Lab Assignment 2
Freak on a Leash

Assigned: Tuesday February 22, 2011
Due: Saturday March 5, 2011 11:00 PM
Assignment Type: Team

Lab Objectives

- Continue to develop skills in software construction.
- Continue to develop a test-first mindset.
- Continue to develop a daily build mindset.
- Create effective abstractions to model a concrete problem.
- Solve a larger problem in terms of solutions to smaller problems.
- Select and use appropriate collections to solve problems.
- Practice working as a member of a team.

Lab References

- Chapter 6: Queue
- Chapter 7: Stack
- Lecture notes

Problem Description

Maze solving is a great example of a problem that has many practical applications and is one that lends itself to different strategies supported by different collections. Practical applications include variations of search problems, autonomous navigation problems, and artificial intelligence. (By understanding how to construct maze solutions you can also avoid being drug out of a rural corn maze by your pet dog. Doh!)

You must write a program in Java that reads in maze specifications and uses two different strategies to compute solutions to the mazes if solutions exist. The strategies that you must use (independently) are: (1) depth-first search (dfs) with a stack and (2) breadth-first search (bfs) with a queue. For each of your strategies where it is possible, you must not only solve the maze but also store the solution path and be able to provide it on demand to a client. **You are prohibited from using existing source code or maze solvers as part of your solution**.

You are given almost complete design freedom in this assignment, but you will be evaluated on the quality of your design. Your solution must not consist of one class. The components of your solution should be designed with an eye toward good design and reusability. For example, your components should be able to be used as-is or with slight modification in other software that explore mazes. Some "hard-wiring" is okay, but it should be minimal.

The driver class for your solution must be named MazeSolver. The main method in MazeSolver must take a directory name from the command line as input. Each file in the supplied directory will contain exactly one maze specification (see below). Your program must apply both strategies (dfs, bfs) independently and in sequence to process the maze in each of the files in the directory. So, the sample command line invocation below would cause your program to process all the mazes specified in the files contained in the directory maze_test_files.

```
> java MazeSolver maze_test_files
```

If the command line is left blank the default directory name must be "maze_test_files." If the command line argument specifies a directory that is empty or does not exist, your software should display an appropriate error message but not crash or throw an exception. You must not assume that the supplied directory is a subdirectory of the directory in which your software is running. It must not crash no matter what the state of the command-line input.

Each maze specification file will start with one line containing two integer values that give the dimensions of the maze. The maze description will begin on the second line of the file and will use the following notation:

S (upper case 'S'): marks the start cell of the maze (entrance)
F (upper case 'F'): marks the finish cell of the maze (exit)
x (lower case 'x'): marks a wall/obstacle
- (dash): marks an open/passable cell

For example:

```
5 5
S----
-x-xx
x--x-
--xx-
x---F
```

The remaining lines of the file must be ignored. This will allow comments or notes about the maze to be recorded in the same file as its specification. Corrupt maze specification files must not cause your software to crash.

To process a given maze specification file, your program must read in the specification and create an internal representation of the maze. Then your program must explore the maze, searching for a solution, using both of the required strategies independently and in sequence. A solution to a maze is a sequence of cell-to-cell moves that begins at the start cell, ends at the finish cell, and does not cross the same cell twice. For example, the solution to the sample specification above would be:

(0,0) → (0,1) → (0,2) → (1,2) → (2,2) → (2,1) → (3,1) → (4,1) → (4,2) → (4,3) → (4,4)

Note that the (0,0) position is located at the top-left corner of the maze. You must follow this numbering convention for the coordinates of the maze cells when reporting solution paths.

Some mazes have no solution, some have exactly one solution, and some have multiple solutions. If a maze has no solution, both strategies must detect and report this. If a maze has multiple solutions, both strategies are free to detect and report any one of them.

Your main goal for both strategies is to find *any* possible solution to the maze. However, a small number of points will be allocated to judge how well you address efficiency. You can address efficiency in different ways: for example, finding the shortest path from start to finish, finding a solution by exploring the fewest number of cells, or choosing the next cell to explore based on your current position's relative relationship to the exit. Each strategy can address efficiency in a different way.

When exploring a maze, movement is allowed in only four directions: up, down, left, and right. No diagonal moves are allowed. As your program explores the maze, it must mark all cells that it explores with a period ('.'). It must also mark all cells that are part of the discovered solution with a lower case o ('o'). The start and finish cells must remain marked with 'S' and 'F', however.

For each maze specification, your program must do the following for output.

1. Display the original maze.
2. For each of the two required strategies:
    a. Display the explored maze (with . and o marks in the appropriate cells).
    b. List a solution in the format (x1,y1) → (x2,y2) as shown above or state that there is no solution.
    c. List the number of steps (cells) in the solution that was discovered.

d. List the total number of explored cells, both as a raw number and as a percentage of the total maze cells.

In addition to creating a solution to this problem in Java, you must also create a README file for this project. The README can be in any commonly readable format (e.g., Word, PDF, plain text). This file must contain a concise, well-written description of your project. You must discuss your design, including a description of all the classes involved and how you arrived at the design that you chose to implement. You must also include a discussion of alternative strategies to at least one sub-problem that you considered, and a justification of your choice. The README file must also contain complete descriptions of how to compile and run your project and the output from an example run.

Lab Turn-In

You must upload your submission to Blackboard no later than the date and time specified. Your submission must be a single zip file that contains the README file and all the Java source code files and input files needed to compile and run your solution. No late submissions will be accepted. Your submission will be graded according to a rubric that will be posted in the assignment directory this week. This rubric will be discussed in class.

The work that your team submits must be solely the product of your team. You must not include the work of others as part of the design and/or solution that you turn in.