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

Partner Names and ID Numbers: Chloe Nguyen 56897907 (No Partner)

Team Name: SixDollars

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. My heuristic function evaluates a state by giving an overall sum of scores for AI's multiples, and deducts the score with the weighted values of the opponent's multiples. The larger the multiple, the greater the values linearly. Small multiples are worth 10, 2 pieces away from a winning multiple is 25, 1 piece away from a winning multiple is 100, while the winning multiple is a flat 1000 for the overall evaluation.

   I valued multiples in a row: horizontally, diagonally, and vertically. Numerically, I have added to the evaluation score with AI multiples and deducted with opponent multiples at 10*n in a row. Winning moves and near winning moves are given significantly higher score of a flat total of 1000 and 100 per multiples in a row, respectively.

   My function scans the board vertically, horizontally, and finally diagonally. Vertically moves column by column while horizontally moves row by row in the graph of the board. These two would check if a piece exists on a spot and add to the piece count of the respective player, while resetting the opposite player's score to zero. This is necessary when finding multiples in a row, otherwise this method falls apart as opposing player pieces disrupt the odds for multiples in a row. When it reaches an empty spot on the board, it finally adds to the evaluation of the state based on the multiples of player and opponent multiples.

   Diagonally, it works the same way, although multiples smaller than near winning conditions are more difficult to count and I've ran out of time implementing a better code structure to evaluate multiples.

   This heuristic is not successful at forming multiples, however, or winning despite the winner function being logically solid for finding winning lines and giving a hard evaluated score of 1000 which should exceed other heuristic code. Probably because the odds of a winning move will be available is too early for the heuristic to determine with the limited amount of friendly pieces. There's also the case of the AI running out of time to pick an ideal move. Finally, it also does not take single spaces into account well, which would be helpful for blocking enemy winning conditions.
2. Alpha- Beta was implemented with recursive MinMove and MaxMove functions. Both functions take the parameters: the coordinates of the current move, the board state with the current move placed, the current depth limit, the start time of the program, and finally the alpha, beta values.

   The IDS loop begins with positive infinite limit for beta and negative infinite limit for alpha. It then creates or continues the decision tree by starting with a call to MinMove. MinMove creates the decisions that Min(the opponent) would choose; their optimal decision being the lowest scoring path for Max (this AI) to choose for the AI's current move. MaxMove would be the inverse scenario where Max has to choose the highest scoring path, taking into account Min's decision. Going back to the description of MinMove, it compares the score of the move with beta. If it is lower than beta, it will reassign beta to the move's value and then call MaxMove. MaxMove would compare the score of the move with alpha; replacing alpha with the move's score if it is higher.

   The alpha beta pruning happens if the alpha value is greater than the beta value, cutting off the minmax search through the children and returns the value of the current move. It helps reduce the amount of time needed to find a high scoring move. It can simply be turned off by commenting out the alpha >= beta comparison. Finally, if the time has been reached or the depth limit has been reached for either the MinMove or Max move function, then the search stops, and then evaluates the score of the move with the heuristic function.
3. IDS was implemented using a while loop with the condition that the time for the IDS loop does not exceed 1000 milliseconds before the deadline is up. It starts at a depth limit of 0 and increases by 1 with each execution of the loop.

   First, the time was initialized with time_t keeping track of the time of the AI's execution. The current time is updated at the end of each IDS loop with the difference of the program's starting execution time with the current time. As the difftime() function returns the time in seconds, I've multiplied it with 1000 to count it in miliseconds.

The main difficulty with an implementation of IDS is finding efficient checkpoints for updating the current time before the deadline. I've implemented it at the end of the IDS loop, and the beginning of MinMove and MaxMove as the two functions call each other recursively to search through the decision tree.

4. This AI makes use of the std::vectors, std::priority_queues and std::unordered_map data structures. The std::vectors stores a list of generated moves, along with their evaluated scores. The priority queues sort through moves with scores for the use of the IDS, MinMove and MaxMove. The unordered map keeps states of the board as keys with their scores as the value in order to save time for already explored nodes.

   There are two priority queues in the IDS loop and one for each of MinMove and MaxMove. One priority queue, PQ, in the IDS loop keeps track of moves to return with the move's largest score being given the highest priority in the queue. The priority queue "m" is used as a temporary container of moves with their states' scores to iterate through with the moves with the highest scores being higher priority. This allows for an effective use of time with the deadline in consideration since the best looking moves are searched through first. The Priority Queues in MinMove and MaxMove keep track of all of the moves and their scores; MinMove keeping the lowest scoring move as highest priority while MaxMove keeps the highest scoring move as highest priority.

   First, the IDS function begins by generating all the possible moves by scanning the current gameState's board for empty spots. Then the IDS loop begins by checking if the unordered_map does a random search with one of the possible moves to see if the decision tree has already been created. If no tree has been created, proceed to create a tree with an initial depth limit of 0 and a call to minMove. If the tree has already been created, then create a priority queue with the list of Moves and the scores of their respective states. The scores for the move are obtained from the unordered map. Then proceed to continue the tree with a call to MinMove. With each call of MinMove and MaxMove, the local depth increases by 1. The depth limit keeps the tree from exceeding beyond the current depth if it matches or exceeds it. The unordered_map only keeps track of all of the best moves and their states at the end of each MinMove/MaxMove call.

   The priority queue, PQ, and unordered_map helped a lot with saving time as the former sorts the best moves as highest priority, and the latter keeps track of explored children of IDS.