Project 2: The Amazing Maze
(The Glory of Queues)

Total: 100 points

Posted 10/07/2016

Due 11/08/2016

Objective

This project provides experience implementing an array based queue and a linked list sequence. You will also review how to construct and use Java classes as well as obtain experience with software design and testing.

ABET Program Learning Outcomes:

- The ability to recognize the need for data structures and choose the appropriate data structure (b,c,i,j)
- The ability to evaluate the performance of an algorithm (b,c,i)
- The ability to implement and use stacks and queues (a,b,c,i,j)
- The ability to implement and use linked list and array based structures (a,b,c,i,j)

The Problem

The programming problem in this project is to find a path from the **entry** cell (upper left corner) to the **exit** cell (lower right corner) of a rectangular maze (labyrinth).

The maze is a grid of cells. Each cell is bordered by 4 walls. Some of these walls are passable to the neighboring cell. The border line of the whole maze that is, the perimeter of the rectangle is not passable. You may consider the external walls of the entry as passable one-way in and the external walls of the **exit** are passable one-way out, though these assumptions are irrelevant to the solution of the programming problem. The program

- obtains input to build the a maze; input supplied by an external file
- finds a path through a maze if such a path exists
- displays the solution (or one of the solutions), that is a path traversing the maze; the display shows all the locations of the cells along the path on the console; the cells listed from entry to exit
- determines the farthest cell on an attempted path in the case the traversal failed and displays the path leading to the max distance cell
- creates a graphical representation of the solution (optional for bonus points)

There are figures given in an attached file Figures.doc that illustrate the requirements specified in the Problem description.

Analysis and Design

1. The focus of this project is a class named **`LinkedMaze`** that represents a rectangular maze. The maze consist of a two-dimensional array of cell objects, instances of the **`Cell`** class. Each cell knows which ones of the four neighbors are accessible from it. A cell can be viewed at as a super-node with five links rather than one (or two). There is a link for each of the directions of north, east, south and west. For an impassable wall the link is null, for a passable wall the link is the neighbor cell in the given direction. The cells maintain yet another link called backLink which points to the predecessor cell on a potential path. In addition to the links a cell has two int type fields to store the indices of the location it takes in the two dimensional array and a boolean field **visited** which takes true when the cell has been visited during the path finder algorithm. You will have to implement the **`Queue`** class based on a circular array which will serve as the main tool for the algorithm searching for the path, and a class **`Applications`** to test your program and to run it for several maze applications.

2. Searching for a path

The entry location or starting cell is by definition the upper left corner of the grid, the exit or finish location is the lower right corner. To search for a path from start to finish we use the so called “breadth-first” algorithm. The logic of this algorithm is quite simple and is described by the following pseudo-code.

1. Mark the entry cell visited and enqueue
2. Repeat:
    - dequeue
    - if **exit** is a neighbor of the current (dequeued) cell, stop the algorithm and build the output for the successful search
    - otherwise check the four directional links of the current cell
    - for each non-null and non-visited link mark the link visited, add it to the queue and assign the added link’s backlink the current cell
3. Stop the algorithm if the queue is empty and build the output for the failed search

The pseudo-code above must be refined in order to deal with max distance cell. If **row** is the row index and **column** is the column index of a cell in the two dimensional array, the distance of that cell from the entry is measured by **row + column** (sometimes this distance is referred to as the Manhattan distance). A variable maxDistance must store largest distance value found this far and another variable maxCell referes to the cell where the maxDistance has been attained. Every time an east link or a south link is added to the queue, the distance of the link is to be compared to the current maxDistance. In case of a greater link distance, the maxDistance variable is updated and link is the new value for maxCell. Note that north and west links never increase the distance thus they are ignored.

A partial specifications for the classes are given as follows.

3. Cell

Fields

row, column – int type to store the location coordinates of the Cell in the grid (array)

nLink, eLink, sLink, wLink, backLink – Cell references; nLink is null if the north wall of the cell is not passable

visited – boolean; marks the cell with true if it has been enqueued; value never changed back to false later

Methods

Cell( ) – constructor, takes two int parameters for the array coordinates, initializes the coordinates. Note that at Cell instantiation all the links keep the default null value and visited stays false.

linked( ) – static utility method, returns boolean; takes two cells for parameter and determines if one is an accessible neighbor of the other, that is if one is a directional link of the other

toString( ) -- returns a string that shows row and column

sum( ) – convenience method returns row+column; helps to maximize distance

4. Queue

Fields

manyItems, front, rear (int types)

cellQueue (a circular array of cell objects)

Methods

Constructor (takes a parameter for array size and instantiates the array (NOT the array elements))

nextIndex

ensureCapacity

enqueue

dequeue

isEmpty

5. LinkedMaze

Fields

The class needs the following **fields**:

- pathfinder; a Queue reference
- entry, exit, maxCell; Cell references
- cells; a Cell[][] array
- length, width, maxDistance; int type variables to store the dimensions of the cells array and the maximum distance from the entry cell

Methods

The **behavio**r of LinkedMaze is suggested as follows:

- LinkedMaze ( ) -- constructor; takes the grid dimensions and a Queue reference for parameters to initialize those fields. Instantiates the **cells** array to the given dimensions. Populates the array with Cell objects. Assigns **entry** the **cells[0][0]** element and **exit** the lower right corner element of the array
- connectMaze( ) – void, takes two Scanner objects to read the files characterizing the north-south walls, and the other file having the east-west walls. Runs a nested pair of for loops to assign correct values to the directional links of each **cell**
- findPath( ) runs the search algorithm; see the pseudo code and additional hints above; returns boolean
- reversePath ( ) takes a cell for parameter and returns a cell. Given a cell current on a path, the backLinks create a **linked list** such that current is the head and entry is the tail. The method reverses the links and makes entry the head and current the tail. Note that the reverse must not be called if additional cell operations will be needed along the path elements, only when it comes to display the path.
- displayPath ( ) prints the coordinates of the cells along a path, starting at the entry cell (before this method, must call reversePath( ))

Implementation, Testing and Applications

- You must use the classes, class names, data structures and functionalities as described in the Design
- Additional methods and/or fields are allowed, but if you choose so, must give very thorough documentation to help the grader to identify the functionalities
- You must include the full specification of the LinkedMaze class. Specification of other classes and methods is optional
- Make sure that data structure classes follow the traditional usage of field and method names
- Using generic type(s0 is not required

MazeApplications

This class drives all the maze applications.

<u>The maze configuration</u>

- Solicit the input file names on the console is input directly in the code. The attached Figures.doc file shows maze examples and associated input files.
- The grid dimensions are contained in the first row of the eastWest file. The dimensions must read first, before the actual wall information reading is processed

- The eastWest file contains the east-west wall information listed row-by-row. Notice that only the east directions are listed, since a passable east wall is at the same time a passable west wall for the next cell to the right. For the same reason the northSouth file gives a listing on the south direction only.
- Instantiate Scanner objects to read the files and read the dimension first.
- Knowing the dimensions instantiate the Queue object pathfinder. Meditate a bit about a good choice for the length of the queue (not too long still making the ensureCapacity call unlikely)
- Having the dimensions and pathfinder obtained, instantiate a LinkedMaze object **maze**
- Call the connectMaze( ) method (you have the Scanners at hand for parameter)
- Call the findPath( ) method and store the return value in a boolean variable **flag**
- Call the displayPath( ) method; depending on the **flag** value display the path traversing the entire maze, or just a partial path running to the farthest cell found and print out the maximum distance

Documentation: As specified above

Evaluation (100 points)

1.	Documentation and Style…….……	20pts
2.	Correct implementation of the LinkedMaze class constructor…	10pts
3.	Correct implementation of the findPath( ) method.....	20pts
4.	Correct implementation of the other LinkedMaze methods…	10pts
5.	Correct implementation of the Queue class…….....	10pts
6.	Correct implementation of the Cell class……	10pts
7.	Correct implementation of the application class……	10pts
8.	Consistent specification of the LinkedMaze class……	10pts
Total	……	100pts

Submit:
Zip the project folder containing all source codes and the report. Upload at Blackboard