Project 2 - Fifteen Puzzle

Due: Wednesday, October 4, 2017 at 11:59 pm

The Fifteen Puzzle is one variation of the N-Puzzle. The Fifteen Puzzle was made famous by Sam Lloyd who in 1878 offered $1000 to anyone who could solve an unsolvable variation of the puzzle. The puzzle consists of a 4x4 grid with the numbers from 1 to 15 on the grid leaving one grid position empty. The numbers are mixed up the puzzle is solved when the number are put into the following order:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

To solve the puzzle, a number that is next to the empty position is moved into the empty position. By "next to", the number can be above, below, to the left or to the right of the empty position. The empty position will now occupy the grid position were the number had been.

Your program is to implement the Fifteen Puzzle using the Java Swing Library GUI elements. One method to do this is to create a 4x4 grid of 16 buttons. Use the text of the buttons to specify which number (or the blank) is stored at that grid position. When the user clicks on a button that is next to the "blank" button, the text on the clicked button and the blank button is exchanged. Clicking on any other button does nothing. It is NOT expected that the program has the value of the button clicked "slide" into position of the blank button (as some apps do).

Once the puzzle is solved, display a dialog box congratulating the user and telling the user how many moves it took and the "complexity" of the original puzzle. The number of moves "so far" and the complexity rating of the puzzle are to be displayed at all times on the GUI.

Your program will need a menu that will perform the following operations.

- Quit the program

  This menu button is in addition the quitting your program by clicking on the X on the right side of the title bar.

- Display an About box

  The About box can be a simple dialog box that gives the name of the program's author, when the program was written, why ("The 2nd programming assignment for CS 342") and if the program attempted any extra credit (this last one will help the TA's when grading).

- Provide help on how to use your program.

  This can be a dialog box that describes the basics of your program. This should include a description of all menu operations.

- Mix the numbers in the grid.

  This should rearrange the numbers in the grid into some random but solvable order (see discussion on solvable below).

- Undo previous move.

  This should put the puzzle back into the previous positioning of the values. To do this, the program will need to maintain a stack of moves. As each move is made, information about that move is pushed onto the stack. Using the undo menu item will remove/pop the last move from the stack and restore the display to match current value on top of the stack.

- Undo all moves (with animation).

  Undo all of the moves done by the user, restoring the puzzle to its state after its last randomization. This should be done by repeatedly "Undoing the previous move" until the stack of moves is empty. Your program must pause for some short period of time after each undo to provide a sense of animation. Not pausing at all (or pausing for too short of a period of time) will seem to reset puzzle immediately without giving any sense of animation.

- Auto solve (with animation).

  This is to show the moves that will solve the puzzle with the fewest steps (via the use of a breadth-first-search, see discussion below). Once the shortest sequence of moves are known that will solve the puzzle from its current arrangement, step through the sequence of moves until the solved puzzle is displayed. Your program must pause for some short period of time after each move to provide a sense of animation.

Creating a Random and Solvable Puzzle

A discussion of what makes an ordering of the numbers in the puzzle solvable or not can be found at http://mathworld.wolfram.com/15Puzzle.html. The basic idea is whether there are an odd or even number of permutation inversions compared to the solved ordering of the numbers. Any arrangement (with the blank piece in the bottom row) that has an even number of permutation inversions is solvable, while any arrangement that has an odd number of permutation inversions is unsolvable.

The easiest way to determine if an arrangement is solvable or not is to count the inversions made for each position. First think of the 4x4 grid as a one dimensional array were the position in the array is (row-1)*4 + col. Thus the positions of the grid as:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

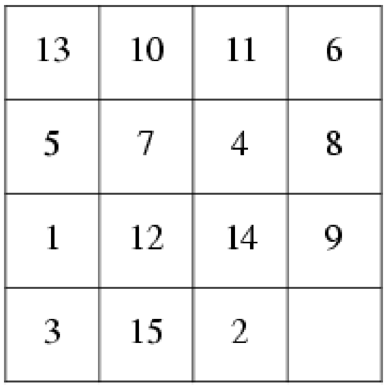
correspond to the array:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

For each value in the array, count the number of values in a higher position with a smaller value. This will give the number of permutation inversions. If this number is even, the puzzle is solvable. If this number is odd, the puzzle is not solvable. The above list of 16 numbers has zero inversions. However, if we were to flip the 1 and the 2, the list would have one inversion: the value 2 at position one has one smaller value at a higher position (the value of 1 at position two).

2	1	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The following discussion is taken from http://mathworld.wolfram.com/15Puzzle.html. Consider the following puzzle:



13	10	11	6
5	7	4	8
1	12	14	9
3	15	2	

The number 13 is at position 1. Since there are 12 values less than 13 at position higher than 1, the inversion count for the 13 is 12.

The number 7 is at position 6. Since there are 4 values less than 7 at positions higher than 6 (the 4 at position 7, the 1 at position 9, the 3 at position 13 and the 2 at position 15), the inversion count for the 7 is 4.

For the entire puzzle, the inversion counts are 12, 9, 9, 5, 4, 4, 3, 3, 0, 3, 3, 2, 1, 1, and 0, giving the total inversion count of 59. Since this number is odd, the above arrangement of the puzzle cannot be solved. Note that this example has the blank at position 16 and the blank is considered to contain the number 16. Since no number is larger than 16, it is excluded from the inversion count.

So when randomizing/shuffling the positions in the puzzle, determine the inversion count. If the inversion count is even, go ahead and use this randomization for the puzzle. If the inversion count is odd, you cannot use the current randomization. Note that half of the randomizations should result in an even inversion count, while half of the randomizations should result in an add inversion count.

Note that the above discussion randomizes the positions 1 through 15 on the grid, leaving the blank in the 16th position (its “original” location). This is important!

Complexity of the Puzzle

We will use the inversion count as the "complexity" of the program. The maximum number of inversions count could be 105; however, that creates an unsolvable puzzle. So the maximum "complexity" of a puzzle is 104. Note that the inversion count may not be the best/only way to determine the complexity of a puzzle (minimum number of moves need to solve might be better), but this gives us a knowable range and potentially useful debugging information (which helps us as developers if not as players).

Extra Credit

You may earn 15 pts additional credit by having an image divided into 16 parts and displaying 15 of those parts instead of using the numbers from 1 to 15. Once the problem is solved (you will need to automatically detect this), display the 16th part of the image. It is your responsibility to get and prepare the image(s) for your program. Note that doing this may want you to use JLabels rather than buttons. In that case, you will need to make the JLabels clickable as shown during an example during lecture.

You can earn an extra 5 points of credit if you allow the user to select their own image. This image would need to be divided into 16 parts before it can be used. This code will need to become part of your program. The Java bookClasses library (used by the CS 111 class in Fall 2015 and earlier from the Media Computation book in Java by Guzdial and Ericson) should make this code rather easy. Look for the Cropping Code in the example from Lect1029a.java on the Code Examples page. The code for filename manipulation from Lect1029c.java might also be helpful (depending on your approach).

To help the TA's grade your project (and not forget to give you credit your earned), include in the About dialog box an obvious statement specifying if you attempted the first or both of the extra credit parts.

Finding the Shortest Solution

To find the shortest solution, you will need to represent the puzzle as a graph and perform a breadth-first-search that starts on the graph node representing the current arrangement of the puzzle pieces and ending on the graph node representing the "in order" arrangement of the puzzle pieces. However, the number of nodes in such a graph would be 10,461,394,944,000 (assuming the information/calculation I have is correct). So the typical approach of first creating all of the nodes and edges for the graph before running a breadth-first-search is not really a viable solution. Instead we will need to create the graph as we are running the breadth-first-search algorithm.

To do this, we first need some way to represent the current arrangement of puzzle pieces. This was also needed for the Undo stack, so you might be able to develop an object that might work for both the Undo and Solve requirements.

We will also need to keep track of a sequence of puzzle piece moves (again this may be similar to the Undo stack).

The breadth-first-search algorithm will require the use of a queue and a set. The use of the Java Collection Classes can be used here; however, it is good to note that this program may need to perform additional operations not found in the Java Collection Classes. This may depend on the approach used to implement the algorithm.

The general breadth-first-search algorithm for this is as follows:

```
Create an empty queue and an empty set.
Let P represent the present  arrangement of puzzle pieces.
Create the tuple (P, null)
Add (P, null) to the end of the queue (enqueue)
Add P to the set (mark it as visited)

While the queue is not empty
    Current = front item in the queue; remove this item from the queue (dequeue)
    If arrangement in Current is the solved puzzle
        Stop – found solution
    For each arrangement R that can be formed by moving one piece from
         arrangement in Current
        If R is not in the set
            Create the tuple (R, M) where M is the movement list from Current
                with the addition of the move that was used to form
                arrangement R
            Add (R, M) to the end of the queue (enqueue)
            Add R to the set

If queue is empty
    No solution exists (the randomization created an unsolvable
```

Once the solution is found, we will need to use movement list in Current step the puzzle from the present arrangement P to the solution arrangement.

Note that most times this will run relatively quickly. However, some solutions are can be up near 80 moves, which will take some time to compute.

Multiple Source Code Files

You will need to break down your program into multiple source code files. Each object/class you create should be in its own file (with very few exceptions).

Programming Style

Your program must be written in good programming style. You should know what this means without having to explicitly state that here.

Submission of the Program

The intent is to use GitHub to submit your program. We will provide explanations on how to do this. (If the GitHub idea takes too long to get working, we may have you submit your program via the assignment link for Project 2 in Blackboard. Hopefully not.)