CONNECT-K FINAL REPORT

Partner Names and ID Numbers: Hui-Yim Tong (56271961), Linh Chu (10473576)

Team Name: Coding Cats

Note: this assumes you used minimax search; if your submission uses something else (MCTS, etc.), please still answer these questions for the earlier versions of your code that did do minimax, and additionally see Q6.

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 our heuristic evaluation function, we first wanted to see what was a good strategy to determine whether our game play was good or not. We determined that the more groups of our own pieces we had, then our evaluation would be better. We defined groups as being any row, column, or diagonal "in-a-row" pieces that we had. We decided to do this because we want to find a way to extend our longest available grouping in order to win against the opponent. The longer the grouping, the better it is because it will be closer to winning the game. Thus, we used a weighted sum to calculate the value. For groupings that had spaces on either side of it to add to, we would cube the number of pieces in the group and add it to the value of the function. For groupings that had only one space available to put a piece down to extend the group, we would square the number of pieces in that group and add it to the value of the function. For groupings that had no available pieces on either side of the group (meaning the opponent blocked the group or it is at the edge of the board), then we would simply add the number of pieces in that group to the value of the function. Since the function value will be exponential if it is doing well, it's easy to determine whether or not one player is better than the other. We also decided to disregard groupings with only one piece since we're interested in groups that have more than one piece to continue extending on. After adding up all the values found from the groupings, we would divide that value by the number of groupings we had in total. Our heuristic function also takes into account of the opponent's heuristic. We use the same way to determine the "goodness" of the opponent's game play and subtract it from our player's heuristic value.

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 implemented Alpha-Beta pruning by making an alpha-beta function that would set alpha = negative infinity and beta = positive infinity every time it was called. Then we use a for-loop to get all the possible moves and apply them for each player's turn. If it was the player's turn, then it would check if the move's heuristic value is greater than the previous move. If it is greater, then set the variable that was keeping track of the best move to be equal to that move that was just checked. Then we would compare the value of that variable to the current value of alpha and take the maximum value between the two and set it to alpha. If it was the opponent's turn, then we would check if the move's heuristic value is less than the previous move. If it us, then we set the variable keeping track of the best move to be equal to that move that was just checked. Then we compare the value of that variable to the current value of beta and set beta to the minimum value between those two. After determining the values of alpha or beta, we would then check if alpha is greater than or equal to beta. If it is, we immediately return the best move and move on to the next player's turn. We recursively call the alpha-beta function when every move is made/applied. Alpha-Beta pruning helped to speed up the process of determining what the next best move was. It significantly saved a lot of time because without it, our AI would take a much longer time to make a move.

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

We implemented Iterative Deepening Search by using a for-loop to keep track of the depth of the tree. For every iteration, we would set the maxDepth variable to the current depth, increased the depth with every iteration, and find the best move within that depth. Regarding time management, we would also have a timer that would check how

long the AI was looking for the next best move. If we would check the time against the deadline. If the time was up, then we would break out of the for-loop and immediately make the best move that we found within that time. We did run into some difficulties. One problem we had was that when the timer was up, it would return the best move found before it actually finished searching through the whole tree. This would cause the AI to make the wrong/"bad" move. To fix this problem, we would save the best move that was found from the previous iteration if the current iteration was not done.

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?

Whenever depth was equal to 0 and our queue was not empty, then we would sort. We would also save the moves in a sorted queue (sorting from largest to smallest heuristic value for alpha and sorting from smallest to largest heuristic value for beta). Then we would do recursion through the sorted list of moves to prune more. This helped to reduce time to find the next best move since the best move was usually one of the moves that were checked first.

5. [Optional] Did you try variable depth searches? If so, describe your quiescence test, Quiescence(S). Did it help?

6. [Optional] If you implemented an alternative strategy search method, such as MCTS, please describe what you did, how you implemented it, and how you decided whether to use it or your minimax implementation in the final submission.