Homework Assignment 3: Game Tree Search

CSC 384 – Winter 2003

Out: February 24, 2003
Due: March 10, 2003: in class

Be sure to include your name and student number with your assignment.

In this assignment you will implement a computer program to play "othello" (a.k.a. reversi). This is a two-player zero-sum game with very simple rules, but yet the game is quite challenging. The assignment is divided in four parts. By the end of the fourth part you will have incrementally built and refined a prolog program to master the game. This assignment has a programming component as well as a theoretical component. Ideally you should do the programming first and then answer the theoretical questions based on your results, however if you run out of time and your program isn't complete, you can still answer all of the theoretical questions.

First, some of background regarding the game. Othello is a board game with discs that are white on one side and black on the other side. Two players take turns in placing one disc at a time on the board. The first player always places the discs on the board with the black face up and the second player always places the discs on the board with the white face up. The game is usually played with an 8x8 board but for the purpose of this assignment we will use a smaller 4x4 board.

Game objective

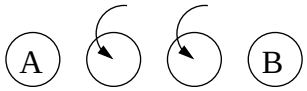
The objective is to have the majority of your colour discs on the board at the end of the game.

Possible moves

- "pass" (forfeit your turn) or
- "outflank" your opponent's disc(s), then *flip* the outflanked disc(s) to your colour.

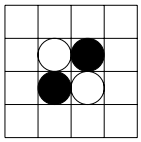
To *outflank* means to place a disc so that you have a disc of your color at each end of an opponent's row, column or diagonal of discs of his color. Here's one example: white disc A was already in place on the board. The placement of white disc B outflanks the row of two black discs.

White flips the outflanked discs and now the row looks like this:



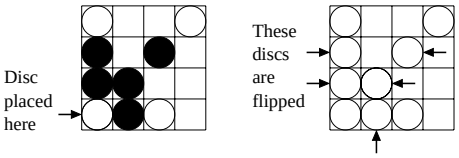
Rules

1. The game always begins with the following setup and black always moves first.



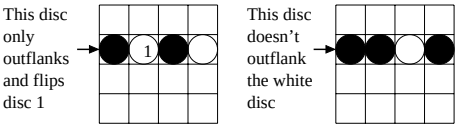
2. If on your turn you can outflank some opponent discs, then you *must* outflank and flip at least one opponent disc. In other words you cannot pass when it is possible to outflank. If you cannot outflank any opponent disc then your only choice is to pass.

3. A disc may outflank any number of opponent discs in a continuous straight line in one or more directions (horizontally, vertically or diagonally).

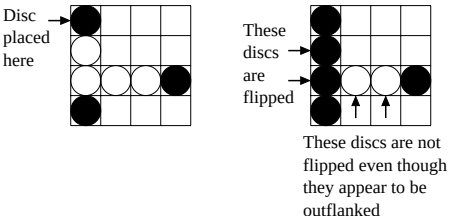


In theory, discs in all 8 directions could be flipped.

4. You may not skip over your own colour disc to outflank an opponent disc.



5. Discs may only be outflanked as a direct result of a move and must fall in the direct line of the disc placed down.



6. All discs outflanked in any one move must be flipped, even if it is to the player's advantage not to flip them at all.

7. The game ends when both player are forced to pass (i.e., neither of them can outflank). Note: it is possible for the game to end before all squares are occupied by a disc.

8. When the game is over, the score is determined by the number of black discs minus the number of white discs. If the score is positive, black wins. If the score is negative, white wins. A tie occurs for a zero score.

Before starting the assignment, it is important that you understand well the game so I encourage you to a play the game a few times. Here is a website with an applet to play online: `http://www.rainfall.com/othello`. Here is another one describing a sample game and giving some tips for a winning strategy: `http://www.pressmantoy.com/instructions/instruct_othello.html`.

On the course website, an interactive game playing shell `play.P` has been provided, implementing depth-first mimimax search. By simply calling the predicate `play`, this program will prompt you to input moves, and call minimax search to allow the computer to choose its moves. This search procedure has a depth bound, which you can set as appropriate. The program is generic and requires that several predicates be specified that are specific to the game. A second file `othello.P` (also available online) contains all of the predicates specific to Othello except the predicate `nextState`.

1. Your first job is to implement the predicate `nextState(Plyr,Move,State,NewState,NextPlyr)`. In the game playing shell, `nextState` is called after each move to determine the new state of the game and the next player. At the beginning of the file `othello.P`, the format of a state and a move is specified. Using this format, compute the new state `NewState` resulting from the move `Move` by player `Plyr` in state `State`. When the move is pass, this is trivial, however when it involves outflanking opponent discs, a bit more work is required.

   Note that the predicates `nextState` and `validmove` are similar so you may want to inspire yourself from `validmove`. The predicate `validmove` tests that a move to outflank opponent disks does indeed outflank some disks, whereas `nextState` actually flips the outflanked discs. In `nextState`, you can safely assume that the move `Move` is valid since it is always tested by `validmove` in the game playing shell before it is passed to `nextState` as an argument.

   Feel free to use any predicates already defined in `othello.P` or `play.P`. In particular, have a look at the predicates `replPos`, `getDisc`, `color`, `flip`, `nextPos`, etc.

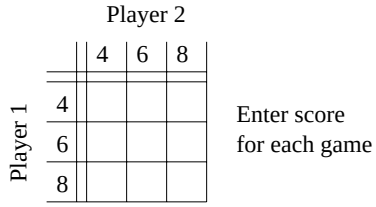
   What to hand in: Hand in a listing of your code (all relevant predicate listings) and a printout of a prolog session showing that your predicate `nextState` works correctly on five well-chosen test cases. Choose those test cases yourself and give a short (one sentence) explanation of what distinguishes each test case from the others.

2. Once the `nextState` predicate is implemented, you can play a game against your computer by typing `play.` Your next job will be to assess the quality of play of your program when you vary the depth bound of the minimax search. Note: if you run out of time and your `nextState` predicate is not working you can still answer parts b) and c) of this question.

   Evaluate the quality of play for different depth bounds by making your program play against itself. You can do this by uncommenting a stub that makes the computer select moves on behalf of the human player. Both players will then make moves selected by a minimax search. Vary the depth of the search for each player.

   What to hand in:

   (a) Record in the following table the score of each game when we set the depth bound to 4, 6 or 8 for each player.

   (b) Discuss how the depth bound influences the quality of play based on the scores recorded in the table for part a). In particular, explain *why* the moves selected tend to be better as we increase the depth bound. If you were not able to complete part a) you can still answer the question.

(c) As we increase the depth bound, you'll notice that the computer takes more time to select moves. In tournaments, in order to keep games relatively short, players are often required to select moves within a limited amount of time. If we select a depth bound that is too large the program may not select a move before the time limit. How would you turn the minimax algorithm into an "anytime" algorithm? An *anytime* algorithm is an algorithm that improves its answer with time and can be stopped anytime to retrieve the best answer it has computed so far. In other words, how would you modify the minimax algorithm so that it quickly computes a move and then incrementally refines it until it is stopped and forced to execute its best looking move? Do *not* implement your solution. Again you can answer this question without completing part a).

3. Your computer program is based on minimax search; but without any pruning. Replace the predicate `mmeval(Plyr,State,Value,Move,Depth)` with a new predicate `alphabeta`, that evaluates the game using minimax evaluation with alpha-beta pruning. The arguments to this predicate should be of your own choosing.

   What to hand in:

   (a) A code listing of your `alphabeta` implementation. Be sure to document your code well. If you make changes to various predicates in the `play.P` file, be sure to make clear note of these.

   (b) If you implement `alphabeta`, rerun the game where both players have a depth bound of 8 and indicate how much pruning was obtained for each move selection. More precisely, add a counter to `mmeval` and `alphabeta` to count the number of nodes expanded. Run the game once with `mmeval` for both players and report the number of nodes expanded for each move selection of each player. Similarly, run the game once with `alphabeta` for both players and report the number of nodes expanded for each move selection of each player. Note that the moves selected and the outcome of the game should be the same whether `mmeval` or `alphabeta` is used.

   (c) The order in which the children of a node are evaluated can influence the amount of pruning in alpha-beta search. Discuss how heuristics can be used to maximize the amount of pruning. You can easily answer this question without completing parts a) and b).

4. **[Bonus 10%]** In the file `othello.P`, the `h` predicate defines a simple heuristic which evaluates a state by the score obtained if the game was over. This heuristic is used to evaluate states when the depth bound is reached. In this question, you will use your creativity to design a better heuristic. For this, it helps to understand human playing strategies. Think about the board features a human would normally use to play well. You may also want to do an internet search on "othello heuristics". Othello is a classic game that has been extensively studied in computer science, so many websites offer good ideas.

   What to hand in:

   (a) Include a code listing of your heuristic and briefly explain in English how the heuristic works. In particular, explain why it should perform better than the default heuristic based on the scoring function.

   (b) Report the score of 3 games where player 1 uses your heuristic and player 2 uses the default heuristic. Set the depth bound (for both players) to 4 in the first game, 6 in the second game and 8 in the third game. Discuss the results. Explain how the quality of a heuristic influences the quality of play.