpha, double beta. We initialized the variables alpha to -infinity and beta to +infinity. For each move, we assigned alpha and beta values by comparing the move's score to alpha or beta (comparison depended on whether we were inside minimizingValue or maximizingValue recursive functions). If the move was better, we saved the score and point coordinates to return later. With alpha beta pruning we improved our search up to depths 4-6 compared to depth 3 with standalone minimax before timing out. The reason for minimax's sluggishness was its attempt to search up to the depth cutoff for every value, whether it was useful or not, which wasted time and space. The alpha beta pruning algorithm was much better in terms of performance and could beat PoorAI if our AI went second and tie otherwise.

3. Describe how you implemented Iterative Deepening Search (IDS) and time management. Were there any surprises, difficulties, or innovative ideas?

We implemented iterative deepening to search as deep as the time limit allowed for. We used Java's System.nanoTime() to give the most precise timing. Under a while(true) loop, the program ran alpha beta pruning on every available move and increased the idsCutoff variable by 1 to search deeper in the next iteration. In each recursion of alpha beta pruning, we passed in depth + 1 to maximizingValue and minimizingValue functions along with an idsCutoff variable. In our terminal conditions for alpha beta pruning, we compared depth <= idsCutoff to ensure the algorithm did not search farther than the iterative deepening cutoff and compared the algorithm's runtime with the system's time to not exceed the time limit.

The difficult aspect of implementing IDS was figuring out how to save the previous value and where to check cutoff time. We resolved this issue by creating a variable that stored the index of the best move in our ArrayList so we could find it later. We discovered we had to check the time at the beginning of each iteration and inside our alpha-beta pruning functions to make it fail proof and account for buffer time in case the program ran longer than expected. If we were to reimplement IDS, we would incorporate a priority queue and sort coordinates from the center of the board. Another implementation would have been a HashMap or HashSet to remember and access the best move with an O(1) operation.

4. Describe how you selected the order of children during IDS. Did you remember the values associated with each node in the game tree at the previous IDS depth limit, then sort the children at each node of the current iteration so that the best values for each player are (usually) found first? Did you only remember the best move from a given board? Describe the data structure you used. Did it help?

We used a function to generate a list of all the available moves on the board. We looped through that list and ran iterative deepening search to store the best move from the game state. We initially used a vector to store the list but we modified this to an ArrayList for better performance after learning that vectors were synchronized functions and more suited for multithreaded applications. We could have improved our search by using a priority queue to sort by moves centered on the board but our main priority was debugging and research on improving our heuristic function. We thought ArrayLists were a decent choice for the time being since we would have to loop through all the available points even with a priority queue (we would get the best move faster however we had issues with our heuristic) and since we were only using add and get operations to run IDS. By changing to an ArrayList implementation from Vector, the time saved was about .3 seconds faster, however, there was no difference in AI performance.