CONNECT-K FINAL REPORT

Partner Names and ID Numbers: Jeffrey Lee Thompson 12987953

Team Name: WIP

1. Our heuristic evaluation function evaluates the current board by evaluate K pieces of the board one at a time. We create a class called EvalObject that would calculate and keep track of the score. The structure of EvalObject is three different counters, a score, a circular queue of size K and a gameover flag. We would keep pushing pieces into the queue, and update the count then pop the queue when it is full and update the count. The score is being updated according to the queue.

   The score calculation is calculated on the fly so that we do not go over the board twice, one for just input and one for just calculating. The way we calculate EvalObject is that we have three different counters, which are empty, us, and enemy. These three counters would always add up to K except when the beginning of the game when we are filling up the queue inside the EvalObject. When the count of us and enemy is both greater than 0 we would just return 0 for a set of K pieces of score, because no one can make a winning move within that K pieces. When there are no pieces in for a set of K pieces it would return 0 because both side can make a move freely. When us or enemy has more than 0 pieces in a set of K pieces the score is calculated as 30*count minus the 5*empty count, because the more one has in the K set of pieces the more advantage one has but the empty spaces would set one back a bit. The ratio of the weight between regular count and empty count has to be big enough so that the score would not become negative due to the larger count of empty count compare to regular count. Whenever either our or enemy's count is K we would turn the gameover flag to be true and then return the corresponding score for each player. The idea of this heuristic is that the the more pieces in a set of K pieces you are closer to winning without enemy interference.
2. Our search is a recursive algorithm which is the function ids(). This function called at progressively greater depths until the time is up in which the search is halted and the best move is returned. This first call of ids() is passed Alpha = INT_MIN, Beta = INT_MAX and the depth of the search. If the depth passed is greater than 1 then for each possible move in the search space there is a call to ids() with the current values of Alpha and Beta and depth-1. If the depth passed to ids() was 1 then the heuristic function is used to determine the score instead of the recursive call.
   The return value from ids() and heuristic function is the score from that board position. If this level is a max and the return value is greater than Alpha, then Alpha is updated. Likewise, if this is a min level and the return value is less than Beta then Beta is updated. Then if as result of this update Alpha >= Beta the ids function returns the value of the best move so far at this stage. This then happens at the next level down. If this move did not result in a prune, then the next possible move is checked until all have been exhausted and the best move is returned.

   We did not include a switch to turn the pruning on and off as the first that functionality was mentioned was in the report-template, which we received after implementing the Alpha=Beta pruning. At this point our focus was on further improvements so if this functionality is not required there was no point in adding it.
3. This was mostly described in #2. Points not mentioned is that our min-max tree is linked list based where each node is a possible game state. Each node has a pointer to the next possible gamestate from it's parent and a pointer to the the list of possible moves from the expansion of that node. This list is "walked down" with each possible move being evaluated. If a move (game state) is determined to be the best found so far among the possibilities, then it is moved to the front of the linked list. This way the next level search will check the best move found by the prior level search first which assist in the pruning.

We resorted to multithreading as we could not find a good way to implement a timer without them. It also seemed to us that there is little reason to throw away the min-max tree each move then rebuild from scratch the next move. In real life you can think while your opponent is deciding so why not have our AI do so? It turns out that didn't help much though as the opponent doesn't think the same way we do so we likely pruned their move. Even if we didn't, searching depths 1 through k-1 takes less time than just depth k, so likely we get halted at the same depth search we would have been without that functionality.

The main thread is the one that communicates to the Java program, but at startup a "builder" thread is created that builds the min-max tree. The main thread updates "builder" when the opponent moves and has the timer telling "builder" when to stop. We originally did this utilizing C++11 methods, but it turns out the openlab gcc has not been updated to C++11 support so we had to fallback to pthreads (Thank you for adding the flag to the build script for us). Luckily they work pretty much the same way, only with C style syntax. Debugging a multithreaded program in C++ is surprisingly difficult as the language does not have native support for this. Even simple things such as using std::cout from multiple threads resulted in mixed output that often mixed with the communications with the Java threads.

We ended up creating our mini-tournament program in C++ so a full debugger could be used instead of just print statements and we created a flag so the multithreading could be turned on and off while keeping the logic mostly in-tact. Synchronizing the two threads was troublesome as well, but we eventually figured out how to keep the two in sync after a lot of trial and error.

One funny bug we had is our AI was stable and could often beat GoodAI but was losing or crashing against PoorAI. The issue turned out to be that the PoorAI was returning with a move before the second thread had registered the last so the gamestate was getting messed up. A re-evaluation of how the threads communicated fixed the issue.

4. We do remember the values associated with each node in the game tree at the previous IDS depth limit but they are only used as a terminal test - if the game can be decided at that level there is no reason going deeper. We do “sort” the possible game-states somewhat, but not fully. As described in prior answers when a better move is found it is moved to the front of the linked list. The best move is at the front and better moves are likely near the front, but not guaranteed. The important part is that the best moves from the previous depth is searched first so that alpha-beta pruning works well.
5. This is mostly described in #1. There are four ways one can win a game, K pieces in a row, K pieces in a column, K pieces in two different diagonals like X. Hence, we have 4 different EvalObjects to evaluate each type of winning. We pass each corresponding pieces in order to the same EvalObject. Like for the row EvalObject we would pass each rows into the same EvalObject, or like for one of the diagonal EvalObject we would pass the same direction of diagonals into the same EvalObject. At the end we ask for the score that each EvalObject calculated and add them up to get the score of the current scoring. The scoring of different EvalObjects are weighted the same because they are equally important.
6. We would suggest you go with a MAKEFILE so that your compile script can just call it instead of requiring a certain language or method of compilation. A template of this MAKEFILE could be provided so most students won't mess with it, but those who want to do something different are not confined by defaults. In the very least detailed instructions of expected filenames and how the files will be compiled should be provided - this was not until the 3rd stage of the draft.

   It is actually very hard to debug on the C++ with just print statements. It would be great if there was a native C++ test program so we can run the program in a full debugger. This would result less error when running the program so that TA does not need to extend the deadline for draft AI.