**CS61B, Fall 2012** **Project #3: Lines of Action** **P. N. Hilfinger**

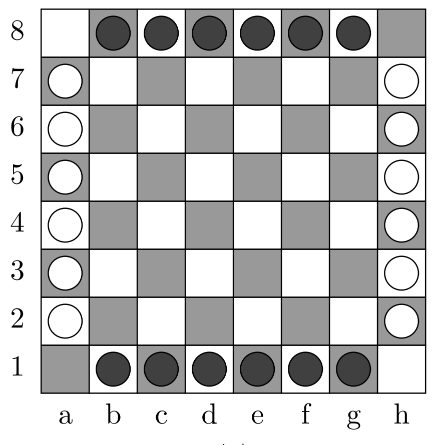
**Due:** Monday, 3 December 2012 at 2400

## 1 Background and Rules

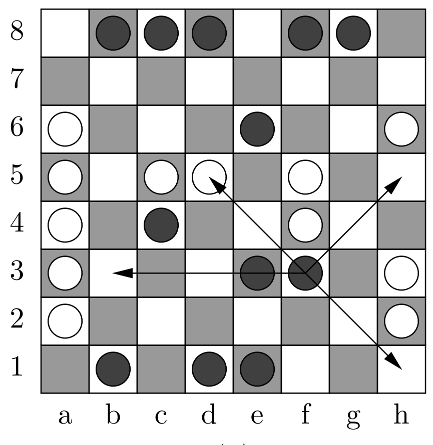
*Lines of Action* is a board game invented by Claude Soucie. It is played on a checkerboard with ordinary checkers pieces. The two players take turns, each moving a piece, and possibly capturing an opposing piece. The goal of the game is to get all of one's pieces into one group of pieces that are adjacent horizontally, vertically, or diagonally.

Initially, the pieces are arranged as shown in Figure 1a. Play alternates between Black and White, with Black moving first. Each move consists of moving a piece of your color horizontally, vertically, or diagonally onto an empty square or onto a square occupied by an opposing piece, which is then removed from the board. A piece may jump over friendly pieces (without disturbing them), but may not cross enemy pieces, except one that it captures. A piece must move a number of squares that is exactly equal to the total number of pieces (black and white) on the line along which it chooses to move (the *line of action*). This line consists of both the squares behind and in front of the piece that moves. A piece may not move off the board, onto another piece of its color, or over an opposing piece.

Figure 1b illustrates the four possible moves for a black piece in the position shown[1] The move f3-d5 is a capture; all others are ordinary moves.



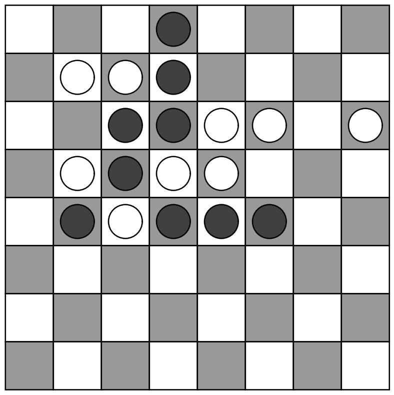
(a)



(b)

**Figure 1:** (a) Initial position, showing standard designations for rows and columns. (b) Possible moves for the black piece at f3.

[1]The examples here are taken from the BoardSpace website at http://www.boardspace.net. The page allows a Java-enabled browser to find the legal moves for any of the other pieces as well.


(a)

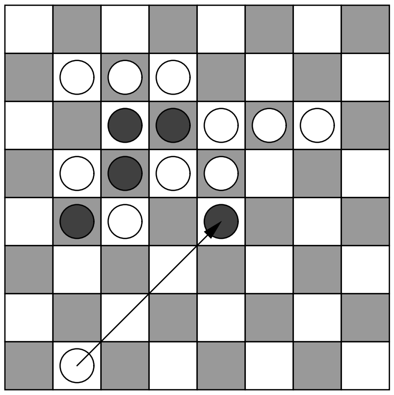

(b)

**Figure 2:** End positions. Position (a) is a win for Black. In position (b), White can move as shown, capturing an isolated black piece and giving *both* players contiguous pieces. Since it is White's move, however, the result is counted as a winning position for White.

The game ends when one side's pieces are contiguous: that is, there is a path connecting any two pieces of that side's color by a sequence of steps to adjacent squares (horizontally, vertically, or diagonally), each of which contains a piece of same color. Hence, when a side is reduced to a single piece, all of its pieces are contiguous. If a move causes both sides' pieces to be contiguous, the winner is the side that made that move (thus, there are no ties). Figure 2a shows a final position. Figure 2b shows a board just before a move that will give both sides contiguous pieces. Since the move is White's, White wins this game.

### 1.1 Notation

We'll denote columns with letters a–h from the left and rows with numerals 1–8 from the bottom, as shown in the illustration of the initial position on the previous page. The square at column $c$ and row $n$ is denoted $cn$. A move from $c_1n_1$ to $c_2n_2$ is denoted $c_1n_1$-$c_2n_2$.

## 2 Textual Input Language

Each execution of the program will play a single game, depending on the command-line arguments (see §4). When you are supposed to input a move on the standard input, it should consist of a single line using the notation described in §1.1. At any time, you may also enter any line beginning with '`s`' (for "show"), which should cause the program to print the current state of the board in the format described in §3. Any line beginning with '`q`' should cause the program to exit, abandoning the current game. The end of file on the standard input should have the same effect.

Leading whitespace is ignored in all commands. Each of the commands described here consists of a single word (sequence of non-whitespace characters), delimited by whitespace or the end of input. Anything after the first command on a line is treated as commentary and is ignored. For example:

```
    b8-d6   Move toward the center.
```

A command that starts with '`#`' is a comment; it and anything after it on the line should be ignored. Blank lines likewise are ignored.

Players that are AIs only start delivering moves after you issue a command starting with '`p`' (for "play"). At that point the AI(s) take over input for its (their) respective player(s). Before the '`p`' command, any moves entered on the standard input are interpreted as ordinary moves by Black or White, alternately, just as if two human players were playing. Subsequent '`p`' commands have no effect. This arrangement allows you to create input files that set up an initial position before starting an AI.

As long as the commands described so far work properly, you may add any additional commands you want.

**Errors.** Moves must be legal, or your program must reject them without affecting the board. Humans are expected to make errors, your program should ask for another move when this happens. Similarly, your program should respond to other invalid commands by simply reporting the error and prompting for a new command. AIs must never make illegal moves.

**Input Command Summary.** Any of the following commands may be followed by arbitrary text, except that moves must be delimited on the right by whitespace.

| **Command** | **Effect** |
|---|---|
| s | Show the current board, side next to play, and number of moves, using exactly the format described in §3. |
| p | Start any AIs specified on the command line. Has no effect if there are none or if they are already started. |
| q | Quit. Exit the program. |
| # | Comment. Ignored. |
| $c_1r_1$-$c_2r_2$ | Move piece on $c_1r_1$ to $c_2r_2$ (e.g., `b8-b6`.) |

## 3 Output

The '`s`' command should print the board in *exactly* the following format:

```
===
  - b b b b b b -
  w - - - - - - w
  w - - - - - - w
  w - - - - - - w
  w - - - - - - w
  w - - - - - - w
  w - - - - - - w
  - b b b b b b -
Next move: black
Moves: 0
===
```

Here, '`-`' indicates an empty square, '`b`' indicates a black piece, and '`w`' indicates a white piece. The "Next move:" line tells which side moves next (ignoring whether the game has ended at this point), and the "Moves:" line tells how many moves have been made since the beginning of the game. Don't use the two '`===`' markers anywhere else in your output. This command gives the autograder a way to determine the state of your game board at any point. It does not change any of the state of the program.

Each time the program expects a move, it should prompt for it. You may prompt however you please with a string that ends with `>` followed by any number of blanks (one does not typically print a newline after a prompt.) Write prompts to the standard output. It is probably wise to "flush" `System.out` explicitly after printing a prompt with

```
System.out.flush();
```

Do not print a `>` character except as a prompt.

Whenever an AI moves, your program should print the move on the standard output using *exactly* the following formats:

`W::a2-c2` *for a move by White*
`B::g1-g3` *for a move by Black*

As for '`===`', do not use '`::`' for other messages. Do *not* echo moves by human players or setup moves with this notation.

When one side or the other makes the winning move, print one of the messages

```
White wins.
Black wins.
```

(on a separate line) as appropriate (the rules do not allow draws). Do not print these messages at any other time. Your program should not exit until it receives a 'q' (quit) command or reaches the end of its input.

You are free to produce any other output you want, subject to the restrictions above (which are there to make autograding easier). So, for example, you might want to print the board automatically from time to time, especially when at least one player is an AI. As long as you do so without using the '`===`' markers, you are free to produce whatever output you want.

## 4 Running Your Program

Your job is to write a program to play Lines of Action. Appropriately enough, we'll call the program LOA. To run your program, one a command line with the format

```
java loa.Main  [ --white ] [ --ai=N ] [ --seed=N ] \
               [ --time=LIM ] [ --debug=N ] [ --display ]
```

Square braces indicate optional parameters.

`--white` indicates that you play White (by default, you play Black.)

`--ai=`*N* where *N* is either 0, 1 (the default), or 2, indicates how many of the players are AIs. If $N = 0$, both players are human (and all moves are entered from the keyboard). If $N = 1$ (the default), then the opponent is an AI. If $N = 2$, then both players are AIs (an AI enters moves instead of you.) The AIs do not actually start playing until you enter a '`p`' command. As a result, you can enter an arbitrary set of moves to set up a position before actually starting an AI.

`--seed=`*N* where *N* is any long integer, selects a seed for a pseudo-random number generator. This parameter may have no effect if you have not used randomness in your AIs. However, if you do, then two runs of your program with the same seed and the same inputs should have the same output.

`--time=`*LIM* where *LIM* is an integer or floating-point number, sets a time limit (in minutes) for each side's total moves. It is not necessary to stop immediately when one side runs out of time, but if the time limit is exhausted when a side is about to move, the game is over and the other player should be announced as the winner. If at least one player is an AI, timing does not start until the '`p`' command is entered.

`--debug=`*N* where *N* is an integer. Use this as for project 2. Your program should *not* produce debugging output except in the presence of this switch.

`--display` indicates that you will communicate through a graphical interface (GUI). Your program must work without this option (that's how we test it). It is an error to use this option if your program does not provide a GUI.

If this command line is erroneous, your program should print a message and exit with a return code other than 0, as is conventional. Otherwise, your program should exit with exit code 0. For this project, even if someone enters invalid commands during a session, your program should exit normally (and, of course, print error messages in response to the error on the standard error output.)

## 5 Your Task

Please read *General Guidelines for Programming Projects.* The staff directory will contain skeleton files for this project in `proj3`.

Be sure to include tests of your program (that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them. To help with testing and debugging, we will provide our own version of the program, so that you can test your program against ours (we'll be on the lookout for illegal moves). More details will follow.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that

```
gmake check
```

runs your tests.

Your AI should at least be able to find forced wins within a small number of moves. Otherwise, we won't be too particular. In fact, we suggest that you first aim to produce an AI that is simply capable of making legal moves.

We'll be running a tournament among programs that pass our tests. The tournament will use a `--time` value of "several" minutes.

### 5.1 Extra Credit

For extra credit, you can implement the `--display` option, and play the game using a graphical user interface (GUI). Don't even think about this until you get your project working! However, you might consider how to structure your solution to make the addition of a GUI simple. The package `ucb.gui` contains classes that make it pretty easy to construct a simple GUI. You will also have to examine the classes `java.awt.Graphics` and `java.awt.Graphics2D` to see how to draw things.

## 6 Advice

At first glance, it might seem that with all the options available, the program would have to be a mass of **if** statements covering all possible cases. But with proper use of abstraction, this does not have to be.

We've included documentation for the `ucb.util` package in the on-line materials. Look particularly at `CommandArgs` and `StopWatch` or `Ticker`. If you need random numbers, take a look at `java.util.Random` and Chapter 11 of *Data Structures (Into Java).*

I suggest working first on the class `Board`, which is supposed to represent the game board and state of play. Next, implement the human player. Then you can tackle writing a machine player. Start with something really simple (perhaps choosing a legal move at random) and introduce strategy only when you get everything working properly.