CONNECT-K FINAL REPORT

Partner Names and ID Numbers:
Steven Tran 83611188
Valeria Vikhliantseva 12831698

Team Name:
Team TrashcanDumpster

1. Our AI used two blocks of code for our heuristic evaluation function: 1) one that checks threats of the other player winning and whether there is a location on the board that would allow our AI to win (with a larger priority on the winning position) and 2) a block of code that checks a theoretical position and gave a weight to each location of the board based on whether there is a chance of winning to either the AI or the opponent, and subtracting these values from each other in order to give a value for our minimax function to work with it in finding moves. Throughout the course of our AI, our group tried multiple heuristic functions in order to find a better way in trying to find the best move. We tried assigning weights based off of whether there were two in a rows, three in a rows, and four in rows, assuming that five in a row was the winning condition, but it resulted in failure, and this would simply result in the AI placing pieces closer to the right edge of the board inexplicably. We also tried assembling weights for the board in other ways, but they always resulted in breaking our code, so we stuck with what we have now which is similar to the winning function in the java code given to us by the professor. In our first block of code, our threat checker checks the board for any streaks of pieces of the opponent and AI that are in a row and chooses/blocks those streaks in order to win (it will not block traps such as three-in-a-row in a connect-five game, but it will at the very least block any possible winning positions or choose the position that allows it to win). We tried implementing a way for our code to block three in a rows but failed to figure out a way to without creating an AI that just places blocks semi-randomly (the code would find any group of threes regardless of whether or not they were in a row). It was difficult in preparing a heuristic evaluation function that would work and we tried to change it several times, but our original one was the only one that worked best.

2. We implemented our Alpha-Beta Pruning by using the pseudo code from our textbook. The code starts with a given alpha and beta of max and min integer values and through the course of the search, these values changed to reflect whether or not pruning is necessary. Our code was also recursive, which streamlined the process and kept it simple and easy to understand by looking at it, and both our max and min were entirely implemented in one function rather than separately, which made writing the function a bit harder due to having to change what is returned and keeping track of everything, but it was successful in the end. The process of Alpha-Beta Pruning helped our code by visibly making it faster when we implemented our code the first time around (we set up a timer and tested how fast it placed pieces down at certain depth and there

was a very sizeable difference), but was less visible when we implemented IDS sorting, because we had to take into account the time management part of the AI. We can easily cut off alpha-beta pruning by commenting out the if-statements that prune the min-max earlier (if bestmove >= beta and if alpha >= bestmove).

3. When implementing our iterative deepening search and time management, we ran into quite a lot of trouble in figuring out how the deadlines worked and the passing of the times back and forth into the minimax searching. Also, while implementing the code, keeping track of the necessary checks to see if the deadline was reached caused a lot of confusion. We implemented IDS and time management through a while loop, which checked each depth of the game tree to find the best move, saved the best move, and searched the next depth to see if anything was better. Time management was used in this while loop and ended it if it went over the deadline - we used C++'s clock functions and found the current time when IDS started as a baseline. This start time was passed into the minimax search and ended the search abruptly if the deadline was reached (by checking the current time after every recursive call and checking whether it is over the deadline), by returning a best move heuristic that is extremely long and then placing a check for that specific heuristic, in order to be able to escape the recurrence in time. Initially, this check provided us with problems (the first time we turned IDS in), because we would return NULL as the bestmove, but forgot to get out of the recurrence, so it would continue to use NULL as a heuristic instead of escaping the function, resulting in the minimax to occasionally return 0,0 as a move (even if 0,0 was already taken up).

4. For our IDS sorting, we created a function that would take in the previously created vector of possible moves and place the best possible move found in the previous iteration at the front, because sorting the vector completely would take too much time and wouldn't increase the speed that much, and having the best move first would theoretically still provide the biggest speed increase. We did not really notice a big difference with the best move switched to the front of the vector, regardless.