CS61B, Fall 2015 **Project #2: Lines of Action** **P. N. Hilfinger**

Due: Tuesday, 17 November 2015 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 contains both the squares behind and in front of the piece that moves, as well as the square the piece is on. A piece may not move off the board, onto another piece of its color, or over an opposing piece.

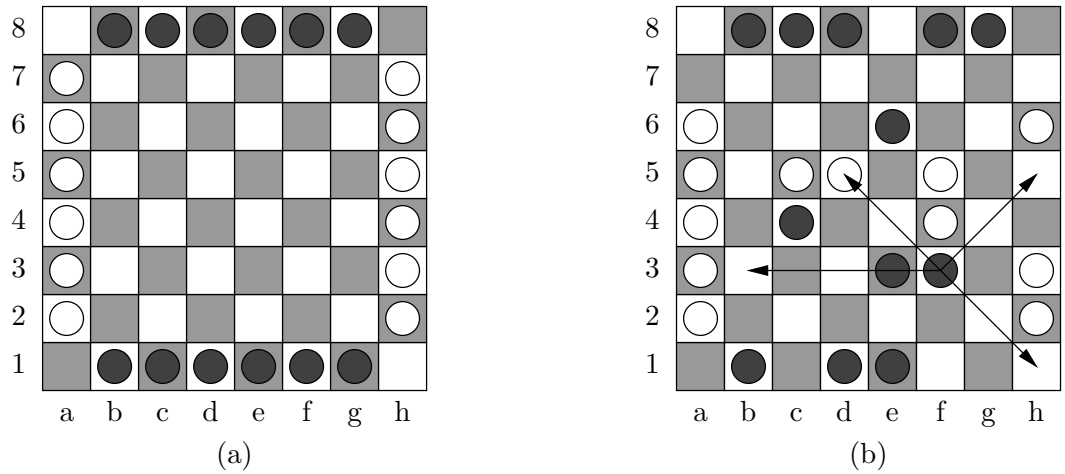
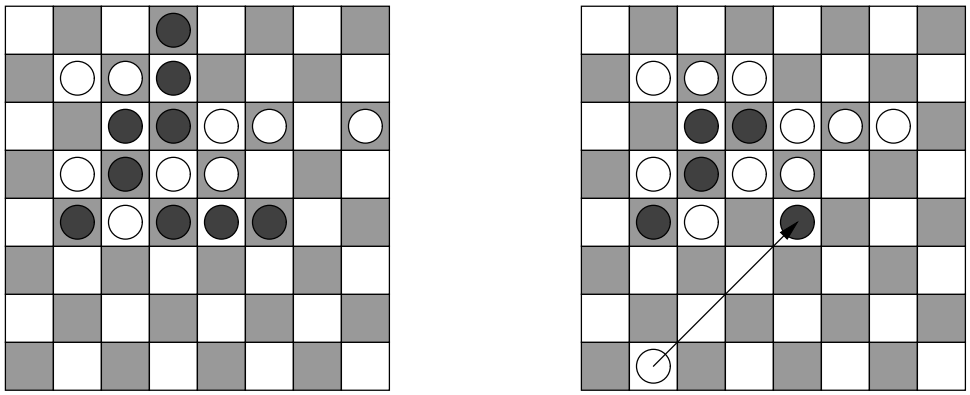


Figure 1: (a) Initial position, showing standard designations for rows and columns. (b) Possible moves for the black piece at f3.

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. The diagonal moves are all two

[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.

squares, since there are two pieces along each of the diagonals shown. The horizontal move is four squares because of the four pieces in row 3.

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[2]). 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

Your program should respond to the following textual commands (you may add others). There is one command per line, but otherwise, whitespace may precede and follow command names and operands freely. Empty lines have no effect, and a command line whose first non-blank character is '`#`' is ignored as a comment. Extra arguments to a command (beyond those

[2]One can have infinite games, where players just repeat positions indefinitely. In some versions of this game, there are rules that repeated positions lead to ties. We're not doing that here, but since our testing will always include time limits, somebody will eventually lose if two players repeat positions many times.

specified below) are ignored. An end-of-file indication on the command input should have the same effect as the '`quit`' command.

`clear` Abandons the current game (if one is in progress), and clears the board to its initial configuration. Playing stops until the next `start` command.

`start` Start playing from the current position, if not doing so already (has no effect if currently playing). Takes moves alternately from Black and White.

c_1r_1`-`c_2r_2 As indicated in §1.1, indicates a move from c_1r_1 to c_2r_2 (e.g., `b8-b6`.) This command is valid only during play (after `start` is issued). The first and then every other move is for the Black player, the second and then every other is for White, and the normal legality rules apply to all moves.

`auto` P Stops the current game until the next `start` command and causes player P to be played by an automated player (an AI) on subsequent moves. The value P must be "black" or "white" (ignore case—"black" or "BLACK also work). Initially, White is an automated player.

`manual` P Stops the current game until the next `start` command and causes player P to take moves from the terminal on subsequent moves. The value of P is as for the `auto` command. Initially, Black is a manual player.

`set` CR P Stop any current game. Depending on P, set the contents of square CR (see §1.1). P is either '`b`', '`w`', or '`e`' (for Black, White, or empty) indicating what to place in the given square. When P is `b`, the next player to move when play resumes (as a result of `start`) becomes White, and when P is `w`, the next player to move becomes Black.

`dump` This command is especially for testing and debugging. It prints the board out 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
===
```

with the '`===`' markers at the left margin (indicated by the vertical line above) and the board indented four spaces. Here, '`-`' indicates an empty square, and 'w' and 'b' indicate white or black pieces. Don't use the two '`===`' markers anywhere else in your output. This 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.

`seed` *N* *If* your program's automated players use pseudo-random numbers to choose moves, this command sets the random seed to *N* (a long integer). It has no effect if there is no random component to your automated players (or if you don't use automated players them in a particular game). It doesn't matter exactly how you use *N* as long as your automated player behaves identically each time it is seeded with *N*. In the absence of a `seed` command, do what you want to seed your generator. The idea behind `seed` is to make it possible to have reproducible results when testing an AI.

`help` Print a brief summary of the commands.

`quit` Exits the program.

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.

3 Output

Each time the program expects a move from a human player, it should prompt. 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.

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, you'll type

```
java loa.Main
```

5 Your Task

Please read *General Guidelines for Programming Projects.* The shared directory will contain skeleton files for this project in `proj2`.

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.

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 eventually run a tournament among programs that pass our tests.

6 Testing Harness and Staff Solution

There is a staff solution available to play against or to better understand the spec. To run it, type `staff-loa` on the instructional machines. It is also available in `cs61b-software` for those of you at home (you'll need to `git pull --rebase` that directory to get it). At the moment, it just has textual output; there is no GUI.

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

```
make check
```

runs your tests.

We have provided a testing script, `test-loa`, which is available on the instructional machines and in the `cs61b-software` branch of the shared repository. You will need it to run `make check` on your home setup. The command format is

```
test-loa FILE1.in FILE2.in ...
```

which runs the `.in` files and, if there are corresponding `FILE.out` files, checks outputs. This testing harness is a bit elaborate. It allows you to run a single program or to run two separate programs against each other, feeding moves output by one (as `W::...` or `B::...` to the other. To allow testing of automated players (whose moves may be unpredictable), it permits you to specify places where the test script is to accept any valid moves and to communicate moves from one program to the other. Type plain '`test-loa`' (without arguments) to see documentation of the format of `.in` files. When there is a `.out` file for a given input file, only the dumps and win messages are compared (that's why it is important to use a standard format for these: it allows `teat-loa` to extract the relevant output.)

To diagnose a problem with a particular test, use

```
test-loa --verbose FILE1.in ...
```

This will print the actual input to and output from each program. Lines starting "`>Program 1:`" indicate input to the first (or only) program; "`<Program 1:`" lines indicate output from the first program; "`<Program 1(e):`" lines indicate error outputs from the first program. Likewise for the second program (if used).

6.1 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.

7 Advice

We've implemented some fussy bits or possibly useful classes in the skeleton. Always remember, however, that you don't have to use them. Your job is to provide tests and make `java loa.Main` conform to the specification, period. 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 representation of the board (the skeleton has a class `Board`, which is supposed to represent the game board and state of play.) We have also included a skeleton `Game` class that uses `Board` to keep track of a series of games and interpret commands. Get this to work with a manual ("human") player first. Then you can tackle writing an automated player `MachinePlayer`. Start with something really simple (perhaps choosing a legal move at random) and introduce strategy only when you get the basic game-running machinery working properly.