CSS 343 Data Structures, Algorithms, and Discrete Mathematics II

2019 Winter

Assignment 3, 100 possible points (12%)

Both Part 1 & Part 2 due by: February 19 (Tuesday) 11:59 pm, submit to canvas

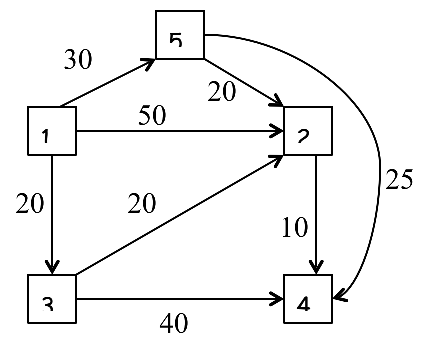
Part 1 (6%). Programming (Dijkstra's shortest path algorithm)
This project is to implement **Dijkstra's shortest path** algorithm. Your code should be able to read a data file consisting of many lines (an example file called **data31.txt** will be given, explained below) and will be able to find the lowest cost paths (**findShortestPath** function) and display the cost and path from every node to every other node (**displayAll** function). Another display routine (**display** function) will output one path in detail.

In the input data (e.g., **data31.txt)**, the first line tells the number of nodes, say n, followed by a text description of each of the 1 through n nodes, one description per line (50 chars max length). After that, each line consists of 3 integers representing an edge. If there is an edge from node 1 to node 2 with a label of 10, the data is: 1 2 10. If the first integer is zero, it means the end of the data for that one graph. There may be several graphs, each having at most 100 nodes. For example, Figure 1 (a) shows part of data from **data31.txt** (the data file you will be given) where it contains 5 nodes. For clarity, (b) shows a graph drawn based on data in (a). However, Figure 1 (b) is just for your reference. It's not part of your input data or the output you need to produce. For this lab (including part 2,) you may assume the input data file has correctly formatted, valid data.

Sample Input | picture (not part of data)

```
5
Aurora and 85th
Green Lake Starbucks
Woodland Park Zoo
Troll under bridge
PCC
1 2 50
1 3 20
1 5 30
2 4 10
3 2 20
3 4 40
5 2 20
5 4 25
0 0  0
```

(a)



(b)

Figure 1. (a) Part of the data in the provided data31.txt; (b) a graph drawn based on data in (a)

Develop the class:

1. **Class attributes** include an array of NodeData (**nodedata.h** and **nodedata.cpp** will be given), the adjacency matrix, number of nodes, and TableType array.

```
class GraphM {
 public:
    ...
 private:
    NodeData data[MAXNODES];              // data for graph nodes
    int C[MAXNODES][MAXNODES];            // Cost array, the adjacency matrix
    int size;                             // number of nodes in the graph
    TableType T[MAXNODES][MAXNODES];      // stores visited, distance, path
};
```

Here, TableType is a struct to keep the current shortest distance (and associated path info) known at any point in the algorithm.

```
struct TableType {
        bool visited;          // whether node has been visited
        int dist;              // shortest distance from source known so far
        int path;              // previous node in path of min dist
     };
```

T is a 2-dimensional array of structs because we want to work on from all nodes to all other nodes.

2. The pseudocode given in class is for only one source node (node 1 was used). To allow for checking all nodes to all other nodes, use the following pseudo code as the basis for your implementation.

```
for (int source = 1; source <= nodeSize; source++) {
      T[source][source].dist = 0;

      // finds the shortest distance from source to all other nodes
      for (int i = 1; i<= nodeSize; i++) {
         find v //not visited, shortest distance at this point
         mark v visited
         for each w adjacent to v
           if (w is not visited)
            T[source][w].dist=min(T[source][w].dist, T[source][v].dist+C[V][W])
      }
   }
```

Note: in T, index 0 is not used. This is to match with the node numbering style. Also the above pseudocode is for keeping the shortest distance only, you will need to think **how to record associated path info by yourself**.

3. You do not need to implement a complete Graph class. The only methods you must have are

- **constructor**: among others that need to be initialized, the data member T is initialized to sets all *dist* to infinity, sets all *visited* to false, and sets all *path* to 0.
- **buildGraph:** builds up graph node information and adjacency matrix of edges between each node reading from a data file.
- **insertEdge**: insert an edge into graph between two given nodes
- **removeEdge**: remove an edge between two given nodes
- **findShortestPath**: find the shortest path between every node to every other node in the graph, i.e., TableType T is updated with shortest path information

- **displayAll:** uses couts to demonstrate that the algorithm works properly. For the data in Figure 1, it will produce the sample output below (similar to, use the general format, but blanks do not need not be exact):

```
Description                From node  To node  Dijkstra's  Path
Aurora and 85th
                                1        2      40          1 3 2
                                1        3      20          1 3
                                1        4      50          1 3 2 4
                                1        5      30          1 5
Green Lake Starbucks
                                2        1      ---
                                2        3      ---
                                2        4      10          2 4
                                2        5      ---
Woodland Park Zoo
                                3        1      ---
                                3        2      20          3 2
                                3        4      30          3 2 4
                                3        5      ---
Troll under bridge
                                4        1      ---
                                4        2      ---
                                4        3      ---
                                4        5      ---
PCC
                                5        1      ---
                                5        2      20          5 2
                                5        3      ---
                                5        4      25          5 4
```

- **display**: uses couts to display the shortest distance with path info between the fromNode to toNode. For the data in Figure 1, a call of G.display(1,4) is going to produce the following output (similar to):

```
   1       4      50          1 3 2 4
Aurora and 85th
Woodland Park Zoo
Green Lake Starbucks
Troll under Aurora bridge
```

- **Some utility functions** may be needed.

List of supporting files

1. **data31.txt:** input data file;
2. **nodedata.h** and **nodedata.cpp**: NodeData class;
3. **lab3.cpp**: containing main(), to help clarify the functional requirements; **NOTE: this is used for part 2 too, so comment out things related to part 2 when you are focusing on part 1.**
4. **lab3output.txt**: correct output in using **lab3.cpp**; **NOTE: this includes results from part 2 too.**
5. **dataUWB.txt**: an additional data set for part 1, **no correct output will be provided**.

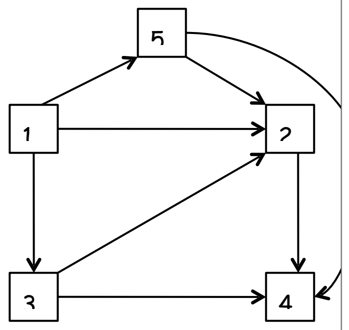
Submission Requirements:
All the rules, submission requirements, and evaluation criteria are the same as the assignment 1.

Part 2 (6%). Programming (depth-first search)
This project is to **display the graph information** and **implement depth-first** (search always **starts at node #1**).

Similar to Part 1, in the input data (e.g., **data32.txt)**, the first line tells the number of nodes, say n, followed by a text description of each of the 1 through n nodes, one description per line (50 chars max length). After that, each line consists of **2 integers** (instead of 3 integers in Part 1) representing an edge. If there is an edge from node 1 to node 2, the data is: 1 2. A zero for the first integer signifies the end of the data for that one graph. In the file, all the edges for the 1st node will be listed first, then all the edges for the 2nd node, etc. Take them as they come, **no sorting**. There may be several graphs, each having at most 100 nodes. For example, Figure 1 (a) shows part of data from **data32.txt** (the data file you will be given) where it contains 5 nodes. For clarity, (b) shows a graph drawn based on data in (a). However, Figure 1 (b) is just for your reference. It's not part of your input data or the output you need to produce. Figure 1 (c) shows the sample output where the last line is produced by calling **depthFirstSearch** function and the rest of them are produced by **displayGraph**. As you may see, edges in the input are always shown in **reverse order** in the output (see the NOTE in the Develop the class Section for more discussion). For this lab (including part 1) you may assume the input data file has correctly formatted, valid data.

Sample Input: picture (not part of data): Sample Output

```
5
Aurora and 85th
Green Lake
Starbucks
Woodland Park Zoo
Troll under bridge
PCC
1 5
1 3
1 2
2 4
3 4
3 2
5 4
5 2
0 0
```



```
Graph:
Node 1       Aurora and 85th
  edge 1 2
  edge 1 3
  edge 1 5
Node 2       Green Lake
Starbucks
  edge 2 4
Node 3       Woodland Park Zoo
  edge 3 2
  edge 3 4
Node 4       Troll under
bridge
Node 5       PCC
  edge 5 2
  edge 5 4

Depth-first ordering: 1 2 4 3
5
```

(b) (b) (c)

Figure 1. (a) Part of the data in the provided data32.txt; (b) a graph drawn based on data in (a); (c) output produced by calling **displayGraph** function followed by **depthFirstSearch**

Supporting data types:

In part 2, you will use an adjacency list (array of lists) for graph implementation. In **GraphNode**, the field `edgeHead` points to the head of a list of **EdgeNode** (which stores edge info); `visited` is used to mark whether the node has been visited; `data` is a pointer to **NodeData** that has the information about this GraphNode. Here structs are used for **GraphNode** and **EdgeNode** for simplicity, you may use classes if desired.

```
struct EdgeNode;       // forward reference for the compiler
struct GraphNode {     // structs used for simplicity, use classes if desired
   EdgeNode* edgeHead; // head of the list of edges
   NodeData* data;     // data information about each node
   bool visited;
};

struct EdgeNode {
   int adjGraphNode;  // subscript of the adjacent graph node
   EdgeNode* nextEdge;
};
```

Develop the class:

1. **Class attributes** include an array of GraphNodes among others as you see necessary.

```
class GraphL {
 public:
    ...
 private:
    // array of GraphNodes
    };
```

2. You do not need to implement a complete Graph class. The only methods you must have are

- **constructor** and **destructor**
- **buildGraph:** builds up graph node information and adjacency list of edges between each node reading from a data file.
- **displayGraph**: Displays each node information and edge in the graph (e.g., the top portion of Figure 1 (c))
- **depthFirstSearch**: Makes a depth-first search and displays each node in depth-first order (e.g., the last line in Figure 1 (c))
- **Some utility functions** may be needed.

NOTE: To simplify the process, you should always insert EdgeNodes at the beginning of the adjacency list for a GraphNode. Your output of the edges for each node will, thus, be in the reverse order in which they are listed in the input file (see Figure 1 (a) vs. (c)) Make sure to follow this simplification and **process the edges in the order they are in the list**, since it affects the depth-first ordering that you will get.

List of supporting files

1. **data32.txt:** input data file;
2. all the other files as in part 1

Submission Requirements:
All the rules, submission requirements, and evaluation criteria are the same as the assignment 2.