Navigation

Project 2: Lines of Action

Please check Piazza for updates, useful resources, and a changelog.

Introduction

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 connected. Two pieces are connected if they are adjacent horizontally, vertically, or diagonally. Initially, the pieces are arranged as shown in Figure 1. 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.

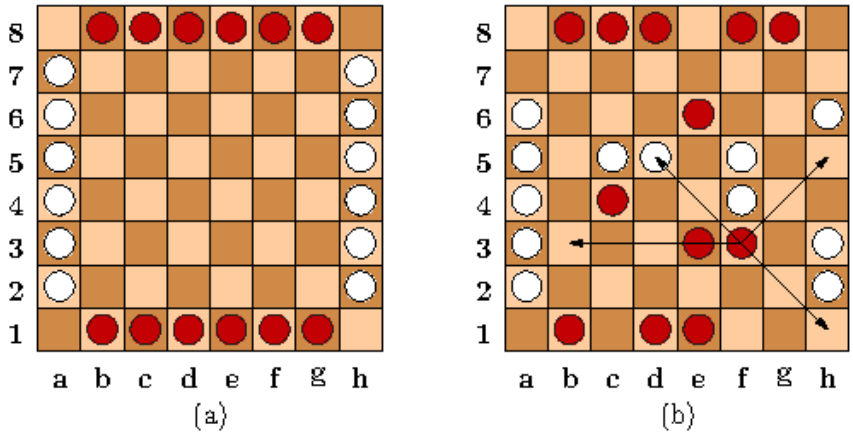

Figure 1. (a) Initial position, showing standard designations for rows and columns. (b) Possible moves for the black piece at f3. In (b) the piece at f3 cannot move horizontally to the right because it would be forced to move 4 steps, which would push it off the board. The same piece cannot move vertically up because there are 4 pieces in the vertical line of action and moving 4 steps up would require it to move over enemy pieces

Figure 1b illustrates the four possible moves for a black piece in the position shown. (The examples here are taken from the BoardSpace website.) The move f3-d5 is a capture; all others are ordinary moves. The diagonal moves are all two 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. One can have infinite games, where players just repeat positions indefinitely. We will prevent this with a move-limit rule: if the current move limit is L moves (the default is 60), then after the two sides both make L moves combined, the game ends in a tie. Our testing will always include time limits; somebody will eventually lose if two players repeat positions many times. 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.

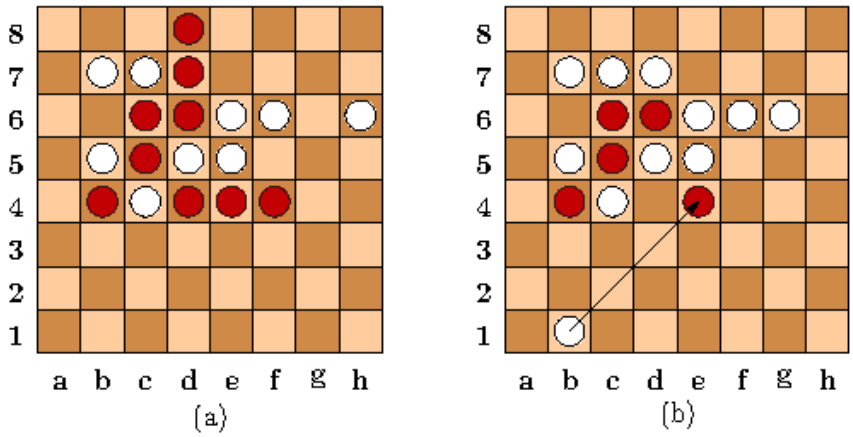

Figure 2. End positions. Position (a) is a win for Black. In position (b), White can move as shown, capturing an isolated black piece and giving *both* players contiguous pieces. Since it is White's move, however, the result is counted as a winning position for White.

Notation

We'll denote columns with letters a–h from the left and rows with numerals 1–8 from the bottom, as shown in Figure 1 and Figure 2. 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

Textual Input Language

Your program should respond to the following textual commands (you may add others). There is one command per line, but 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 specified below) are ignored. An end-of-file indication on the command input should have the same effect as the `quit` command.

- **new**: Abandons the current game (if one is in progress), and clears the board to its initial configuration. Sets the current player to Black. Takes moves alternately from Black and White.
- c_1r_1 –c_2r_2 : As indicated under Notation, denotes a move from c_1r_1 to c_2r_2 (e.g., `b8-b6`.) This command is not valid after a game has ended and before the board has been cleared for a new game. 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: 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: 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 N: Depending on P, sets the contents of square CR. P may be `black`, `white`, or `-` (denoting empty). Make N (`black` or `white`) the next player to move. As for `auto` and `manual`, case is irrelevant. This command is intended for setting up particular positions quickly for testing or study, and is not intended for normal play.
- **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 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 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. For example,

```
import java.util.Random;
Random r = new Random();
r.setSeed(100000000);
System.out.println(r.nextInt(100));
System.out.println(r.nextInt(100));
```

(The call `nextInt(int r)` returns an integer between 0 and the `r`.) When we run this program, we will deterministically get the same output every time. For example, if we get 80 and then 30 printed, the next time we run the program, 80 and 30 will print again, even though the `nextInt` method returns a "random" number (this is why we officially call these "pseudo-random

numbers.")

- **limit** N Sets the move limit to $2 * N$ (initially, it is $N = 30$). If both players make N moves and neither side has won, the game is a draw. The value of N may not be less than or equal to the number of moves made so far by either player.
- **help**: Prints 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.

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 format:

```
* a2-c2
```

(with asterisk shown). Do not print these lines out for a manual player's moves.

When one side wins, the program should print out one of

```
* White wins.
* Black wins.
* Tie game.
```

(also with periods) as appropriate. Do not use the `*` character in any other output you produce.

Your program should not exit until it receives a `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.

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 -ea loa.Main [--display] [--log=LOGFIL
```

Here, square braces enclose optional arguments. The `--display` option applies only if you do the extra-credit GUI. The `--log` argument specifies a file into which the program writes all the commands and moves entered into it (good for capturing the details of a session). The `INPUT-FILE` specifies a source of commands; by default it is simply the standard input (generally your terminal). Finally, the `OUTPUT_FILE` specifies a file to receive output printed by the program; by default it is simply the standard output.

Your Task

The shared repository will contain skeleton files for this project, which you can merge into your repository as usual with

```
git fetch shared
git merge shared/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).

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. A forced win in N moves is a situation in which one side can always win after making N moves with proper play, no matter what the opponent does. We'll eventually run a tournament among programs that pass our tests. If you want to understand how to implement your AI, please watch Lecture 22 (Game Trees).

As usual, we'll run a style checker on your code. The skeleton contains deliberate style errors that are there to draw your attention to certain things, including skeletons for extra-credit code. Just because you didn't write the lines that cause them, don't think you can ignore them. You are responsible for all the code you hand in, regardless of who originally wrote it.

Testing Harness and Staff Solution

There is a staff solution available to play against or to better understand the spec. Type `staff-loa` on the instructional machines to run it. It takes the same arguments as your solution is supposed to take.

Our testing of your projects will be automated. The testing program will be finicky, so be sure that

```
make check
```

runs your tests on the instructional machines.

The integration test program, `test-loa`, feeds commands to one or two running programs and passes appropriate moves from one to the other, allowing you to test your program and to test it against another program (such as the staff version). To run `test-loa`, you'll use

```
python3 testing/test-loa TESTFILE-1.in
```

to run `TESTFILE-1.in` through your program and

```
python3 testing/test-loa TESTFILE-1.in TESTFILE
```

to run two programs simultaneously so that each one sends all of its AI's moves (such as "`* b4-d6`" as described previously) to the other program. (Replace "TESTFILE" with the actual name of your test file.)

Each `.in` file should start with a Unix-style command for running a program preceded by "`#*`", such as

```
#* java -ea loa.Main
```

(You will probably use just the command command above; the autograder will sometimes replace it with an invocation of `staff-loa`.) The rest of the `.in` file is fed to this program as the standard input, except for lines that start with "`#*`" in the first column, which are special instructions to the testing script.

- The command `#*time` *MOVE GAME* puts a time limit of *MOVE* seconds on each move in a game and *GAME*

seconds for one side's moves in an entire game (i.e., an entire sequence of moves controlled by one of the `move/win` commands below).

- The command `#*move` means "wait for the program to output an AI move, and then continue with the script." When used with the two-argument form of `test-loa`, it also sends this move as input to the other program.
- The command `#*move/win` is intended for use when both players are AIs, and means "wait for the program to output a complete sequence of AI moves, followed by "`* ... wins.`" It does not print either the moves or the "win" message.
- The command `#*move/win+` is the same as `#*move/win`, but also prints the "`* ... wins.`" message.
- The command `#*win+` waits for a "`* ... wins.`" message from the program and prints it.
- All lines that don't start with `#*` are sent to the program being tested.

A few other commands apply only to the two-argument form of `test-loa`. They are intended to allow two programs to play each other.

- The command `#*remote move/win` means "Wait for an AI move from the other program, give it to this program, then execute a `#*move` command. Repeat until one side sends a win message. Do not print the moves or win message.

- The command `#*remote move/win+` is the same as `#*remote move/win`, but prints the "win" message.

The idea with these two commands is that one of the two scripts will, at a certain point, contain the commands

```
#*move
#*remote move/win
```

and the other will contain

```
#*remote move/win
```

so that the first sends a move from its AI to the other program, which then waits for a response from its AI to send back, and so forth.

For the `remote` commands, both programs should generate "wins" messages, and `test-loa` will check that they are the same.

The `test-loa` script throws out any other output from either program except for properly formatted board dumps, as are supposed to be produced by the **dump** command described previously. You can see all the output by running it with

```
python3 testing/test-loa --verbose TESTFILE-1.
```

or

```
python3 testing/test-loa --verbose TESTFILE-1.
```

which will show all the commands sent to each program and all their output.

The `test-loa` program will report an error if a program hangs or times out, or if it exits abnormally (with an exception or an

exit code other than 0). Finally, if there is a file `TESTFILE-1.std` or `TESTFILE-2.std`, `make check` will check it against the output from the program for `TESTFILE-1.in` (likewise for `TESTFILE-2.std` against the output for `TESTFILE-2.in`); `make check` uses on the script `testing/tester.py` to do this comparison.

Checkpoint

Information about the checkpoint is on Piazza here. The checkpoint is due Friday April 3 11:59 PM.

Extra Credit

Instructions to submit the EC are on Piazza.

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.gui2` 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.

We will be manually grading the GUI and awarding points based on this rubric:

- 2 pt - displays the current board, can make moves

- 1 pt - can show a help page
- 1 pt - can undo moves
- 1 pt - can switch a player (black or white) between AI and manual midway through the game

---

Advice & Resources

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 -ea 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.

The TAs have created helpful walthrough videos for the `numContig` method in the `Board` class, `findMove` for the `MachinePlayer` class, and heuristics for the `MachinePlayer` class. They can be found on Piazza here.