CONNECT-K FINAL REPORT ***[TEMPLATE --- do not exceed two pages total]***

Partner Names and ID Numbers: Pok On Cheng (74157306) Mengqi Li (92059150)

Team Name: HiroGO

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.

In this project, our evaluation function acts the most important role in determining each valid move's score for being used in Minimax. Suppose the opponent has made a move in the game, so that our AI needs to figure out the best move to make by using the search tree approach. The evaluation function will only generate the part of the search tree that originates from the current game state. Also, it may take more time at the early in the game because there might be having a huge possible valid moves set to estimate.

In the process of searching, the AI will generate as much of the relevant subtree as is practical, using the resulting game states to guide us in selecting a move that we hope will be the best.

In the Evaluation class, we created a copied board for the estimating moves. First, evalFunc() will call the another in-class function, boardCheck(), to count every piece on the board. Second, the evalFunc() will call the GameStateChecker class to check all directions of the move we are estimating and we give the move a score to rate its priority in the list, so that we can efficiently pick the best move in the turn.

After getting the evaluation value, it will pass it to the class, MinMax, to perform the next step estimation. In our program, it will perform a sort of depth-first search on the game tree, which we use a recursive method to perform the search, negating the need to actually build and store the game state to the struct in Move.h, called MoveInfo. Furthermore, the depth (dp) is to limit the depth of our search, so that the program don't have to simulate all situation till the end every time.

At last, we also consider the gravity issue, we simply use a if-else clause to perform both situation of having gravity or not.

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).

Alpha-Beta pruning helps on reducing AI "thinking" time sometimes. Because it depends on the order in which children are visited. If children of a node are visited in the worst possible order, it may be that no pruning occurs.

When we are searching for the max value, we can just visit the best child. Thus, we won't waste our time on exploring weaker child. In the other hand, when we are searching for min value, we can check the worst child first.

After the program pruned the brunches out, the searching performance is actually improved.

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

We were surprised that iterative deepening search is faster than depth-first search.

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?

The Program starts with a one ply search. And then, it will increase the depth for searching. The process will be repeated the process time reaching the time limit. We used a struct which is built in the Move.h. It includes the col and row of the move, and also the evaluation value and the gameState. It helps a lot to double-check the status and we don't have run the evalFunc() again to obtain the value.

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.