CS 302 – Assignment #10

Purpose: Learn concepts regarding basic graph data structures and shortest path algorithms.
Due: Wednesday (11/14) → Must be submitted on-line before class.
Points: Part A → 75 pts, Part B → 50 pts

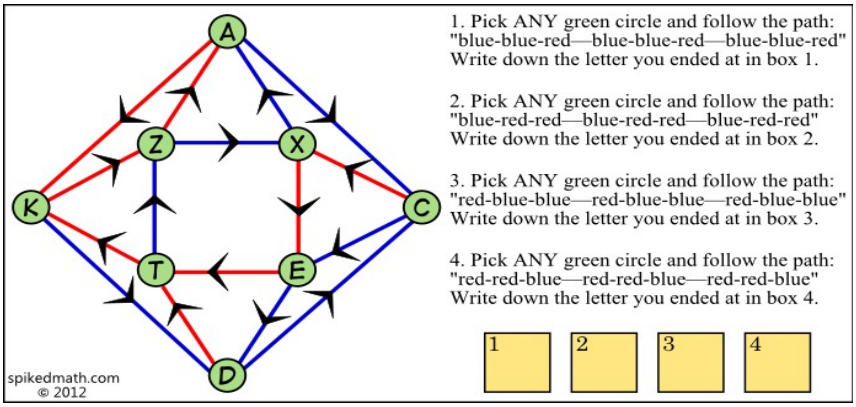
Assignment:

Part A:

Design and implement a C++ class, ***graphAlgoritms***, to implement a topological sort[1] using Kahn's algorithm and Dijkstra's single source shortest path algorithm[2]. *Note*, this class may be expanded and updated on future assignments.

In addition, Dijkstra's single source shortest path algorithm will use the priority queue object from a previous assignment.

A main will be provided that performs a series of tests. Refer to the UML descriptions for implementation details.



Part B:

Create and submit a brief write-up including the following:

- Name, Assignment, Section.
- Summary of adjacency list and adjacency matrix data structures. Include the trade-offs associated with dense and sparse matrices.
- Note some applications for the topological sort shortest and shortest path algorithms.
- Big-O for the topological sort and Dijkstra's single source shortest path algorithms.

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.
- Submit a copy of the write-up (PDF format).

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make**. You must have a valid, working *makefile*.

It should be noted that there are many variations on both these algorithms. These are the algorithms that must be implemented. Copying code from the net or another student will result in a zero for the assignment and referral to the Office of Student Conduct.

1 For more information, refer to: http://en.wikipedia.org/wiki/Topological_sorting
2 For more information, refer to: http://en.wikipedia.org/wiki/Dijkstra's_algorithm

Class Descriptions

- Graph Algorithms Class
  The graph algorithms class will implement functions specified below.

graphAlgorithms
-vertexCount: int
-title: string
-graphMatrix: **int
-dist: *int
-prev: *int
-topoNodes: *int
-topoCount: int
+graphAlgorithms(int=5)
+~graphAlgorithms()
+newGraph(int): void
+addEdge(int, int, int): void
+readGraph(const string): bool
+getVertexCount() const: int
+printMatrix() const: void
+topoSort(): void
+dijkstraSP(int): void
+printPath(const int, const int) const: void
-showPath(const int) const: void
+getGraphTitle() const: string
+setGraphTitle(const string): void
+printDistances(const int) const: void
-destroyGraph(): void
+getArgs(int, char *[], bool &, string &, int &, int &, bool &): bool

Function Descriptions

- The *graphAlgorithms()* constructor should initialize the class variables (0, "", or NULL as appropriate). If a vertex count is passed, it must be ≥ 5. If not, an error should be displayed (and no memory allocated). If the vertex count is ≥ 5, the *newGraph()* function should be called to allocate and initialize the adjacency matrix.
- The *~graphAlgorithms()* destructor should call the *destroyGraph()* function to delete the dynamically allocated memory. This includes the adjacency matrix, distances array, previous array, and the topological sort array. Additionally, reset all variables.
- The *newGraph()* function should allocate and initialize the adjacency matrix to all 0's. This function should call the *destroyGraph()* function before allocating memory to ensure there are no memory leaks.
- The *getGraphTitle()* function should return the current graph title.
- The *setGraphTitle()* function should set the graph title, over-writing any previous title.
- The *printMatrix()* function should print the current adjacency matrix in a formatted manner. This is primarily used for testing. See example output for formatting.
- The *getVertexCount()* function should return the current vertex count.

- The *getArgs()* function accept and validate the command line arguments. The arguments include the testing flag ("-t", bool), the graph file name ("-gf <fileName>" string), the source node ("-sn" <sourceNode>, integer), destination nodes ("-dn" <destNode>, integer), and optionally the show matrix flag ("-sm", bool). If no arguments are entered, a basic usage message should be displayed. The function should accept "-t" as the only argument which should set the testing flag to true (false otherwise). If the "-t" is not included the "-gf", "-sn", and "-dn" are required. The "-sm" is optional. Refer to the example executions for argument handling.
- The *addEdge()* function should add an edge to the adjacency matrix. The function will expect the *from* vertex, the *to* vertex, and the weight (in that order). The function must ensure the two passed vertexes are valid ($\leq$ *vertexCount*) and not equal. If not valid, an error should be displayed and nothing changed in the adjacency matrix. Refer to the example execution for output formatting.
- The *readGraph()* function should read a formatted graph file. The file will include a title, the vertex count, and a series of lines for the edges. The edge lines will include the *from* vertex, the *to* vertex, and the weight (in that order). The function must verify that the vertex count is ≥ 5. If not, an error message should be displayed and no memory allocated. Otherwise, the function should create a new graph by calling the *newGraph()* function and adding each edge to the adjacency matrix by calling the *addEdge()* function.
- The *topoSort()* function should perform a topological sort on the current graph. The function should implement Kahn's algorithm which stores the nodes in the *topoNodes[]* array and updates the *topoCount*. The algorithm is outlined in the topological sort section. When completed, the function should display the topological sort by displaying the *topoNodes[]* array contents.
- The *dijkstraSP()* function should implement Dijkstr's Algorithm to find the shortest path from the passed source node to each node in the graph. The function should ensure the passed source node is valid, and if not display an error message and return. The results should be placed in the dynamically allocated *dist[]* array and the *prev[]* array updated. The distance from a node to itself is 0 by definition. The function should use the Dijkstra's Shortest Path with adjacency matrix and priority queue algorithm as outlined in the assignment and in class. The last step would be to call the *printDistances()* function to display the final results.
- The *printDistances()* function should print the shortest paths from the passed source node to each node as contained in the distances array (as set in the *dijkstraSP()* function) and the previous node. Each field should be a size of 10 (i.e., **`setw(10)`**) If the distances array is not set, the function should display an error. See example output for formatting. The function should display "not reachable" if that vertex is not reachable in the graph from the source node.
- The *printPath()* function should print the path from source node (1st argument) to the destination node (2nd argument). *Note*, this function can only be called after the *dijkstraSP()* function with a matching source node (which is addressed in the provided main). The function should verify that the destination node is reachable, and if not display an appropriate error message. Otherwise, the function should display a header (see example output) and call the private *showPath()* function.
- The *showPath()* function should recursively print the path from the source node to the destination node. Only the destination node is passed. Must be recursive.
- The *destroyGraph()* function should delete all dynamically allocated memory and reinitialize the class variables as appropriate.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

Limits

The correct way to access the maximum integer is to use the C++ max() function as follows:

```
maxIntOnThisMachine = std::numeric_limits<int>::max()
```

This requires the `#include <limits>`.

Dijkstra's Shortest Path Algorithm

The following is the Dijkstra's Single Source Shortest Path Algorithm using an adjacency matrix and a minimum priority queue.

- Distances / Previous Nodes
  - dynamically create a distances array of vertex count size
  - dynamically create a previous nodes array of vertex count size
  - initialize all distances to infinite
  - initialize all elements of previous array to -1
- Priority Queue → create empty priority queue
  - value = vertex number
  - key = distance
- Insert source node
  - insert source node into priority queue
    - value = source node, key = 0
  - set distance array of source node to 0
- While priority queue is not empty
  - extract minimum distance element from priority queue
    - ***u*** = value of minimum distance element
  - iterate thought all vertexes, ***v***, adjacent to ***u***
    - new distance is; *dist*[***u***] + distance between ***u*** and ***v***
    - if there is a shorter path distance to ***v*** from ***u***
      (e.g., if current distance, new distance is < *dist*[***v***])
      - update *dist*[***v***] to new distance
      - set *prev*[***v***] to ***u***
      - insert ***v*** into priority queue (even if it is already there), key = new distance
- Print distance array *dist*[] and array *prev*[] to show all the shortest paths.

Note, the minHeap can be modified to accommodate both the vertex and the distance.

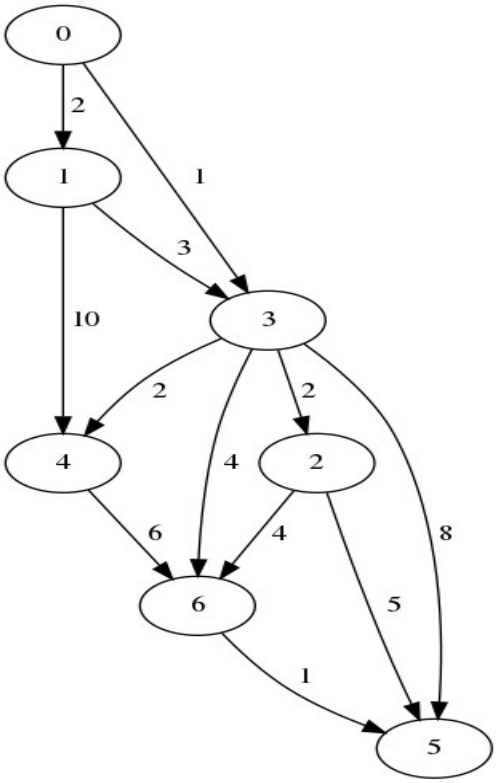
Topological Sort:

The following algorithm outlines a depth first search approach for performing a topological sort on a graph. The topological sort algorithm is as follows:

- Perform initializations
  - create in-degree array and initialize all elements to 0
  - create topo[] nodes array and initialize all elements to 0
  - initialize topo count to 0
- Compute in-degree for each vertex in graph
- En-queue each vertex with an in-degree of 0
- While queue is not empty
  - remove a vertex from queue
    - decrease in-degree by 1 for all of it neighbors
    - if in-degree of a neighbor is reduced to 0, add to queue.

The topo[] nodes array holds the topological sort.

Example:
For example, given **`graph1.txt`**




```
Graph 0 Simple Example
7
0 1 2
0 3 1
1 4 10
1 3 3
3 4 2
3 6 4
3 5 8
3 2 2
2 5 5
2 6 4
4 6 6
6 5 1
```

The adjacency matrix would be as follows:

```
Graph Adjacency Matrix:
   Graph Title: Graph 0 Simple Example

        0    1    2    3    4    5    6
     ----------------------------------
  0|    0    2   --    1   --   --   --
  1|   --    0   --    3   10   --   --
  2|   --   --    0   --   --    5    4
  3|   --   --    2    0    2    8    4
  4|   --   --   --   --    0   --    6
  5|   --   --   --   --   --    0   --
  6|   --   --   --   --   --    1    0
```

The adjacency matrix would be as follows:

```
Topological Sort:
0 1 3 2 4 6 5
```

The shortest-paths would be as follows:

```
Shortest Paths:
From Node: 0 to:
    Vertex       Dist       From
         0          0          -
         1          2          0
         2          3          3
         3          1          0
         4          3          3
         5          6          6
         6          5          3

---------------------------------------
Path from 0 to 5
  0 - 3 - 6 – 5
```

Example Execution:

Below is an example output for the main with the testing option selected.

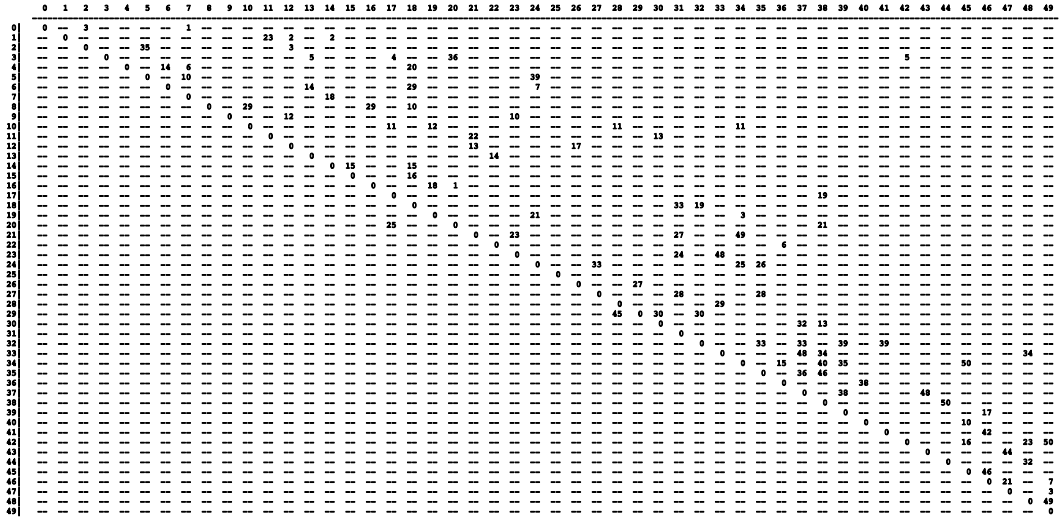
```
ed-vm% ./paths -t
================================================================
Error testing:

graphAlgorithms: Error, invalid graph size.
graphAlgorithms: Error, invalid graph size.
newGraph: Error, invalid graph size.
newGraph: Error, invalid graph size.
addEdge: error, invalid vertex.
addEdge: error, invalid vertex.
addEdge: error, invalid vertex.
addEdge: error, vertex to and from can not be the same.

topoSort: Error, no graph data.

printMatrix: Error, no graph data.

dijkstra: Error, no graph data.

dijkstra: error, invalid source.

****************************************************************
Game Over, thank you for playing.
ed-vm%
```

Below are some example outputs for the main using several different graph input files.

```
ed-vm% ./paths -gf graph1.txt -sn 0 -dn 5 -sm
****************************************************************
CS 302 - Assignment #10

================================================================
Graph Algorithms

Graph Adjacency Matrix:
   Graph Title: Graph 0 Simple Example (0 - 5)

        0   1   2   3   4   5   6
      ----------------------------
  0|    0   2  --   1  --  --  --
  1|   --   0  --   3  10  --  --
  2|   --  --   0  --  --   5   4
  3|   --  --   2   0   2   8   4
  4|   --  --  --  --   0  --   6
  5|   --  --  --  --  --   0  --
  6|   --  --  --  --  --   1   0

Topological Sort:
0 1 3 2 4 6 5

Shortest Paths:
From Node: 0 to:
    Vertex      Dist      From
         0         0         -
         1         2         0
         2         3         3
         3         1         0
         4         3         3
         5         6         6
         6         5         3

----------------------------------------
Path from 0 to 5
  0 - 3 - 6 - 5


****************************************************************
Game Over, thank you for playing.
ed-vm%
ed-vm%
ed-vm%
ed-vm%
```

```
ed-vm% ./paths -gf test/graph1.txt -sn 0 -dn 5
****************************************************************
CS 302 - Assignment #10

================================================================
Graph Algorithms

Topological Sort:
0 1 3 2 4 6 5

Shortest Paths:
From Node: 0 to:
    Vertex      Dist      From
         0         0         -
         1         2         0
         2         3         3
         3         1         0
         4         3         3
         5         6         6
         6         5         3

----------------------------------------
Path from 0 to 5
  0 - 3 - 6 - 5


****************************************************************
Game Over, thank you for playing.
ed-vm%
ed-vm% ./paths -gf graph3.txt -sn 0 -dn 9 -sm
****************************************************************
CS 302 - Assignment #10

================================================================
Graph Algorithms

Graph Adjacency Matrix:
   Graph Title: Graph 3 (0 - 9)

        0   1   2   3   4   5   6   7   8   9
      ----------------------------------------
  0|    0   4  --   8  --  --  --  --  --  --
  1|   --   0   8  --   6  11  --  --  --  --
  2|   --  --   0  --   8   4   2   7  --  --
  3|   --  --  --   0  --   7  14   9  --  --
  4|   --  --  --  --   0  10  --  --   9  --
  5|   --  --  --  --  --   0   2   4  10  --
  6|   --  --  --  --  --  --   0   1   6  --
  7|   --  --  --  --  --  --  --   0   7   1
  8|   --  --  --  --  --  --  --  --   0   2
  9|   --  --  --  --  --  --  --  --  --   0

Topological Sort:
0 1 3 2 4 5 6 7 8 9

Shortest Paths:
From Node: 0 to:
    Vertex      Dist      From
         0         0         -
         1         4         0
         2        12         1
         3         8         0
         4        10         1
         5        15         1
         6        14         2
         7        15         6
         8        19         4
         9        16         7

----------------------------------------
Path from 0 to 9
  0 - 1 - 2 - 6 - 7 - 9


****************************************************************
Game Over, thank you for playing.
ed-vm%
```

```
ed-vm%
ed-vm% ./paths -gf graph4.txt -sn 0 -dn 49 -sm
****************************************************************
CS 302 - Assignment #10

================================================================
Graph Algorithms

Graph Adjacency Matrix:
   Graph Title: Larger Graph (0 - 49)
```



```
Topological Sort:
0 1 3 4 8 9 25 2 11 42 6 10 16 5 12 13 19 20 7 21 26 22 24 17 14 23 29 27 34 15 28 30 36
18 33 40 31 32 45 35 41 37 38 39 43 44 46 48 47 49

Shortest Paths:
From Node: 0 to:
    Vertex      Dist      From
         0         0         -
         1       not reachable
         2         3         0
         3       not reachable
         4       not reachable
         5        38         2
         6       not reachable
         7         1         0
         8       not reachable
         9       not reachable
        10       not reachable
        11       not reachable
        12         6         2
        13       not reachable
        14        19         7
        15        34        14
        16       not reachable
        17       not reachable
        18        34        14
        19       not reachable
        20       not reachable
        21        19        12
        22       not reachable
        23        42        21
        24        77         5
        25       not reachable
        26        23        12
        27       110        24
        28        95        29
        29        50        26
        30        80        29
        31        46        21
        32        53        18
        33        90        23
        34        68        21
        35        86        32
        36        83        34
```

```
        37        86        32
        38        93        30
        39        92        32
        40       121        36
        41        92        32
        42        not reachable
        43       134        37
        44       143        38
        45       118        34
        46       109        39
        47       130        46
        48       124        33
        49       116        46

---------------------------------------
Path from 0 to 49
  0 - 7 - 14 - 18 - 32 - 39 - 46 - 49


****************************************************************
Game Over, thank you for playing.
ed-vm%
ed-vm%
```

Note, the font size was changed on the matrix output only on this PDF for formatting purposes.