CS61B, Fall 2013 **Project #2: Jumping Cubes, version 3** **P. N. Hilfinger**

Due: Friday, 15 November 2013

1 Background

The KJumpingCube game[1] is a simple two-person board game. It is a pure strategy game, involving no element of chance. For this second project, you are to implement our version of this game, which we'll call `jump61`, allowing a user to play against a computer or against another person, or to allow the computer to play itself. The basic interface is textual, as before, but for extra credit, you can produce a GUI interface for the game.

2 Rules of Jump61

The game board consists of an $N \times N$ array of squares, where $N > 1$. At any time, each square may have one of three colors: red, blue, or white (neutral), and some number of *spots* (as on dice). Initially, all squares are white and have no spots.

For purposes of naming squares, we'll use the following notation: $r : c$ refers to the square at row r and column c, where $1 \leq r, c \leq N$. Rows are numbered from top to bottom (top row is row 1) and columns are numbered from the left.

The *neighbors* of a square are the horizontally and vertically adjacent squares (diagonally adjacent squares are not neighbors). We say that a square is *overfull* if it contains more spots than it has neighbors. Thus, the four corner squares are overfull when they have more than two spots; other squares on the edge are overfull with more than three spots; and all others are overfull with more than four spots.

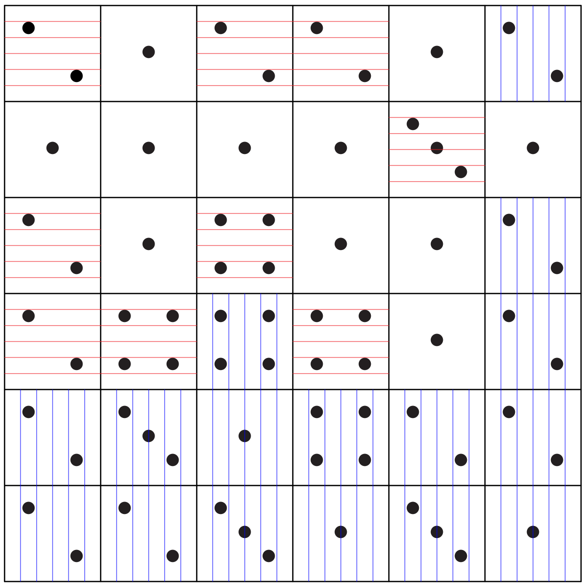
There are two players, whom we'll call Red and Blue. The players each move in turn, with Red going first. A move consists of adding one spot on any square that does not have the opponent's color (so Red may add a spot to either a red or white square). A spot placed on a white square colors that square with the player's color. After the player has moved, we repeat the following process until no square is overfull or all squares are the same color:

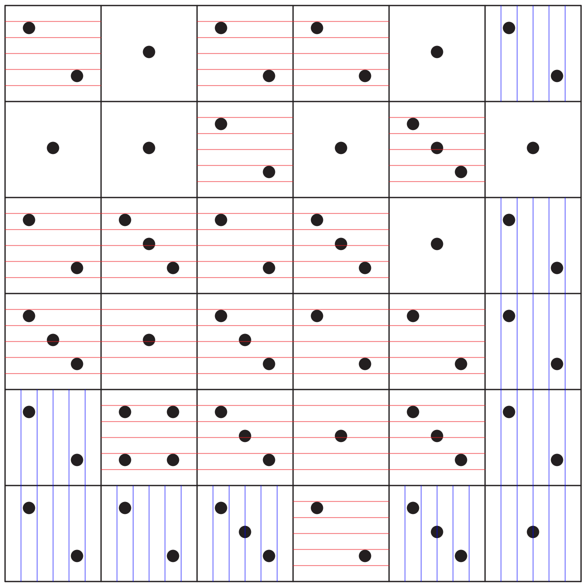
1. Pick an overfull square.
2. For each neighbor of the overfull square, move one spot out of the square and into the neighbor.
3. Give each of these neighboring squares the player's color (if they don't have it already).

The order in which this happens, as it turns out, does not usually matter—that is, the end result will be the same regardless of which overfull square's spots are removed first, with the exception that the winning position might differ. A player wins when all squares are the player's color.

[1]Distributed under the GNU Public License as part of the KDE project. Copyright 1999, 2000 by Matthias Kiefer. It was inspired by an old game on the Commodore 64.

For example, given the board on the left ($N = 6$), if Red adds a spot to square 3:3, we get the board on the right after all the spots stop jumping.





Legend: ▤ ▥

Red square: Blue square:

The rules hold that the game is over as soon as one player's color covers the board. This is a slightly subtle point: it is easy to set up situations where the procedure given above for dealing with overfull squares loops infinitely, swapping spots around in an endless cycle, unless one is careful to stop when a winning position appears.

3 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 everything from a '`#`' character to the end of a line is ignored as a comment. Extra arguments to a command (beyond those specified below) may be 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 (all squares neutral). 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 Red and Blue according to their color and the current move number.

`quit` Exits the program.

`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 "red" or "blue" (ignore case—"Red" or "RED" also work). Initially, Blue 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, Red is a manual player.

`size` *N* Stops any current game, clears the board to its initial configuration, and sets the size of the board to *N* squares. Initially, $N = 6$.

`move` *N* Stop any current game, set the number of the next move *N*. You usually use this command after setting up an initial position (with `set` commands.) Initially, the number of the next move is 1.

`set` *R C N P* Stop any current game. Put *N* spots at row *R* and column *C* (see §3.1). *P* is either '`b`' or '`r`' (for blue or red,) indicating the color of the square. When *N* is 0, *P* is ignored and the square is cleared.

`dump` This command is especially for testing and debugging. It prints the board out in *exactly* the following format:

```
===
    2r -- 2r 2r -- 2b
    -- -- 2r -- 3r --
    2r 3r 2r 3r -- 2b
    3r 1r 3r 2r 2r 2b
    2b 4r 3r 1r 3r 2b
    2b 2b 3b 2r 3b 1b
===
```

with the '`===`' markers at the left margin and other lines indented four spaces. Here, '`--`' indicates a neutral square, '*N*`r`' indicates a red square with *N* spots, and '*N*`b`' indicates a blue square with *N* spots. 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). This command has no effect if there is no random component to your automated players (or if you don't use 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.

`help` Print a brief summary of the commands.

3.1 Entering Moves

To enter moves from the terminal (for a manual player), use a command of the form

R C Adds a spot to the square at row R, column C, where R and C are integers in the range 1 to the current board size. Rows are numbered from the top, columns from the left. After the spot is added, spots are redistributed as indicated in the rules above. Like other commands, R and C may be surrounded by any amount of whitespace. Illegal moves must be rejected (they have no effect on the board; the program should tell the user that there is an error and request another move).

The first and then every other move is for the red player, the second and then every other is for blue, and the normal legality rules apply to all moves.

4 Output

When either player enters a winning move, the program should print a line saying either `"Red wins."` or `"Blue wins."` as appropriate. Use exactly those phrases.

When an AI plays, print out the moves that it makes using the format "*P* `moves` *R C*`.`", where *P* is `"Red"` or `"Blue"` and R and C are the row and column numbers at which a spot is added. Use exactly that format. Do not print this message for manual moves.

Finally, as indicated in §3, the '`dump`' command must print out in exactly the format shown.

Otherwise, you are free to prompt for input however you want, and to print out whatever user-friendly output you wish (other than debugging output or Java exception tracebacks, that is). For example, you will probably want to print the game board out after each move is complete (this is distinct from printing the board in the special format required by '`dump`').

When users enter erroneous input, you should print an error message, and the input should have no effect (and in particular, the user should be able to continue entering commands after an error). Make sure that any of this extra output you generate is distinct from the outputs that are required (otherwise, the testing software will flag your program as erroneous.) Regardless of whether users have made errors during a session, your program should always exit with code 0.

5 Other Requirements

On the instructional machines, the Unix command '`make`' must compile your program and '`make check`' must run all your tests. The command '`java jump61.Main`' must run your program, and '`java jump61.UnitTest`' must run your unit tests. In order to be submitted, your program will have to pass the style check, as in previous assignments. When testing your program, we will use the command

```
    java -ea jump61.Main
```

to run it (the '`-ea`' insures that all assertions in your program are checked.)

As before, we will evaluate your project in part on the thoroughness of your testing. Put JUnit test classes with names ending in `Test` (e.g., `BoardTest.java`) in your `jump61` package, including one particular class named `UnitTest`, which should run *all* your individual JUnit tests as a *suite* (see the skeleton). Our autograder script will run `UnitTest` in the JUnit framework, expecting it to pass. For black-box testing, we've provided a program '`test-jump61`', which you can find on the instructional servers in the file `~cs61b/bin/test-jump61`. On Unix home systems (including Macs), you can simply copy this to a directory in your path and use it as you can on the instructional machines. This program allows you to run your program, supply it with input, and check the result, or to run two programs, and play them against each other (taking output like `"Red moves 1 1"` from one program and feeding its move to the other.) See the comments at the beginning of '`test-jump61`' and the sample files in the skeleton `tests` directory for information on how to write your own tests. If you'd prefer to substitute your own test driver instead, go ahead; just make sure you submit all necessary files and that you update your makefile appropriately.

Your AI must reliably find a win from any position in which a forced win is four or fewer moves away, for any board no larger than six squares on a side. Likewise, it should be able to put off defeat for at least four moves (again on boards up to six squares on a side) if there is not a forced win for the other side in that many moves. Simply moving at random won't satisfy these requirements.

Your program will have limited time to move. Don't expect to get more than roughly 15 seconds per move on the instructional servers.

As always, your program must always be in control of its own termination. Terminating by means of an unhandled exception is never acceptable in a finished program.

In order to make it possible for the autograder to interact properly with your program, you must make sure that required outputs that you produce get output immediately. Modern systems tend to *buffer* input or output—collecting it until "economies of scale" make it efficient to deliver it to its destination. Furthermore, buffering is not consistent across file types: input from or output to a file may behave differently from input from and output to a terminal or Unix pipe. Buffering is problematic for an interactive program (there's no point in saving up all the prompts for input and only print them a thousand at a time.) To make sure this doesn't happen, make sure to use the '`flush()`' method on output files after writing a prompt or one of the outputs specified for this assignment. In fact, with the Java library there is no guarantee that the last outputs written to a file object actually get to their intended destination unless you explicitly *close* the file object, using the `.close()` method. So it's a good habit to close all output files before exiting, whether normally or as a result of an error.

You may add commands and other features to your program, as long as it otherwise meets this specification. If you do, be sure to update the output of the '`help`' command or of usage messages accordingly.

For extra credit (and please attempt this only *after* you have met the required conditions,) you can provide a graphical interface. We've set up the skeleton in such a way that you can use the simple strategy of having your GUI communicate with the rest of the program by writing the same commands you can enter by hand and interpreting the output from program to get its moves. Your GUI must operate *only* when the program is executed with

```
    java jump61.Main --display
```

(or '`java -ea`') so that the GUI is *not* initialized when the `--display` option is missing.

6 Advice

As usual, start immediately. Read the skeleton and try to understand its intent, even if you don't use it. We have deliberately included uses of various Java features in the skeleton in part to get you to explore them—to read the on-line documentation and think about how they might be useful.

We have provided a program `staff-jump61` with a solution we've developed. You can use it for ideas and sanity checks but it is *NOT* part of the specification! There might be errors in it for all we know (why, we might even have added a few just to mislead you.) Feel free to use it with `test-jump61` to write tests of your program running against another.

7 Corrections

1. Do not print "*P* `moves` *R C*`.`" for manual moves—only for moves by an automated player. (Originally, these were allowed as an option, but it makes testing unnecessarily difficult.)

2. Neutral sqaures start with no spots, not one spot.