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

Partner Names and ID Numbers: Maximilian Hung (81431912), Phong Huynh (21837531)

Team Name: Team Maybe

**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 the dot product of a vector of weights with a vector of features? What features? How did you set the weights? Did you simply write a block of code to make a good guess? What heuristic did you use? Please use a half a page of text or more for your answer to this question.**

Our heuristic iterates over the board and builds chains to track how each player is progressing. Chains for each player are stored to be considered when evaluating the state of the board: longer chains are scored higher, and boards with longer or more chains are preferred. For each chain the AI owns, the score of the BoardModel is increased. For each chain the opponent owns, the score of the BoardModel is decreased. If there is a winning move for us, we score it the highest possible value to force the AI to take the move, while we score a winning board for the enemy the lowest possible value to avoid that board.

We use an Iterative Deepening Search to simulate potential moves that the AI and its opponent can take. We simulate the AI and the opponent’s moves at differing levels to capitalize on Alpha Beta pruning and try to have the AI make moves that grow the AI’s chains and avoid moves that would help the opponent increase his chains.

We track chains building an ArrayList of HashSets of Points. At each index of the ArrayList holds a HashSet that lists the Points in a chain. Our evaluator goes point by point over the entire board and checks if that spot is occupied by an AI piece or an enemy piece. Depending on who the piece belongs to, the evaluator will check the spaces around that piece to see if another piece that belongs to the same player is adjacent to the piece we are currently evaluating. If it does find another piece, the evaluator will continue in the direction the new piece was found to see how long this chain of pieces stretches. Once it finds the end of the chain, it will store all the Points the evaluator traversed to create this chain into a HashSet. Four HashSets are created to store points in the four possible chain directions. These four HashSets are then stored in an ArrayList that belongs to whichever player the chain belongs to. This ArrayList of chains will be used to determine the worth of the current BoardModel.

The worth of a board scales is calculated by having the evaluation of the board increase with multiple and longer chain that belong to the AI, while having the evaluation decrease with multiple and longer chains that belong to the opponent. The amount the score of the board increases or decreases scales exponentially with the length of the chain. Because of this, longer chains have more weight on the score of the board.

**2. Describe how you implemented Alpha-Beta pruning. Since you put it on a switch so that you can turn it on and off, please evaluate how much it helped you, if any.**

We implemented Alpha Beta pruning by checking the evaluation of each BoardModel with and pruning boards that are not as favorable to the AI as another board. We set the initial alpha and beta values to the maximum and minimum integer values to reflect positive and negative infinity. As we evaluate each board, we update these values to track which moves to take and which moves to avoid.

While switching Alpha Beta pruning on and off, we found that with the pruning on, the AI was able to search deeper down the game tree than with pruning off. The AI performed slightly better with Alpha Beta pruning on than off because it was able to evaluate more boards in the given amount of time.

**3. Describe how you implemented Iterative Deepening Search (IDS). Were there any surprises or difficulties?**

For Iterative Deepening Search, we started with a depth of zero to see what possible moves were available to us. After evaluating each move, we finish the Search so that we at least have a move to give once the turn’s time limit has been reached. Then we proceed to the depth of one to see if the moves that we analyzed at the previous level could lead to a better state than the possible moves from the previous depth limit. If we find a better outcome, then we change the move we intended to return to the move that would lead us to our new best state. We continue to move deeper down the search with increasing iterations, revising our intended move as we analyze more and more BoardModel into the future until we run out of time. At this point, we make the move that we found leads to the best BoardModel that we had analyzed before the deadline was reached.

We used a similar framework of Iterative Deepening Search found in the textbook, so we didn’t have many problems while using it. There were few surprises when other components of the AI started to malfunction while iterating deeper into the game tree. This would lead to the difficult session of debugging the way our AI handled things such as chains and tracking open spaces near occupied spaces. A large portion of the code had to be altered or removed because of how the iteration was interacting with the rest of the AI.

**4. 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? Describe the data structure you used. Did it help?**

We used a TreeMap to store and sort the values associated with each BoardModel so that we can expand the best node each time when we move to the next level of the IDS limit. We used the evaluation of the board as a key and the BoardModel with the move made as the value. Because a TreeMap by default sorts from least to greatest, we had to adjust the key so that the best move for the AI would appear first when iterating from the front of the TreeMap to the back.

This did improve the performance of the AI by allowing it to see if the best move from the previous depth did lead to better moves at lower depths. The overall effect on the AI’s performance was noticeable in that more of the moves it made were better than the version of the AI that did not sort its moves before expanding them.

**5. Describe your quiescence test, Quiescence(S). Did it help?**

Our quiescence test occurs after we decide on a move to make. After deciding on a move, we check all of the possible moves that the opponent can make if we made that move. If any of those moves could win the game for the opponent, we go back and search the game tree once more at a deeper level to see what the next best move would be. We repeat the quiescence test with the new move, and if it does not lead to a winning move by the opponent, we go ahead and make that move.

The quiescence test did help in that it did avoid obvious losing moves, but sometimes the AI could be forced to make the same move that failed the quiescence test. This would manifest itself by either having the AI time out or going forward and making the move that would give the opponent the game.

**6. Any suggestions for improving this project? (One suggestion is to remove the first-player advantage: the first player initially places one single mark, and then the players alternate each placing two marks per turn. But, this would square your branching factor for each ply. I hope to compensate for this in the tournament by having each pair play both sides in alternation.)**

One possible solution for improving this project is to compensate for first-player advantage by decreasing the worth of a win by the first player, and increasing the worth of a win by the second player. A solution to help students implementing the AI would be to offer code examples of how to perform Iterative Deepening Search, or Quiescence Tests and other similar searches.