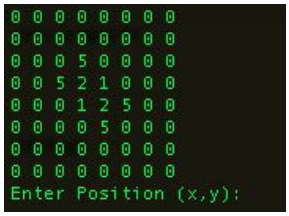
Michael Ferrin: mi6034fe-s
Heqthor Nerdemo: he3486ce-s
EDA132- Spring 2016

Assignment 1: The Reversi Game

Gameplay:

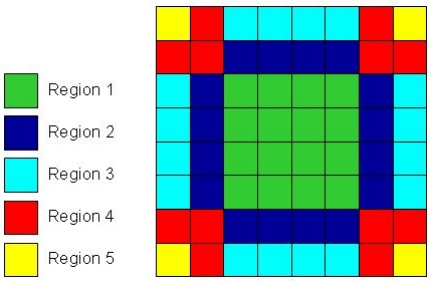
Heqthor and I started the assignment by coding the game of Reversi in C because it was the only language we both knew in common. We used a matrix of arrays for the game board. The 0's represent empty spaces, 1's represent black pieces, and 2's represent white pieces as can be seen in our ASCII grid below:



Our game also has 5's which show where a player is allowed to move on that specific turn as was asked of us by requirement #1. All the game logic is stored in *board.c*. This file includes the board as well as many functions necessary for things like keeping score, checking for valid moves, and flipping pieces. The main method calls all the important functions and runs as long as the game is still active. The program can be run by calling "make board" followed by "./board" from the command line. An executable file, *board*, is included as well that will automatically open our program and run it in terminal. I stored our assignment in my LTH account under *eda132/assignment1* on my home directory (or more specifically */h/d9/r/mi6034fe-s/eda132/assignment1*.

Strategy:

After getting beat by the computer on easy several times I needed to research strategy to use for our heuristics. My first thought was that the minimax should look to see which player had the most pieces on the board. This would have made the minimax easier to implement but it is a naive strategy because this is not a real indicator of how well a player is doing. We manily looked at two different websites to get further stratigic help: *http://mnemstudio.org/game-reversi-example-2.htm* and *http://www.riscos.com/support/developers/agrm/chap09.htm*. The first website gave us the idea to divide the game board into different sections because some squares are more valuable than others:



The second website reinforced this idea and helped us decide how heavily to weight each square. This can be seen in code at the top of *minimax.c* where the board is divided into different sections. This information was later accessed by out minimax algorithm. I also received help storing coordinate values as integers from a stackoverflow post: *http://stackoverflow.com/questions/12700497/how-to-concatenate-two-integers-in-c/12700533#12700533*. By using concatenation it became easier store a coordinate value like (3,4) as the integer 34.

Minimax:

Minimax.c is where the bulk of our minimax algorithm lies. In addition to referencing the textbook, a lot of our minimax algorithm was based on a similar project that I did a year and a half ago for a different class. The code for that project can be found here: *https://github.com/ferrin22/CS61B/tree/master/Jump61*. The main differences between my previous minimax algorithm and the minimax for this project were that the heuristics were completely different, this project involved a timer, and we ran out of time and weren't able to include alpha-beta pruning unfortunately. Other than that the projects were very alike. For abstraction we decided to break minimax into two different parts. The case for depth zero is in a function called *basecase* and the rest is in *minimax* which sometimes calls *basecase*. The goal of our algorithm was to recursively check all possible moves, alternating between players, to figure out which sequence of moves would result in the highest heuristic value later on. At the most simple level, given little time, minimax will simply look at all it's current moves and decide which one is the best. Given more time however, it will play out a sequence of different moves for every move it can make on the current turn. Given that our algorithm did not include alpha-beta pruning, we started it a depth of 6 because anything greater than that took noticeable time. Given a time limit, as many layers of depth as possible are evaluated until the time limit is reached and the move with the highest heuristic value gets selected. I was unfamiliar with the different clocks and timing features in C so once again I turned to stack overflow for help: *http://stackoverflow.com/questions/459691/best-timing-method-in-c*. We decided that it seemed like a good idea to time our algorithm according to processing time rather than real-world time since real-world time varies so much from one computer to another. Hopefully this decision doesn't make our game react wildly different than those in the class who did not use processing time.

Function Descriptions:

board.c

screen - prints out ACSII grid to the screen
newgame - resets the game board to its original state
islegal - checks whether the input move is allowed
slowScore - keeps score of how many pieces each player has on the board
retZero - clears the suggested moves off the board after each turn
checkMove - determines where the current player is allowed to move
putPiece - places pieces on the board and flips the nessicary pieces after a move

minimax.c

basecase - simplest case for minimax when time limit is too small
minimax - minimax algorithm
checkForMoves - same as check move above but in the context of minimax
zero - same as retZero above but in the context of minimax
concatenate - 3,4 -> 34; 6,2 -> 62