John Alexander

CS 165A

MP2

Architecture: My implementation consists of a runner class responsible for making decisions and handling the gameplay (called "Reversi"), a "Board" class that represents the current state of a board, a "PlayTreeNode" class that makes up a node in the minimax tree and handles minimax computations, and a set of helper objects representing minor structures: Score, Position, and PositionWeightPair. There is always one instance of the Reversi class that holds the current master versions of the board and chooses moves using PlayTreeNodes. A board consists of an array of bytes representing the positions of the pieces, and the methods on a board can return all possible moves, check if a move is valid, check if a particular color can move, get the score, and get the weight associated with the board (based on heuristics).

Search: On the surface, I use basic minimax with alpha-beta pruning. At each move, I preserve the calculated tree from the previous iteration by taking the child of the root node corresponding to the moves made by my algorithm and the opponent. I then execute iterative deepening minimax search starting with the current depth of the tree or 4, whichever is greater. Before starting execution of minimax, I calculate an "allowed time" for the current run. This is equal to 1.8 over the remaining number of nodes times the remaining time. After the first depth-first search, I (conservatively) estimate the time for the next minimax execution as 8 times the time taken for the last execution. If this estimated time would still leave me below the allowed time for the current run, I execute the next depth of minimax. Otherwise, I return the current best node. The heuristics used in rating the "quality" of each board from the point of view of the algorithm weight certain positions based on the long-term advantage they provide. All edge pieces except those adjacent to corners are given an additional weight of 5. Edge pieces adjacent to corners are weighted -10. Corner pieces are weighted with 15 times the width of the board squared over 64. This means that as the size of the board increases, the weight of corner pieces does too in order to keep their weight proportional to the number of positions on the board (and thus, roughly, the score of each player. All of these weights are added to the scores of each player, and the score of the opposing player is subtracted from the score of the computer, yielding a value that is negative if the computer is (including heuristics) losing and positive if the computer is winning.

Challenges: I originally wrote the entire program using multithreaded breadth-first search. This functioned using a queue of nodes that should be evaluated and between 5 and 50 threads that pulled from this queue. This approach created a variety of problems, including ensuring that the threads did not overwhelm the main thread and prevent it from continuing and coordinating the activity of the threads. Some problems that I could not even solve were applying alpha-beta pruning to breadth-first search (I am increasingly convinced this is not even technically correct) and ensuring that all branches were evaluated to the same depth (may not be possible without significant coordination between threads). Although the code ran and could beat some basic minimax algorithms, it still did not perform well. I eventually scrapped this entire model (about 6

hours before the project was due) and rewrote most of the minimax logic with iterative deepening. Except for one pesky bug, this was extremely simple and took me about four hours. This is the version I am turning in.

Weaknesses: Technically speaking, it is not a good idea to negatively weight any positions, as I do. In general, my weighting is fairly simplistic compared to some systems (particularly those that regularly won the tournament) and could be improved significantly. Some experiments should also be made to determine the best manner of estimating running time for the next execution of minimax (8 is likely too large a multiplier, particularly with alpha-beta pruning included). Improving this estimation could significantly improve the depth of the search from its current variance between 5 and 10. Finally, micro-optimizations might allow me to eek out some additional levels of analysis.