CSC301 – Data Structures 2 – Final Project Deliverable #2

Project Description

The final project consists of two separate deliverables. The first is a puzzle application (a game) while the second is a solver for the puzzle. The puzzle application is worth 30 points and the solver is worth the other 70 points. Each deliverable has its own deadline. The puzzle itself is described in its own section below.

Individual Work

Remember that as explained in the syllabus, collaboration on the final project is restricted. You may discuss the problem with other students and share ideas but those ideas cannot include any code. You are not allowed to share code with other students or to suggest code to other students.

IMPORTANT: Differences from regular homework

- **No Resubmit Folder**
  There will be no late submission / resubmission folder on any of the project deliverables.

- **Late Policy**
  Similar to regular homework assignments, the submission folders for project deliverables will remain open for a limited time after the original due date. Like homework, the penalty for late submission is 2% per hour the submission is late (0.6 points per hour late on the first deliverable and 1.4 points per hour late on the second deliverable). The submission folder will remain open for an additional 48 hours after the due date to accept late submissions. However, this means that a submission that is a full day late will lose 50% of the points and no points are possible once 2 days have passed. Please plan your time accordingly so that you can submit on time.

- **Allowed changes to classes**
  You may add additional methods to the classes you were given and you may have them implement Comparable if you wish. You may also add other classes (that you write) in their own separate files. Just make sure to include all additional files when you make your submission so that I can run your code.

- **Allowed imports**
  You may import and use any classes from java.util (which includes Java's version of symbol tables) as well as from edu.princeton.cs.algs4 (which includes all of the book's code).

The Game/Puzzle

The game/puzzle consists of a 5 by 5 grid of cells. Each cell could be empty or could contain an 'X'. The object of the game/puzzle is to arrange the X's around the outer edges of the grid (the 16 cells that make up the bottom and top row and the leftmost and rightmost columns). You can shift the cells

around by shifting an entire row left or right or by shifting an entire column up or down. Cells that shift off the end wrap around to the other end. Below are the full details about the 20 moves available to a player:

- a-e: Shift an entire row to the left ('a' shifts the topmost row and 'e' the bottommost row).
- A-E: Shift an entire row to the right ('A' shifts the topmost row and 'E' the bottommost row).
- v-z: Shift an entire column down ('v' shifts the leftmost column and 'z' the rightmost column).
- V-Z: Shift an entire column up ('V' shifts the leftmost column and 'Z' the rightmost column).

The player continues to make moves until either they win by arranging the X's around the edges of the grid or they lose by repeating any of the previously reached board positions. This is probably easier to understand via a demonstration. A demonstration of how the game is played will be given in class, so please make sure to take a look at the lecture recording if you did not attend the class meeting where the demonstration was given.

Deliverable #2 (70 points)

The second deliverable has two related components

- **(40 points)** You must implement the Solver class. In doing so, you are allowed to define other classes to help you (as well as use "built-in" Java classes or the book's classes). For example, the Board class from part 1 of the project can be very useful, if you make a few additions/changes. The point of the solver class is the solve method which takes a board/puzzle configuration represented as a 2D array of booleans and returns a char array containing a minimal sequence of moves that will lead to the solved board (all the cells around the edges being filled). The board configuration is passed in as a 5-by-5 boolean array of Booleans with exactly 16 true cells (filled) and 9 false cells (empty). (This is the same interface as the Board constructor that takes a boolean[][] argument from the first deliverable.) The solve method then returns an array of characters representing a minimal sequence of moves that solves the puzzle. In other words, if the characters from the returned array are used in order as input to the move method on the Board object representing the initial configuration, the resulting board configuration represents the solved board. Furthermore, the solution must be minimal in the sense that there are no solutions that use fewer moves (although there could be other solutions that use the same number of moves). If the input to the solve method is a board configuration that is already solved, then solution requires no moves and an array of size 0 must be returned. The class also has a no argument constructor that will be called to construct/initialize the solver object. It's up to you what this constructor does. (There are perfectly valid solutions to the project where the constructor does absolutely nothing, but there are also solutions where some preprocessing work done in the constructor allows the solve method to be much faster.) Your grade for this portion is based on how your code performs on the two Junit test files, P2Test.java and P2TimingTest.java as described in the Grading section of this writeup.

- **(30 points)** You must answer the questions in the file "DesignDocumentation2.docx". Make sure to submit your edited DesignDocumentation2.docx file along with your Java source files in the submission folder on D2L. Your score on the questions depends on how well you explain

what your code does as well as the actual decisions you made in your solution. Note that a longer explanation is NOT better. You should be able to answer the questions with just a few sentences. Also, if a question asks about the runtime of a given feature or piece of code, don't just say "linear". You must say "linear with respect to _____" and fill in the blank. When we've discussed runtimes of data structures in class, it was always with respect to the size of the data structure (the number of elements in the data structure). This project is about a puzzle/game. It's not at all obvious to the reader what the "size" here is.

- You must include a screenshot of what happens when you run P2Test.java and P2TimingTest.java. Failure to include the screenshots will result in a 2-point deduction per missing screenshot.

Sample Use Case:

To use your code, someone would

1. create a Solver object (using the constructor)
2. create a couple of 5-by-5 array Booleans with 16 trues and 9 falses that represent the filled and unfilled cells in the puzzles.
3. call the solve method with each array given above as input
4. print out the moves in the order they appear in the char[] returned by the solve method.

A small toy example is provided in the main method of the Solver class that was given to you so you can see and how the Solver class will be used.

Grading:

Your grade for deliverable #2 will be the sum of the two components mentioned above and is out of a possible 70 points.

- The name of each test method in P2Test.java describes how much that particular test is worth. You will receive points for each test in P2Test.java that passes. The total points available from P2Test.java is 36 points.

- There is only 1 test in P2TimingTest.java. It is worth 4 points. The score you receive will be based on how fast your code runs on my computer. For reference, when I use my first solution where I was trying to be efficient the Timing test finishes in 13 seconds. I then coded up another solution where I tried to duplicate exactly the book's BFS algorithm, and that one took 49 seconds. Using these solutions as a baseline, I will award a minimum of 2 out of 4 points for any solution that takes a total of 51 seconds or less and all 4 points for any solution that takes a total of 14 seconds or less on my computer. Of course, the running time on your own computer will likely be different, but probably not by too much.

- There are 6 written questions worth 5 points each for a maximum possible score of 30 points. Note that your score on the written questions will be based both on how well you explain what

you did and on how well you solved the problem. Inefficient solutions will lose points. Also, you can't get credit for the last question (about performing just one BFS regardless of how many times solve is called), unless your code actually works that way.

Tips when my tests fail:

The name of each test includes an indication of how many moves the solutions will require. The body of each test includes the initialization of a String[] with five Strings, each of length five. This String[] will be converted to a boolean[][] that is used as the argument to your solve method. The stars represent filled cells (true in the boolean[][]) and spaces represent empty cells (false in the boolean[][]). For example, if I want to call your solve method with the solved configuration, I would initialize stringArray as follows:

```
stringArray[0] = "*****";
stringArray[1] = "*   *";
stringArray[2] = "*   *";
stringArray[3] = "*   *";
stringArray[4] = "*****";
```

The String[] is used to fill in the boolean[][] b and then the array b is fed into your solve method as follows:

```
toBooleanArray(b, stringArray);
answerArray = solver.solve(b);
```

The contents of the answerArray (a char[]) are then used to verify the moves really do solve the board. So if a test fails, you can look at how the stringArray was initialized in order to find out what the starting board configuration was.