COMPUTER SCIENCES AND SOFTWARE ENGINEERING
AUBURN UNVERSITY

COMP 2710
Software Construction

Fall 2014

Lab 3 Shortest Path Maze Solver

Due: November 10, 2014

Points Possible: 100
Due: Written Portion: November 3rd, 2014 by 11:55 pm. Submission via Canvas (25 points)
Program: November 10th, 2014 by 11:55 pm via Canvas (75 points)

No late assignments will be accepted.
No collaboration between students. Students must NOT share any project code with each other. Collaborations in any form will be treated as a serious violation of the University's academic integrity code.

Goals:

- To develop an application that will find a shortest path in a maze from the starting node to the destination
- To develop a maze application that use graph representation
- To learn the use of pointers and dynamic memory allocations
- To learn about how to manipulate graph data structures
- To perform Object-Oriented Analysis, Design, and Testing

Software Process – 25 points:
Create a text, doc, or .pdf file named "<username>-3p" (for example, mine might read "lim-3p.txt") and provide each of the following. Please submit a text file, a .doc file or .pdf file (if you want to use diagrams or pictures, use .doc or .pdf). You are free to use tools like Visio to help you, but the final output needs to be .txt, .doc, or .pdf.

1. **Analysis:** Prepare use cases. Remember, these use cases describe how the user interacts with a maze system (what they do, what the systems does in response, etc.). Your use cases should have enough basic details such that someone unfamiliar with the system can have an understanding of what is happening in the maze system interface. They should not include internal technical details that the user is not (and should not) be aware of.

2. **Design**:

a. Create a Class Diagram (as in Lab 2). Be sure to include:

1) The name and purpose of the classes
2) The member variables and the functions of the class
3) Show the interactions between classes (for example, ownership or dependency)
4) Any relevant notes that don't fit into the previous categories can be added

b. Create the data flow diagrams. Show all the entities and processes that comprise the overall system and show how information flows from and to each process.

3. **Testing**: Develop lists of _specific_ test cases OR a driver will substitute for this phase:

1) For the system at large. In other words, describe inputs for "nominal" usage. You may need several scenarios. In addition, suggest scenarios for abnormal usage and show what your program should do (for example, entering a negative number for a menu might ask the user to try again).
2) For each object. (Later, these tests can be automated easily using a simple driver function in the object)

4. Implement the shortest path maze solver system

5. **Test Results**: ***After developing the shortest path maze solver system***, actually try all of your test cases (both system and unit testing). Show the results of your testing (a copy and paste from your program output is fine – don't stress too much about formatting as long as the results are clear). You should have test results for every test case you described. If your system doesn't behave as intended, you should note this. Note: Driver output will substitute for this phase.

Program Portion:

The graph abstraction is an important one since it is used in many different areas of computer sciences and software engineering. For instance, the Internet makes use of graphs representing the Internet router network configuration in order to determine the best route for forwarding packets from the source at one end of the Internet to the destination at the other end. Once a graph is constructed to represent the Internet configuration accurately, then the router executes an algorithm to determine the shortest path from each node to each destination and stores the information in its routing table.

In Lab 3, you will have fun constructing a similar graph to represent a **pointer-based maze**, which can be used by the user to implement a simple system for finding the shortest path in a maze automatically. You will analyze, design, implement and test the pointer-based shortest path maze solver system. Once the graph is constructed by the system, the system will automatically find the shortest path in the maze from any starting node to any destination node, by attempting each connected direction at each node using breadth-first search technique until it hits a dead-end or the destination is reached. ***Your program must follow the pointer to each node as the system traverse the graph.*** Since the maze configuration is unknown to the system, it will randomly select each direction (or in some pre-defined order) and learn and remember past routes and try to find the correct path to the destination node. When the user reaches the destination node, the program will print out the names of *all the nodes traversed by the user in the correct order,* starting from the start node to the destination node.

What is a graph?

A graph consists of multiple nodes and edges. For instance, a graph can be represented by the figure below, where the circles represent the nodes and the lines represent the edges. An edge connects two nodes. In this lab assignment, we will only consider special graphs, where each node can have up to four normal edges. An implementation detail to note here is that although the edge is represented by a single line, there are actually two links in both directions between the two nodes. For example, the line between A1 and B1 below is implemented by a link from A1 to B1 and another link from B1 to A1.

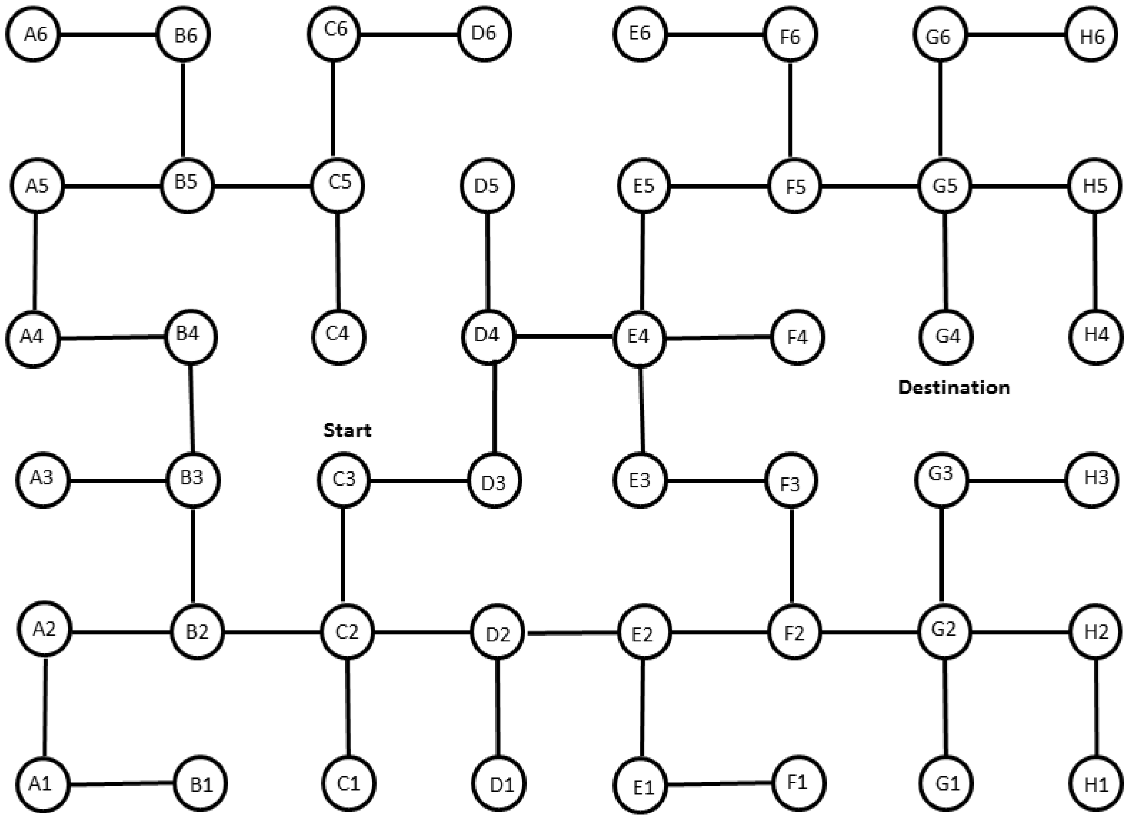

Figure 1. A Sample Maze

What is a Shortest Path Maze Solver System?

Consider a typical maze that consists of a set of rooms and passages connecting the rooms. The maze of rooms and passages can be represented by a graph, such as the graph shown above. The nodes can be thought of as rooms and an edge represents a passage that connects one room to another. Each room can be connected to at most four other rooms, as represented by the graph above.

In this lab assignment, you will implement the maze above using references to instances of a `Node` class, which you will define. Each node in the graph will correspond to an instance of the `Node` class. The edges correspond to the links that connect one node to another and can be represented in `Node` as a pointer variable that points to another `Node` class object. Since there are four possible normal links, `Node` must contain four links (pointers) to other `Node` objects. Your `Node` class must be defined as follows (additional features may also be defined):

```
class Node {
public:
         Node(string newname);
         Node();
         void setNodeName(string newname);
         string getNodeName();
         void attachNewNode(Node *newNode, int index);
         Node *getAttachedNode(int index);
         int getNumberUsed();
         int setNumberUsed(int newNumberUsed);
         void setPrevious(Node *previousNode);
         Node *getPrevious();
         int setMarked();

private:
         string name;
         Node *attachedNodes[4];
         int numberUsed;
         Node *previous;
         int marked;
}
```

Your graph will consist of `Node` objects that are linked together through pointers. The member variable `attachedNodes` list will contain pointers to other nodes, for e.g. `attachedNodes[0]` will contain a pointer to another node, `attachedNodes[1]` will contain a pointer to yet another node and so forth. Since not all the links are used, the member variable `numberUsed` denotes the number of links that are actually used in that particular node. The member variable `name` will contain the name of the node, e.g. "A1". You must implement the constructors `Node(char newname)`, `Node()` and all the other member functions: `setNodeName(char newname)`, `getNodeName()`, `attachNewNode(Node *newNode, int index)`, `getAttachedNode(int index)`, `getNumberUsed()`,`setNumberUsed(int newNumberUsed)`, etc. In the beginning, you must create all the nodes using a ***dynamic array*** of `Node` objects, where the size of the dynamic array is the number of nodes in the graph as provided in the configuration file.

The `Start` variable is a pointer that points to the node where the shortest path solver system will start, for e.g. it may point to node `C3`. The goal is to reach the destination which is the node that is pointed to by the `Destination` variable, for e.g. it may point to the node `G4`.

Graph Configuration

The configuration of the graph that you will use in the program will be read in from a text file called the configuration file. The first line of the file specifies the number of nodes in the graph. The next line specifies the name of the starting node and the following line specifies the name of the destination node. Starting from the fourth line, each line in the rest of the file indicates information on each node, i.e. the name of the node and the links that may exist from that node to another node. There may be more than one other node in which a node can be linked to (up to four other nodes). For example, the first eleven lines of the configuration file for the graph above is as follows:

```
48
C3
G4
A1  A2  B1
B1  A1
C1  C2
D1  D2
E1  E2  F1
F1  E1
G1  G2
H1  H2
…….
```

In the fourth line of the configuration file above, the name of the node is A1. It has two pointers: a pointer to node A2 and a pointer to node B1. The third and fourth pointers are set to NULL and the variable `numberUsed` is set to 2.

Your program will first read the graph configuration file and construct the graph data structures used for the shortest path maze solver. Your program must run correctly with any graph configuration file. Several test configuration files will be released to you close to the deadline for you to test the correct execution of your program. *A graph configuration file will have any number of nodes.*

Shortest Path Maze Solver System

The shortest path maze solver system will traverse the graph starting from the start node and automatically find a shortest path to the destination node. It will then prints out the names of all the nodes that it traverse to reach the destination node. When the shortest path solver program is at a current node, it will determine if it can traverse to another node by following the `attachedNode` pointer using the `getAttachedNode()` function, up to the `numberUsed` of attached nodes. While the program traverses the graph, it keeps track of the nodes that it visited through the `previous` pointer. Upon reaching the destination, it will print out the congratulation banner and the names of all the nodes that it traverses to reach the destination node.

Consider the maze in Figure 1. You can probably solve it immediately, but can you describe the algorithm you use to solve it? There are many algorithms for finding shortest paths in a maze automatically. The most commonly used algorithm is based on the *breadth-first search* algorithm. A breadth-first search (BFS) algorithm works the following way: BFS is a search method that aims to expand and examine all nodes of a graph by systematically searching through every solution. In other words, it tries to exhaustively search the entire graph until it finds the destination.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue).

The informal algorithm is as follows:

- If the destination name being sought is found in this node, quit the search and return a result.
- Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.

The pseudo code is as follows:

```
For each node n
   do unmark n
      set previous to NULL
create a queue Q
enqueue start node onto Q
mark start node
while Q is not empty:
    dequeue an item from Q into n
    for each link l incident on n in Graph:
        let m be the other end of l
        if m is not marked:
            mark m
            set m.previous to n
            enqueue m onto Q

```

An important issue to be aware of is that the maze may have *loops*, e.g. see Figure 1. In order to avoid going round the loop infinitely, you must mark the nodes that you have traversed and when you try new paths, you will not use that nodes that have been marked because you have already tried them.

User Interface

The user interface must repeatedly prompt the user for the name of the configuration file and check if the file can be opened correctly. If there is any error, e.g. invalid filename, then the program must display the appropriate error message and continue to prompt for the correct input. Your program must not exit or terminate when there is an incorrect input value. When the BFS algorithm is running, *it must print out all the nodes that it visited, in the order it which they are visited.* When the destination node is found, it must print the correct path from the start node to the destination. To do this, use the previous pointer and starting from the destination node, push each node into a ***stack***. When popping from the stack, the nodes will represent the shortest path from the start node to the destination. After this the user interface must prompt the user for the next file or accept a quit option.

The name of your program must be called <username>_3.cpp (for example, mine would read "lim_3.cpp"

Use comments to provide a heading at the top of your code containing your name, Auburn Userid, and filename. You will lose points if you do not use the specific program file name, or do not have a comment block on **EVERY** program you hand in.

Your program's output need not exactly match the style of the sample output (see the end of this file for one example of sample output).

Important Notes:
You must use an object-oriented programming strategy in order to design and implement this pointer-based shortest path maze solver system (in other words, you will need to write class definitions and use those classes, you can't just throw everything in main() or other independent functions). A well-done implementation will produce a number of robust classes, many of which may be useful for future programs in this course and beyond. Some of the classes in your previous lab project may also be re-used here, possibly with some modifications. Remember good design practices discussed in class:

a) A class should do one thing, and do it well
b) Classes should NOT be highly coupled
c) Classes should be highly cohesive

- You should follow standard commenting guidelines.

- You DO NOT need any graphical user interface for this simple, text-based application. If you want to implement a visualization of some sort, then that is extra credit.

Error-Checking:

You should provide enough error-checking that a moderately informed user will not crash your program. This should be discovered through your unit-testing. Your prompts should still inform the user of what is expected of them, even if you have error-checking in place.

Please submit your program through the Canvas system online. If for some disastrous reason Canvas goes down, instead e-mail your submission to TA – Steffi Mariya Gnanaprakasa P Arasu – at smg0033@tigermail.auburn.edu. Canvas going down is not an excuse for turning in your work late.

You should submit the two files in digital format. No hardcopy is required for the final submission:

<username>_3.cpp
<username>_3p.txt (script of sample normal execution and a script of the results of testing)

A sample execution is shown below, where the bold fonts indicate input by the user.

> *lim_3*

```
        ============================================================
        |          Welcome to the Shortest Path Maze Solver!       |
        ============================================================

Enter the name of the Maze configuration file: Maze1
The Start Node is: C3
The Destination Node is: G4
Finding the shortest path from the Start to Destination node …
Node visited: C3 D3 C2 D4 D2 C1 B2 D5 E4 E2 D1 A2 B3 E5 F4 E3 F2 E1 A1
B4 A3 F5 F3 G2 F1 B1 A4 F6 G5 G3 H2 G1 A5 G6 G4
Congratulations: Found the shortest path successfully.
The path is: C3 D3 D4 E4 E5 F5 G5 G4

Enter the name of the Maze configuration file: Maze2
The Start Node is: H3
The Destination Node is: D6
Finding the shortest path from the Start to Destination node …
Node visited: H3 G3 G4 G2 G5 H2 G1 F2 G6 H5 F5 H1 E2 H6 F6 E5 E1 D2 F1
D1 C2 E6 E4 F4 E3 D4 C1 B2 C3 F3 D3 D5 B3 A2 C5 C6 B5 C4 A3 B4 D6
Congratulations: Found the shortest path successfully.
The path is: H3 G3 G4 G5 F5 E5 E4 D4 D5 C5 C6 D6

Enter the name of the Maze configuration file: Maze3
The Start Node is: F4
The Destination Node is: A2
Finding the shortest path from the Start to Destination node …
Node visited: F4 E4 E5 E3 F5 F3 F6 G5 F2 E6 G6 H5 G4 G2 E2 H6 H4 G3 H2
G1 E1 H3 H1 F1
Unsuccessful: No path can be found.

Enter the name of the Maze configuration file: Quit

      ============================================================
      |   Thank you for using the Shortest Path Maze Solver!    |
      ============================================================
```