DUE: 9 June 2016

Project #3—Nine Tails Problem

The nine tails problem is as follows. Nine coins are placed in a three-by-three matrix with some face up and some face down. A legal move is to take a coin that is face up and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent). Your task is to find the minimum number of moves that lead to all coins being face down. For example, start with the nine coins as shown in Figure 28.17a. After you flip the second coin in the last row, the nine coins are now as shown in Figure 28.17b. After you flip the second coin in the first row, the nine coins are all face down, as shown in Figure 28.17c.

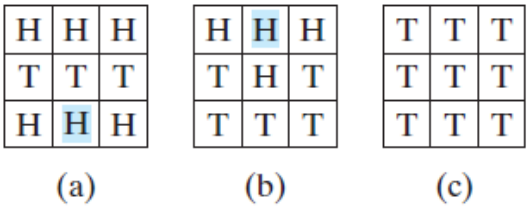


FIGURE 28.17 The problem is solved when all coins are face down.

Write a program that prompts the user to enter an initial state of the nine coins and displays the solution, as shown in the following sample run.

Sample Run:

```
Enter the initial nine coins Hs and Ts: HHHTTTHHH
The steps to flip the coins are
HHH
TTT
HHH

HHH
THT
TTT

TTT
TTT
TTT
```

Each state of the nine coins represents a node in the graph. For example, the three states in Figure 28.17 correspond to three nodes in the graph. For convenience, we use a 3 *

3 matrix to represent all nodes and use 0 for heads and 1 for tails. Since there are nine cells and each cell is either 0 or 1, there are a total of 29 (512) nodes, labeled 0, 1, . . . , and 511, as shown in Figure 28.18.

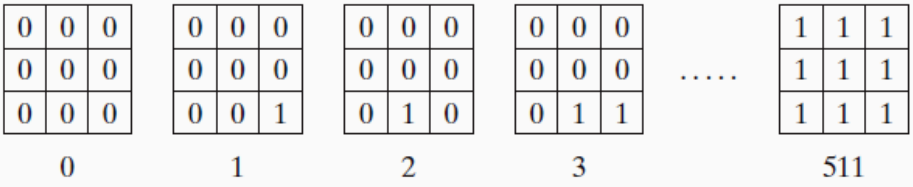


FIGURE 28.18 There are total of 512 nodes labeled in this order: 0, 1, 2, . . . , 511.

We assign an edge from node v to u if there is a legal move from u to v. Figure 28.19 shows a partial graph. Note there is an edge from 511 to 47, since you can flip a cell in node 47 to become node 511.

The last node in Figure 28.18 represents the state of nine face-down coins. For convenience, we call this last node the target node. Thus, the target node is labeled 511. Suppose the initial state of the nine tails problem corresponds to the node s. The problem is reduced to finding a shortest path from node s to the target node, which is equivalent to finding a shortest path from node s to the target node in a BFS tree rooted at the target node.

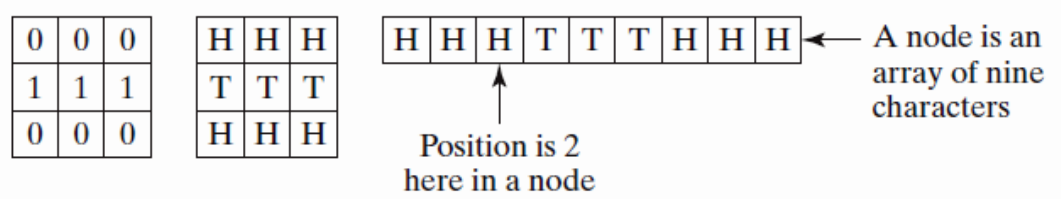
Now the task is to build a graph that consists of 512 nodes labeled 0, 1, 2, . . . , 511, and edges among the nodes. Once the graph is created, obtain a BFS tree rooted at node 511. From the BFS tree, you can find a shortest path from the root to any vertex. We will create a class named NineTailModel, which contains the method to get a shortest path from the target node to any other node. The class UML diagram is shown in Figure 28.20.

Visually, a node is represented in a 3 * 3 matrix with the letters H and T. In our program, we use a single-dimensional array of nine characters to represent a node. For example, the node for vertex 1 in Figure 28.18 is represented as {'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'T'} in the array.

The getEdges() method returns a list of Edge objects. The getNode(index) method returns the node for the specified index. For example, getNode(0) returns the node that contains nine Hs. getNode(511) returns the node that contains nine Ts. The getIndex(node) method returns the index of the node.

Note that the data field tree is defined as protected so that it can be accessed from the WeightedNineTail subclass in the next lesson.

The getFlippedNode(char[] node, int position) method flips the node at the specified position and its adjacent positions. This method returns the index of the new node. The position is a value from 0 to 8, which points to a coin in the node, as shown in the following figure.



For example, for node 56 in Figure 28.19, flip it at position 0, and you will get node 51. If you flip node 56 at position 1, you will get node 47. The flipACell(char[] node, int row, int column) method flips a node at the specified row and column. For example, if you flip node 56 at row 0 and column 0, the new node is 408. If you flip node 56 at row 2 and column 0, the new node is 30.

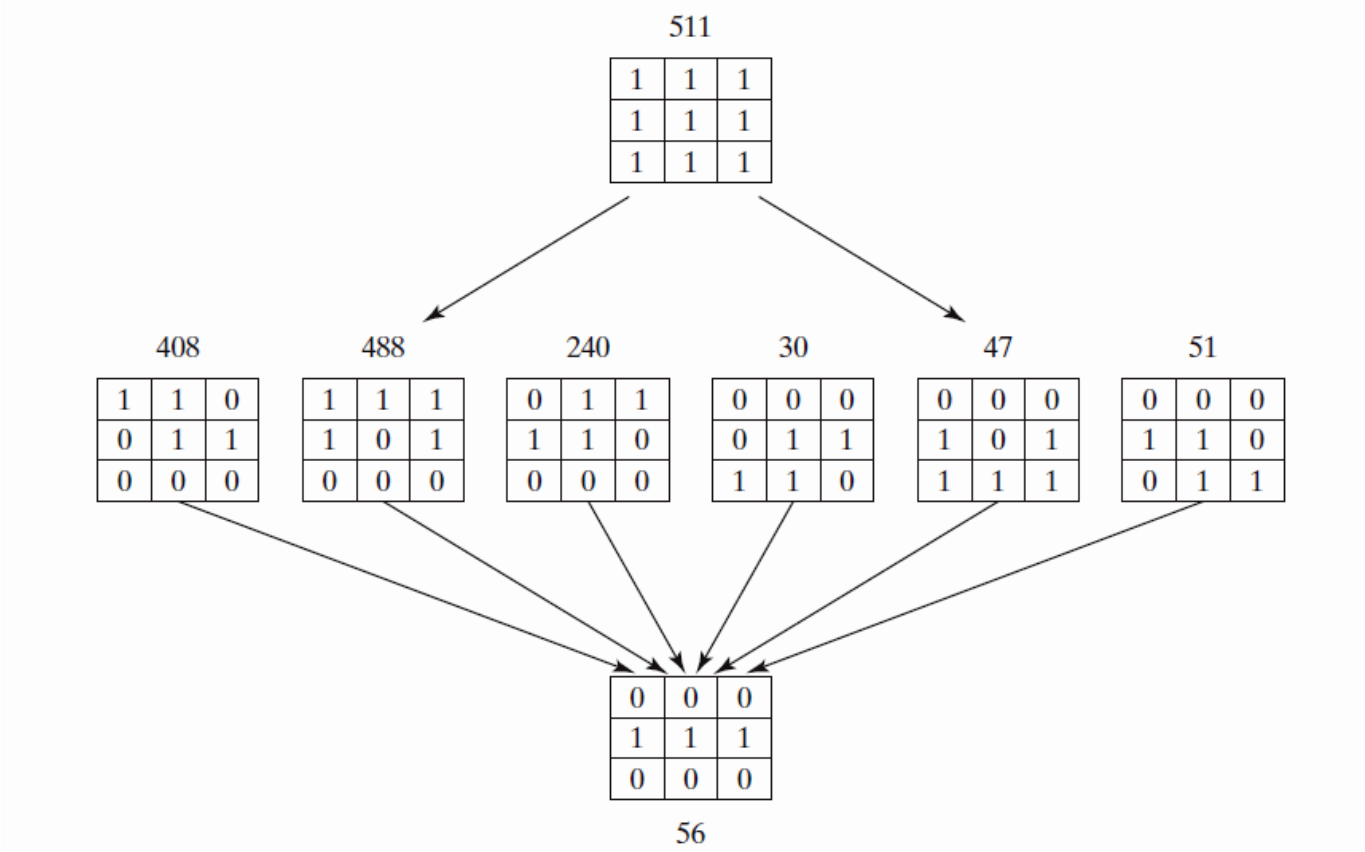


FIGURE 28.19 If node u becomes node v after cells are flipped, assign an edge from v to u.

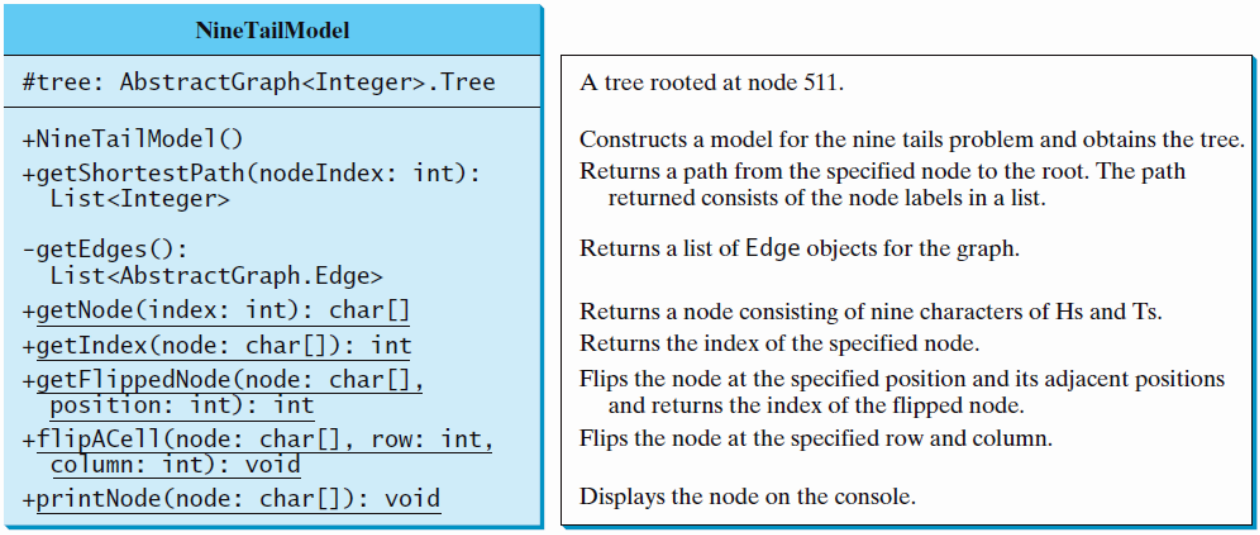


FIGURE 28.20 The `NineTailModel` class models the nine tails problem using a graph.

EXTRA CREDIT (Optional) There is numerous ways to obtain extra credit for this project:

1. Modify the application so that user sets an initial state of the nine coins visually via a GUI (see Figure 28.22a) and click the Solve button to display the solution, as shown in Figure 28.22b. Initially, the user can click the mouse button to flip a coin. Set a red color on the flipped cells.

FIGURE 28.22 The program solves the nine tails problem.

2. In the nine tails problem, when you flip a coin, the horizontal and vertical neighboring cells are also flipped. Rewrite the program, assuming that all neighboring cells including the diagonal neighbors are also flipped.
3. In the nine tails problem, when you flip a coin, the horizontal and vertical neighboring cells are also flipped. Rewrite the program, assuming that all neighboring cells including the diagonal neighbors are also flipped.
4. The NineTail.java, presents a solution for the nine tails problem. Revise this program for the 4 * 4 16 tails problem. Note that it is possible that a solution may not exist for a starting pattern. If so, report that no solution exists.
5. The nine tails problem in the text uses a 3 * 3 matrix. Assume that you have 16 coins placed in a 4 * 4 matrix. Write a program to find out the number of the starting patterns that don't have a solution.

6. Rewrite the program to enable the user to set an initial pattern of the 4 * 4 16 tails problem (see Figure 28.23a). The user can click the Solve button to display the solution, as shown in Figure 28.23b. Initially, the user can click the mouse button to flip a coin. If a solution does not exist, display a message to report it.

FIGURE 28.23 The problem solves the 16 tails problem.

Program Requirement. Your programs must contain/do the following:

1. The project should start with a Splash Screen that closes itself after so many seconds and it should contain a meaningful About form activated from the Help menu.
2. There is no pre-designed GUI for this project--you have to design one with appropriate GUI controls (JButtons , JMenuItems, display controls, etc .). Use good design principles emphasizing functionality.
3. Ability to enter the state of the matrix for the coins.
4. Appropriate classes and data structures.
5. Javadocs, description of the program, and comments everywhere.
6. Menus that synchronize with corresponding buttons and with at least the following menu choices:
   6.1. File with Clear, Open, Print, Save, and Quit menu items.
   6.2. Help with About menu item for an About form.
7. Ability to create and save data files with random values of specified number.
8. Print the form in its entirety. Use the provided PrintUtilities class.
9. The forms should not be resizable or maximizable (although one should be able to minimize the main form).
10. A free image of graph for icon of the form.
11. ToolTip text for the buttons and menus at least, explaining their function.

You might find these partial comments worthwhile—each counts as a -1 point.

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'Comments by the prof:
'Great effort.  Here are suggestions for improvement:
'1. Use JavaDoc comments throughout the program, not just at the top heading.
'2. A small image of a matrix multiplication would enhance the look of the form.
'3. Did not implement printing of form--it is required.
'4. Make the Display button default--pressing the Enter should fire it.
'5. Make the display JTextArea read-only.
'6. Validate input--do not allow illegal input for the arrays.
'7. It is good style to tab code inside a method.
'8. Validate the correctness of the arrays' dimension and data before performing
operations.
'9. Declare all constants as final.
'10. Read values from external files using JFileChooser and FileReader class.
'11. Change the tab sequence to indicate correct order of entry.
'12. Name the classes and project appropriately.
'13. Provide ability to generate a file with randomly generated values for the
cities.
'14. Follow the Java naming convention for naming variables, methods and classes.
'15. Read the cities data from an external file or database--include package in path.
'16. Include a separate Validation class for the project.
'17. Enable menu choices that synchronize with buttons' functionality.
'18. Provide a generic types in your class designs.
'19. Missing required and meaningful menus.
'20. Disable maximization or resizing of all forms (About in your case).
'21. Add a meaningful About form which describes the project.
'22. Add a Splash screen that starts the project, displayed in center.
'23. Add icons to all forms, including About.
'24. Comapare and print the tours for each algorithm.
'25. Provide ToolTip texts for at least each button to explain their functionality.
'26. Provide time comparison between the two different algorithms.
'27. Give title to all forms.
'28. Save the results using a FileWriter class.
'29. Program crashes on empty.
'31. Incomplete.
'33. Late.
'
'The ones that apply to your project are:
'6, 16, 19, 26.
'
'26+2=28/30
'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
```