Navigation

Project 2: Amazons

Introduction

The game of *Amazons* is a simple but rather interesting board game, usually for two players. It was invented in 1988 by Walter Zamkauskas of Argentina, and originally called *El Juego de las Amazonas* (now a trademark of Ediciones de Mente). The board is a 10 by 10 chessboard. Each player gets four "amazons" (represented as chess queens), white for the player who moves first, and black for the opponent. Initially, the board is set up as in the diagram on the left of Figure 1.

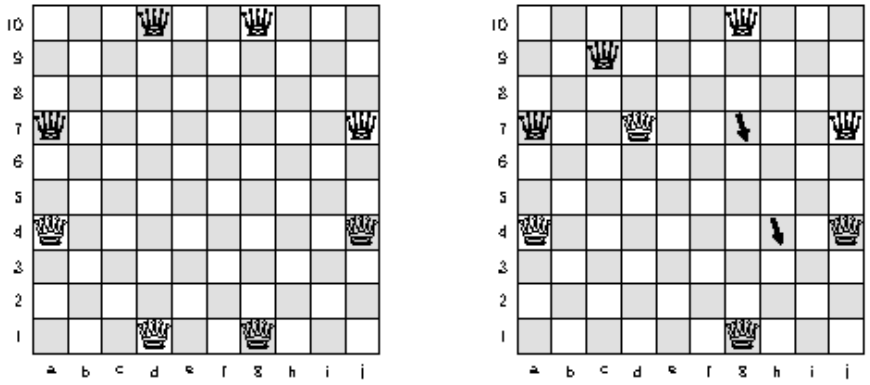

Figure 1. On the left: an Amazon board showing the standard numbering of squares, and the initial placement of the pieces. On the right: the board that results after the two moves `d1-d7(g7)` and `d10-c9(h4)`.

Each move consists of two parts: first, an amazon of the appropriate color makes a chess queen move—any non-

Navigation

vertically, or diagonally with the restriction that the piece may not move onto or through an occupied square. Pieces are not captured in this game; the board keeps getting fuller. Next, the piece that moved "throws a spear" from her final position. Spears move exactly like pieces, and are subject to the same restrictions. The spear sticks permanently in the (previously empty) square in which it lands. That square is counted as being occupied for the rest of the game (so no piece or spear may move through it). For example, the right diagram in Figure 1 shows the result of two moves. From the initial position, white moved from `d1` to `d7` and from there threw a spear to `g7`. Then black moved from `d10` to `c9`, and from there threw a spear to `h4`.

A player who has no legal move loses. For example, after black moves `a7-a6(a7)` in Figure 2, he will have used up all his moves, and white (who still has plenty of room) will win after his next move. Draws are impossible.

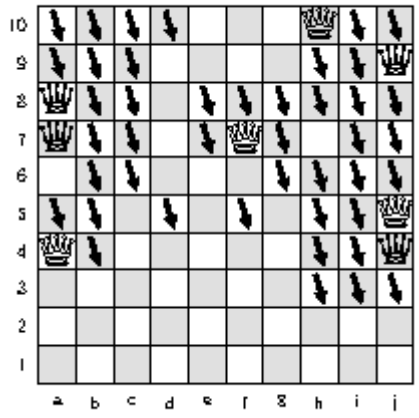

Figure 2. A lost position for black. After `a7-a6(a7)`, black has no move

Navigation

Notation

A square is denoted by a column letter followed by a row number (as in `e4`). Columns are enumerated from left to right with letters a through j. Rows are enumerated from the bottom to the top with numbers 1 through 10. An entire move then consists of the starting and ending position of the piece that is moved, followed, in parentheses, by the position to which the spear is thrown. Thus, `d10-c9(h4)` means "Move from `d10` to `c9` and then throw a spear to `h4`."

Commands

When running from the command line, the program will accept the following commands, which may be preceded by whitespace.

- **new**: End any game in progress, clear the board to its initial position, and set the current player to white.
- A move, either in the format described in Notation or (for convenience) with blanks replacing punctuation, as in `g1 c5 e7`.
- **seed** *N*: If the AIs are using random numbers for move selection, this command seeds their random-number generator with the integer *N*. Given the same seed and the same opposing moves, an AI should always make the same moves. This feature makes games reproducible.
- **auto** *C*: Make the *C* player an automated player. Here, *C* is "black" or "white", case-insensitive.

(entering moves as manual commands).

- **dump**: Print the current state of the board in ***exactly*** the following format:

```
===
   - - - - - - B - - -
   - - B - - - - - - -
   - - - - - - - - - -
   B - - W - - S - - B
   - - - - - - - - - -
   - - - - - - - - - -
   W - - - - - - S - W
   - - - - - - - - - -
   - - - - - - - - - -
   - - - - - - W - - -
===
```

  Here, `B` denotes a black queen, `W` a white queen, and `S` a spear. You must not use the `===` lines for any other output).

- **quit**: Exit the program.

Feel free any other commands you think might be nice.

Output

When an AI plays, it should print out each move that it makes using exactly the format

```
* a1-c3(c6)
```

(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.
```

character in any other output you produce.

You may prompt a manual player for input using the form

```
...>
```

where "..." may be any text. The grading scripts will discard any text from the beginning of a line up to a > character.

Navigation

Your Task

Your job is to write a program to play Amazons. To run it in text mode, use the command

```
java -ea amazons.Main
```

to enter commands from the terminal or use

```
java -ea amazons.Main INPUT
```

to feed it commands from file INPUT.

The AI in your program should be capable of finding a win that is within 10 moves. The branching factor for Amazons is quite high at the beginning of a game, but rapidly declines thereafter. Therefore we suggest that you choose a maximum search depth for a move that depends on how full the board is. Experiment a bit to see what works. The autograder will allow 3 minutes for a fully automated game.

The GUI is an optional (extra credit) part of this project. We will actually do automatic testing only on the commands

```
java -ea amazons.Main
```

```
java -ea amazons.Main INPUT
```

Staff Program

The `staff-amazons` program on the instructional machines runs our solution to the project. This version has additional bells and whistles that you are ***not*** required to duplicate. It is ***not*** the standard for this project, just as example of a solution. In particular, your GUI (if you do it) need not look anything like ours. The autograder will use this program in some of its tests to check your program's output.

Testing

We have only provided a token `UnitTest` file; you can add additional unit test files and list them in `UnitTest` so that they all get run by

```
java -ea amazons.UnitTest
```

(which is what `make check` does).

The integration test program, `test-amazons`, is not part of your skeleton. We will be providing it slightly later. To run `test-amazons`, you'll use

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

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

```
python3 testing/test-amazons TESTFILE-1.in TESTFI
```

sends all of its AI's moves (such as `* c3-f6(16)` as described previously) to the other program. (Replace "TESTFILE" with the actual name of your test file.)

Each `.in` and input file should start with a Unix-style command for running a program, such as

```
java -ea amazons.Main
```

(You will probably use just this command; the autograder will sometimes use this line to run the staff solution against your program.) 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-amazons`, 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.

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-amazons`. 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-amazons` will check that they are the same.

The `test-amazons` script throws out any other output from either program except for properly formatted board dumps, as are supposed to be produced by the **dump**

by running it with

```
python3 testing/test-amazons --verbose TESTFILE-1
```

or

```
python3 testing/test-amazons --verbose TESTFILE-1
```

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

The `test-amazons` 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`, `test-amazons` 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`).

Extra Credit

First get your program working, and then, if you feel the urge, try the extra-credit GUI (Graphical User Interface). If you do, the option

```
java -ea amazons.Main --display
```

should create the GUI. If you don't implement the GUI, this option should cause your program to exit with a non-zero code via `System.exit(2)`. Your GUI does ***not*** have to look at all like ours.

Advice

project, feel free to get together with other students to discuss ideas and plan strategies. Of course, you should always feel free to consult your TA or me.

The board is an obvious place to start. We have provided suggestions for methods that you can use if you want, but you are not required to do so. We have structured the skeleton so that the different kinds of player (ordinary human at the keyboard using text commands, AI, or human using a GUI) are represented as different subtypes of a type `Player`, an example of how using OOP can cut down on pervasive conditional tests for types of player.

Navigation