CMPUT 657: Heuristic Search

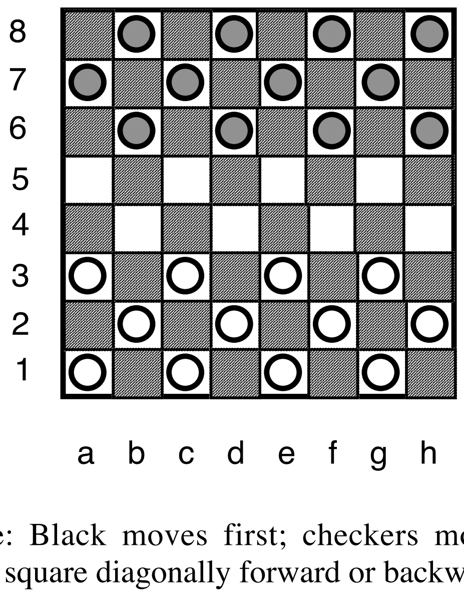
Assignment 1: Two-player Search

Summary

You are to write a program to play the game of Lose Checkers. There are two goals for this assignment. First, you want to build the smallest search trees possible. Second, you want to win the CMPUT 657 Game-Programming Championship (win the coveted championship trophy --- and defend our honor against the Icelandic game-programming champion). Both parts will contribute to your final mark for this assignment. You want the program to build small search trees, but also to run as fast as possible. Both issues are very important, because in the championship each program will be given a fixed amount of time per move. The smaller the search trees you build and the faster your program runs, the deeper your program will be able to search and, presumably, the stronger it will play.

Lose Checkers

Lose Checkers is played on an *nxn* board. Your program should be general enough to handle the 6x6, 8x8 and 10x10 cases. The game is played on a checkerboard, where only the white squares are used. A square can be one of empty, contain a black checker, black king, white checker or white king. The figure below shows the starting position for an 8x8 board. Note that the columns are labeled ‘a’ to ‘h’, and the rows ‘1’ to ‘8’. For the 6x6 board, there are 6 pieces aside, while for the 10x10 board there are 20 aside.



The rules are simple: Black moves first; checkers move one square diagonally forward; kings move one square diagonally forward or backward; when a checker reaches the end of the board it is promoted to a king; checkers and kings can capture; the first person to run out of moves wins.

Checkers can move only forward, diagonally, one square at a time to an unoccupied square. Squares are specified using algebraic notation, by giving the coordinates of the column, “a” to “h,” and row, “1” to “8”. Assuming Black starts at the top of the board, a Black checker on f6 can move to either e5 or g5. When a checker moves to the last rank of the board (squares a1, c1, e1, and g1 for Black; b8, d8, f8, and h8 for White), it is promoted to a king (usually shown in diagrams as two checkers on a square). Kings are allowed to move one square diagonally forward or backward to an unoccupied square.

Checkers and kings capture men by jumping over them. If the square to which a piece could otherwise move is occupied by an opposing piece, and the next square in that direction is vacant, then a capture is allowed. The piece jumps over the opposing man and removes it, landing on the vacant square beyond it. If in the resulting position the same piece can make another capture, you are required to continue jumping. Thus checkers can only capture in the forward direction and kings can capture in any direction. If you have a capture move you must play it. If you have a choice of captures any one will do. The promotion of a checker to a king ends a move; a promotion cannot happen in the middle of a jump sequence.

The goal of the game is to *lose* all your pieces! The game is over when:

1. the player to move has no legal moves (winning condition), or
2. a position has been repeated 3 times (the game is drawn).

Part 1

This component tests the efficiency of your search algorithm. Use alpha-beta search (your choice of variant) with iterative deepening. At interior nodes, you will likely want to do some move ordering. Use a transposition table with a maximum of 256K entries. The evaluation function consists solely of the material difference (i.e. if player White has 10 pieces and player Black has 8, then the material difference is +2 in Black’s favour – less pieces is better than more).

Part 2

This component tests the strength of your program. Anything goes. You can add search extensions or reductions to your algorithm, or any other search enhancement that you choose. You can modify the evaluation function in any way (note that you will want to consider doing this; material by itself may not be a good evaluation function). Do anything you can to improve your program's performance.

Interface

For Part 1, your program should support the following text commands:

Setting up a position:

i n	Initialize the game to use an *n*x*n* board (n = 6, 8, or 10).
B	Black is to move.
W	White is to move.
s	Setup a new position. The position is given by specifying the contents of the board from left-to-right and top-to-bottom. The setup uses “e” for empty, “b” for a black player’s checker, ‘B: for a black player’s king, “w” for a white player’s checker and “W” for a white player’s king. For example, the setup commands for the initial position of the 6x6 board would be as follows:

```
          s
```

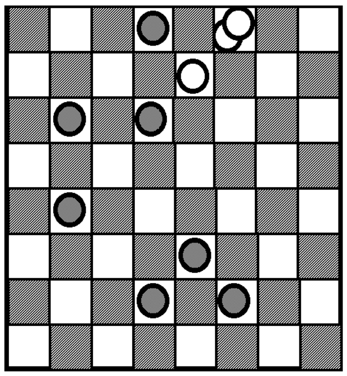
```
bbb
bbb
eee
eee
www
www
B
```

Playing moves:

mx1y2

mx1….y2

Move from square *x1* to *y2* (a piece duplication or a jump, depending on where *y2* is in relation to *x1*). A move like "ma1b2" results in the piece on *a1* moving to the empty square *b2*, while "ma1b3" would be a jump, removing the opponent piece on *b2*. Note that a move could consist of several jumps – each location that the capturing piece lands on has to be specified. In the following example, it is White to move and lose. White has only one legal move *f8g7*, Black must capture *d8f6*, White must capture *g7e5c7a5c3e1g3*, Black plays the winning move e3f2, White must capture *g3e1*, and since Black has no more legal moves, Black wins.



r Retract (undo) the last move played.

Search control:

d n	Set the search depth to "n".
t n	Search for "n" seconds of real time. When the time expires, stop the search. Note that both "t" and "d" can be set. Whenever one of the conditions is true, the search stops and the "best" move is played.
g	Begin searching.
1	Enable subsequent searches to use the Part 1 settings (fixed-depth, simple evaluation function).
2	Enable subsequent searches to use the Part 2 settings (anything goes).

Execution control:

q	Quit from your program.
Cntl-C	Interrupt the search and stop it. Play the "best" move returned by the search.

In addition, you will want to implement your own set of user interface commands to assist in your debugging!

For Part 2, the CMPUT 657 Lose Checkers Championship will be played using the Generic Games Server (GGS). You will get more information on connecting to the server later on in the course.

Output

After a search is complete, your program should display the best move, the best score, and the principal variation. It should play that move and then display the resulting new position. The following statistics should be printed after every search:

Tree size:	The number of interior and leaf/terminal nodes searched.
Time:	The number of seconds that the search took.
Search depth:	For each search depth (in an iterative deepening search), the number of leaf nodes examined in the search.
Trans. table:	The number of TT queries, the number of times the position was found in the TT, and the number of times that the TT entry gave a cutoff.
CutBF:	The average number of successors considered at a node where a cut-off occurs. Count only nodes where at least one move is searched (i.e., not just a transposition) and the value of the node is >= β.

Plan Of Attack

I recommend the following methodology for doing this assignment:

1. Build a program that supports setting up positions and playing legal moves. You should be able to setup a position, generate legal moves, play and retract moves, and end a game.
2. Add alpha-beta – nothing fancy – with the material-only evaluation function. Use shallow search depths to verify that alpha-beta is working correctly. Add assertions to your code so that if an error occurs, you catch it at the earliest possible time.
3. Add transposition tables. If you initially restrict a TT lookup to be valid only if the table depth exactly matches the depth that you need, then the TT will not change the result of a fixed-depth alpha-beta search. It should, however, reduce the number of nodes searched. Verify that this is working correctly!
4. Add in iterative deepening and move ordering. If you do this right, it should not change the final result of the search but, again, it should reduce the number of nodes searched.
5. Only when you are sure all the above is 100% working should you move on to more search enhancements and a better evaluation function.

Assignment Submission

1. By e-mail, send Akihiro Kishimoto (kishi@cs) the code for your program and a Makefile. All programs should compile and run without difficulty on a Linux box.

2. On paper, hand in a short document describing your Part 1 program. Skip the basics (do not explain how alpha-beta works). I want to know how you augmented alpha-beta to reduce the size of the search tree. Justify your search enhancements by providing some experimental data for fixed-depth search trees. This document must be no longer than five pages.
3. On paper, hand in a short document describing the enhancements made in Part 2. Describe how you changed the search and/or evaluation function and why you think this makes a difference in the program's performance. This document must be no longer than three pages.

Good luck!