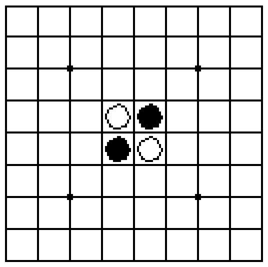
3/2/2015 12:35 PM

CSCE 625
Programing Assignment #5
due: Friday, Mar 13 (by start of class)

Minimax Search for Othello

The goal of this assignment is to implement a program for playing Othello using Minimax search. Othello, also known as Reversi, is a 2-player game, where players take turns placing black and white pieces (or stones) on an NxN board. When two pieces of one player flank a consecutive row, column, or diagonal of an opponent's stones, they may be flipped over. Once the board is filled or neither player has a legal move, the player with the most pieces wins. To familiarize yourself with this game, you can play an online version here: http://www.othelloonline.org/



Minimax search itself is easy to implement (as a pair of recursive functions, like in the textbook). However, since it is infeasible to search all the way to the bottom of the game tree, you will have to truncate the search at a defined depth limit, and then call a board evaluation function to return a score that indicates how good that board state is (i.e. how likely it is to lead to a win or loss). A very simple candidate board evaluation function which you can use for initial development and testing is one that just returns the difference between the number of black and white pieces. However, you might be able to come up with an improved evaluation function that better discriminates between good states and bad states.

To evaluate your programs, we will run them in competition with each other in a tournament. We will use the same depth-limit for all players (set by a command-line argument) for fairness. Ideally, the implementation with the best board evaluation function should win.

Rules of Othello

http://www.site-constructor.com/othello/othellorules.html

Pay attention to how flipping of pieces is determined with each move.

Important: note that flipping at least one of your opponent's pieces is REQUIRED to make a move. If a player cannot move, they FORFEIT their turn.

Implementation

Your program should be interactive. It should accept commands at a prompt. The commands are:

init	places 4 pieces in the middle (standard starting configuration)
put <P> <X> <Y>	place a stone of type P at coords X,Y, and flip any pieces that can be flipped
move <P>	ask the program to choose the next move for player P using minimax (and update the state based on this move)
reset	removes all stones, leaving an empty board
quit	

With these commands, you can play against your program. An example transcript is shown at the end. After calling 'move', the program should print out the chosen move (based on calling minimax)in this format: **(X,Y)** (using the coordinate system shown below).

```
 (0,0)   ... (0,N-1)
....
(N-1,0) ... (N-1,N-1)
```

If there is no legal move for the player, it must output **forfeit.**

For debugging purposes after 'put' or 'move' commands, your program might want to print out the board state and current score (piece differential). Anything printed with a '#' at the beginning will be treated as a comment and ignored.

Here is an example that shows how to start the program, and the put and move commands. Notice how putting the B in 1,2 flips the white piece in 2,2 to black. Minimax chooses 3,1 for the move for white because it has the *lowest* minimax score. Assuming that the utility function is oriented for player black, higher scores are better for black and lower for white. Also, notice how the move command outputs "(3,1)".

```
unix> othello 6 8
> init
# . . . . . .
# . . . . . .
# . . W B . .
# . . B W . .
# . . . . . .
# . . . . . .
> put B 1 2
# . . . . . .
# . . B . . .
# . . B B . .
# . . B W . .
# . . . . . .
# . . . . . .
> move W
# making move for W...
# considering: (1,1), mm=0
# considering: (1,3), mm=0
# considering: (3,1), mm=-2
(3,1)
# score=0
# . . . . . .
# . . B . . .
# . . B B . .
# . W W W . .
# . . . . . .
# . . . . . .
```

The program itself should take three command-line arguments, "othello <N> <C> <D>", where N is the size of the (square) board, C is the color to be controlled by the program, and D is the search depth limit.

You will want the utility function to be oriented toward the color that is passed in as an argument to your program. That is, higher values are better for the color that is passed in.

Although you can use any language you like, if your program is too slow, we might have to lower the depth limit, which will probably result in a detriment in performance compared to other players.

You are welcome (but not required) to implement alpha-beta pruning. It requires only a few modifications to the min_value() and max_value() functions, to pass in, check, and update the alpha and beta parameters.

In the competition, we will possibly change the size of the board (probably between 4x4 and 10x10), we might change the starting state (initial positions of the stones) e.g. for some randomness (even though the official rules define the initial state to be 4 stones in the middle), and the depth limit (adjusted to keep the speed manageable, but done fairly for both opponents).

There will be a monitor program that runs two programs against each other in the competition. This program will track the moves for each player and determine a winner of the game. It is critical that the syntax given in the table above is followed exactly in order to work correctly with the monitor. In addition, it is necessary to flush any buffered input that is written to stdout in order to avoid the monitor waiting for input indefinitely. Therefore, after you write to stdout you will want to follow it with something like the following:

- **Java** – System.out.flush() or terminate input by ‘\n’
- **C** – fflush(stdout)
- **C++** – terminate input by std::endl
- **Python** – system.stdout.flush()

In your developing your code, you will almost certainly have 3 sections:

- The **main loop**, which reads input from typed in by the user, maintains a state of the board, and responds appropriately to commands.
- A **Board class**. This contains an internal representation of the state of the board, such as a character array. You will want some auxilliary functions, such as for copying or printing out board states, computing the score (# black stones - # white stones). The 2 key functions, legal_moves() and make_move() are described in more detail below.
- **Minimax functions**. The signatures of my functions in this section look like this:

```
Move minimax_move(Board board,char player,int limit);
int min_value(Board board,char player,int depth,int limit);
int max_value(Board board,char player,int depth,int limit);
```

Move is a simple class that holds a pair of coordinates.

The critical member functions of the Board class are:

```
Board Board::make_move(char P1,int x,int y);
vector<Move> Board::legal_moves(char P);
```

make_move() put a stone for player P1 at position x,y, *and then flips all of the opponent's pieces that are possible*. Pseudo-code for the flipping algorithm looks like this:

```
temp <- copy current state and put piece for P at x,y
for (int i=0 ; i<N ; i++)
  for (int j=0 ; j<N ; j++)
    if i,j is on same row, column, or diagonal as x,y
      and i,j also contains a piece of type P
      and all the positions between i,j and x,y are occupied
        by the opponent's pieces
      then flip all of the pieces inbetween i,j and x,y to P
```

legal_moves() can then be implemented in a simple way by calling make_move(). legal_moves() can iterate through all the unoccupied board positions that are adjacent to on opponent's piece, call make_move() to try it out temporarily (which makes and modifies a copy of the state, rather than changing the state of the board object itself), and calculates the score to see if the magnitude of the piece difference changed by more than 1 (indicating that at least one of the opponent's pieces got flipped, which is required for a legal move). The legal_moves() function returns a vector of all such moves (or boards if you want, i.e. successor states). Note that this vector could be empty, in which case the player will forfeit their turn.

<u>What to Turn in:</u>

- You will submit your code for testing using the web-based CSCE *turnin* facility, which is described here:

https://wiki.cse.tamu.edu/index.php/Turning_in_Assignments_on_CSNet (you might have to be inside the TAMU firewall to access this)

- Include a brief document that describes how to compile and run your code, and some example output (e.g. a transcript of playing against itself).

Example Transcript

```
compute-linux1> othello 6 4
> init
# . . . . . .
# . . . . . .
# . . W B . .
# . . B W . .
# . . . . . .
# . . . . . .
> move B
# making move for B...
# considering: (1,2), mm=3
# considering: (2,1), mm=3
# considering: (3,4), mm=3
# considering: (4,3), mm=3
(1,2)
# score=3
# . . . . . .
# . . B . . .
# . . B B . .
# . . B W . .
# . . . . . .
# . . . . . .
> move W
# making move for W...
# considering: (1,1), mm=0
# considering: (1,3), mm=0
# considering: (3,1), mm=-2
(3,1)
# score=0
# . . . . . .
# . . B . . .
# . . B B . .
# . W W W . .
# . . . . . .
# . . . . . .
> move B
# making move for B...
# considering: (4,0), mm=3
# considering: (4,1), mm=3
# considering: (4,2), mm=1
# considering: (4,3), mm=3
# considering: (4,4), mm=1
(4,0)
# score=3
# . . . . . .
# . . B . . .
# . . B B . .
# . B W W . .
# B . . . . .
# . . . . . .
> move W
# making move for W...
# considering: (0,2), mm=-2
# considering: (1,1), mm=0
# considering: (1,3), mm=2
# considering: (1,4), mm=2
# considering: (3,0), mm=0
(0,2)
# score=-2
# . . W . . .
# . . W . . .
# . . W B . .
# . B W W . .
# B . . . . .
# . . . . . .
> move B
# making move for B...
# considering: (0,1), mm=3
# considering: (1,3), mm=1
# considering: (2,1), mm=3
# considering: (3,4), mm=3
# considering: (4,1), mm=1
# considering: (4,3), mm=3
(0,1)
# score=1
# . B W . . .
# . . B . . .
# . . W B . .
# . B W W . .
# B . . . . .
# . . . . . .
> move W
# making move for W...
# considering: (0,0), mm=-4
# considering: (1,3), mm=2
# considering: (1,4), mm=1
# considering: (2,4), mm=1
# considering: (3,0), mm=-2
(0,0)
# score=-2
# W W W . . .
# . . B . . .
# . . W B . .
# . B W W . .
# B . . . . .
# . . . . . ..
.
. > move B
# making move for B...
# considering: (4,5), mm=-18
(4,5)
# score=-7
# W W W W W B
# W W B B W B
# W W B B W W
# W W B W B B
# W W W B B B
# W W W B B .
> move W
# making move for W...
# considering: (5,5), mm=-18
(5,5)
# score=-18
# W W W W W B
# W W B B W B
# W W B B W W
# W W B W B W
# W W W B W W
# W W W W W W
game over
```