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

Partner Names and ID Numbers: Chen Lu (51398516), Tetsuichiro Kobayashi (89172212)

Team Name: BetaGo

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).

Our heuristic evaluation has three components to guarantee the AI detects danger and potential checkmate moves. First of all, at the beginning of each move (before it goes into IDS), AI checks if we can win in the next step. If so, we will return that move immediately without using up 5 seconds to search. This function is called winMoves(). WinMoves() calls eight functions (one for each direction) to see if the given player can win the whole game immediately. Each of these eight functions is checking if there are k-1 continuous spots at that direction. When the AI realizes we can't win immediately, it will starts its IDS by calling IDSRecurse() recursively. In each recursive call, we apply the second component: a function called threats(). Threats() reduces search targets. Instead of doing IDS on the whole board, we only do IDS on critical spots that threats() returns. Threats() is similar to winMoves() as it also calls that eight direction functions (but with slightly different parameters to do a more general search). This threats() stops AI from wasting time in searching "peaceful" spots. When we reach the bottom level, we use the third component to check danger, a function call inDanger(). InDanger is similar to threats() but instead of returning guessed critical areas, it is doing a thorough search of dangers in that specific spot. Because spots passed to inDanger() are already potential critical spots, inDanger() use a different, yet more thorough method to prioritize those critical spots. To be more specific, threats() is checking if we have k-1 continuous spots in that direction but inDanger() considers more cases. For example, it can see k-1 spots where we have empty space in the middle of that line. It also tells how dangerous this spot is by analyzing the number of opponent's pieces in a line and number of empty pieces in that line. In inDanger(), each direction is checked and each critical spots is given a weight, indicating "how critical is this point".

To sum up, with a combination of three heuristic components, BetaGo is responsive in checkmate situations (by winMoves()), efficient in reducing searching targets (by threats()) and accurate in prioritizing them (by inDanger()).

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).

An accurate heuristic value is returned by inDanger() at the bottom level. This value (called nextLevelEval) is passed to its parent level and compared to Alpha or Beta (depends on turns). This value might cause the whole level search to be abandoned if the opponent has a worse move to pick and it is our turn (because from concept of AB proning, opponent will avoid the best move from this branch). Alpha and Beta is up-to-date since we passed them to each recursive IDSRecurse().

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

IDS calls winMoves() first to see if we can win in next step. If so, we will return that move immediately without wasting time in IDS. If no move achieves sudden win, we set the depth to 0 and call IDSRecurse on it. Then, we call IDSRecurse on depth 1, depth 2, etc. After each depth finishes, we compare the best move from that level to the best move from previous depths and update it if needed. We will stop searching current step if: we finish searching it, or, Alpha/beta indicates this level is useless, or, time is up.

Time management: we add in the logic "stopping immediately if overtime" between each depth level and inside each recursive IDSRecurse(). We set time limitation at 4500ms.

Difficulties: We once called inDanger in each recursive IDSRecurse(). But we found it was wasting time because a similar board is being search n times (n is max depth). In order to save running time, we refactored the code to replace inDanger() at the bottom level only. Because bottom level has the most complete state so searching that part only is sufficient.

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?

We only evaluate heuristic value at the bottom level to save evaluating time. But each recursive function updates the board from spots added from previous depth so that its imaginary board state is up-to-date. Also, in order to save evaluating time for specific spots, we spend some time to find some most critical spots to add given the board from previous level by calling threats(). We save one best move for each level and do a cross-level comparison when a new level is finished.