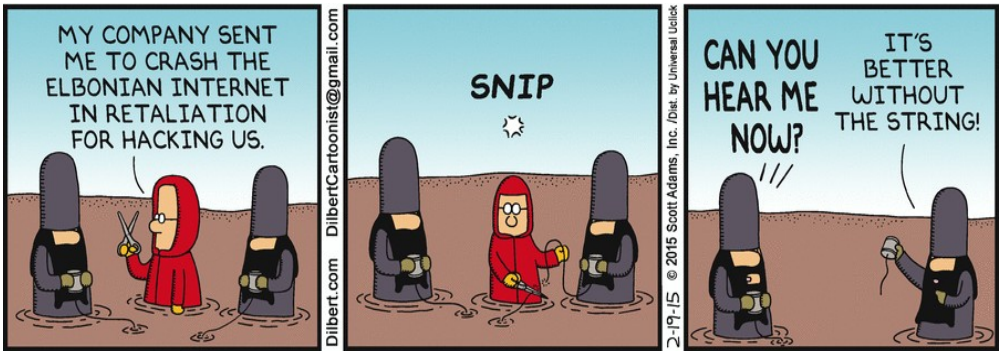
CS 302 – Assignment #09

Purpose: Learn concepts regarding basic graph algorithms.
Due: Tuesday (4/09) → Must be submitted on-line before class.
Points: Part A → 75 pts, Part B → 50 pts

Assignment:

Part A:



The Elbonia[1] Chancellor Vilmos Klavdiya will be installing Internet access for all Elbonia towns. Due to right-of-way issues, all fiber optic cables will be installed along existing roads. However, there is a limited budget and you should determine the least expensive configuration and an estimated cost to successfully complete the installation. Specifically, this means that every town will be connected via some path.

Design and implement two C++ classes, ***priorityQueue*** and ***undirectedGraph***, to implement Prim's[2] algorithm for a minimum spanning tree[3] and help the Elbonian Chancellor.

A main will be provided that performs a series of tests. An input file representing the road nd city layout will be read from the command line. 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 MST algorithm.
- Big-O for the Prim's MST algorithm and the print function.

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 (open document, word, or 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*.

1 For more information, refer to: http://mirror.uncyc.org/wiki/Elbonia
2 For more information, refer to: http://en.wikipedia.org/wiki/Prim's_algorithm
3 For more information, refer to: http://en.wikipedia.org/wiki/Minimum_spanning_tree

Class Descriptions

- Priority Queue Class
  The priority queue class will implement functions specified below.

priorityQueue<myType>
-heapNode: struct
-priority: double
-name: myType
-count: int
-heapSize: int
-myHeap: * heapNode
+priorityQueue(int=1000)
+~priorityQueue()
+entries() const: int
+insert(const myType, const double): void
+deleteMin(myType &, double &): bool
+isEmpty() const: bool
+printHeap() const: void
+readData(const string): bool
+changePriority(myType, double): void
+isIn(myType): bool
-reheapUp(int): void
-reheapDown(int): void
-buildHeap(): void
-resize(): void

Function Descriptions

- The *priorityQueue()* constructor should initialize the binary heap to an empty state. The parameter must be checked to ensure it is at least 1000 and, if not, the default value should be used.
- The *~priorityQueue()* destructor should delete the heap.
- The *entries()* function should return the total count of elements in the heap.
- The *insert()* function should insert an entry into the binary heap. If the count of heap entries exceeds the heap size, the heap must be expanded via the private *resize()* function. The heap properties must be maintained via the private *reheapUp()* function. The count should be updated.
- The private *buildHeap()* function should update the heap to apply the heap properties.
- The *deleteMin()* function should remove the minimum entry from the heap. The heap properties must be maintained via the private *reheapDown()* function. Additionally, the count should be updated. If the heap is already empty, the function should return false and otherwise return the minimum priority information (via reference) and return true.
- The *isEmpty()* function should return true if there are no elements in the heap and false otherwise.

- The *printHeap()* function should print the current heap in level order with a blank line between each level. Refer to the sample output for an example of the formatting.
- The *reheapUp()* function to recursively ensure the heap order property is maintained. Starts at tree leaf and works up to the root. Must be written recursively.
- The *readData()* function should read data from the passed file name. The file format will be <dataItem> and then <priority>. If the file can not be opened or read, the function should return false. If the file can be read, the data should be placed into the heap in the order read, the *buildHeap()* function called, and true returned.
- The *reheapDown()* function to recursively ensure the heap order property is maintained. Starts at the passed node and works down to the applicable leaf. Must be written recursively.
- The *resize()* function should create a new heap array twice the size of the existing heap, copy all entries from the current heap into the new heap, and delete the old heap. The *heapSize* should be updated accordingly.
- The *changePriority()* function should search the priority queue for the passed element and, if found, change the priority to the passed value. If the element is not found, nothing should be changed.
- The *isIn()* function should search the priority queue for the passed element, and if found, return true. The function should return false otherwise.

- Undirected Graph Class

  The undirected graph class will implement functions specified below.

undirectedGraph
-vertexCount: int
-title: string
-graphMatrix: **double
-dist: *double
-pred: *int
-cityNames: *string
+undirectedGraph(int=0)
+~undirectedGraph()
+setGraphSize(int): void
+addEdge(int, int, double): void
+readGraph(const string): bool
+getGraphSize() const: int
+printMatrix() const: void
+prims(int): void
+readCityNames(const string): bool
+getTitle() const: string
+setTitle(const string): void
-printMST() void: const
-destroyGraph(): void

Function Descriptions

- The *undirectedGraph()* constructor should initialize the class variables (0, "", or NULL as appropriate) If a vertex count is passed, it must be ≥ 5. If not, a error should be displayed (and no memory allocated). If the vertex count is ≥ 5, the s*etGraphSize()* function should be called to allocate and initialize the adjacency matrix.
- The *~undirectedGraph()* destructor should call the *destroyGraph()* function to delete the dynamically allocated memory. This includes the adjacency matrix and distances array. Additionally, reset all variables (0, "", or NULL as appropriate).
- The *setGraphSize()* function should allocate and initialize the adjacency matrix to all 0's. If te current graph exits, the *destroyGraph()* function should be called before allocating memory to ensure there are no memory leaks.
- 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 (≥ 0 and < *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 *getTitle()* function should return the current graph title.
- The *setTitle()* function should set the graph title, over-writing any previous title.
- 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. The function must verify that the vertex count is ≥ 5. If not, an error message should be display and no memory allocated. Otherwise, the function should create a new graph by calling the *setGraphSize()* function and adding each edge to the adjacency matrix by calling the *addEdge()* function. If there is an error opening or reading the file, return false. If the file read is successful, return true.
- The *getGraphSize()* function should return the current vertex count.
- The *printMatrix()* function should print the current adjacency matrix in a formatted manner. See example output for formatting.
- The *destroyGraph()* function should delete all dynamically allocated memory and reinitialize the class variables (i.e., *vertexCount*=0, etc.).
- The *prims()* function should implement Prims algorithm using a pririty queue to find a minimum spanning tree for the current graph. If a current graph does not exist, an error should be displayed.
- The *readCityNames()* function should read the Elbonian City names from the passed file. The first entry in the file will correspond to vertex 0 and so forth. The city names array should be created and the city name stored in the array. Only *vertexCount* city names should be read form the file. If there is an error reading the file, return false. If the file read is successful, return true. *Note*, Elbonia does have multiple different cities with the same name.
- The *printMST()* function should access the predecessor array (pred), the distances array (dist), and the city names array to display the final Elbonian Internet configuration in a formatted manner. This will include the MST by vertex number and then again using the city names. If the city names are not available, only the vertex's should be printed. Refer to the sample output for examples.

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.***

Prims Algorithm

```
make a heap for the vertex's
        initialize minHeap with source vertex with priority of 0
        priority of all other vertices's is INF (infinite)
make predecessor array
        initialize source index to 0
        all other elements to -1
make distances array
        initialize source index to 0.0
        all other elements to INF

while (minHeap is not empty) {
        extract min value vertex, u, from minHeap
        for every adjacent vertex, v, of u
                if v in in minHeap AND its priority is < distances[v]
                        set predecessor[v] = u
                        set dist[v] = edge weight [u][v]
                        changePriority(v, edge weight [u][v])
}

for i=1 to number of vertex's
        output from i to pred[i] with dist[i] cost
```

Limits

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

```
maxOnThisMachine = std::numeric_limits<double>::max()
```

This requires the **`#include <limits>`**. The max obtained in this manner can be used as ∞.

Example Execution:

Below is an example output for the test script and a program execution for the main.

```
ed-vm% ./elbonia -t
************************************************************
CS 302 - Assignment #11
Elbonia Internet Installation

============================================================
Error testing:

undirectedGraph: Error, invalid graph size.
undirectedGraph: Error, invalid graph size.
addEdge: error, invalid vertex.
setGraphSize: Error, invalid graph size.
setGraphSize: 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.
addEdge: error, no graph, can not add edge.
readCityNames: Error, no graph defined.
prims: Error, invalid source node.
prims: Error, invalid source node.
prims: Error, invalid source node.
printMatrix: Error, no graph data.

************************************************************
Testing Complete.
ed-vm%
ed-vm%
```

```
ed-vm% ./elbonia mst.1.in
*****************************************************************
CS 302 - Assignment #11
Elbonia Internet Installation

=================================================================
Elbonia Internet Configuration:

Graph Adjacency Matrix:
   Title: Test

            0       1       2       3       4       5
      ---------------------------------------------------
     0|       *    1.00      --    3.00      --      --
     1|    1.00       *    6.00    5.00    1.00      --
     2|      --    6.00       *      --    4.00    2.00
     3|    3.00    5.00      --       *    1.00      --
     4|      --    1.00    4.00    1.00       *    4.00
     5|      --      --    2.00      --    4.00       *


*****************************************************************
Elbonian Internet Configuration:
--------------------------------

By Vertex's:
------------
1 - 0  1.00
2 - 4  4.00
3 - 4  1.00
4 - 1  1.00
5 - 2  2.00

By Elbonian Cities:
-------------------
County - Yellowhead  1.00
Rutigliano - Rachecourt  4.00
Lille - Rachecourt  1.00
Rachecourt - County  1.00
Ogbomosho - Rutigliano  2.00

Total Cost:  $9.00k

*****************************************************************
The Elbonian Chancellor thanks you.
ed-vm%
ed-vm%
ed-vm% ./elbonia mst.2.in
*****************************************************************
CS 302 - Assignment #11
Elbonia Internet Installation

=================================================================
Elbonia Internet Configuration:

Graph Adjacency Matrix:
   Title: small test

            0       1       2       3       4       5
      ---------------------------------------------------
     0|       *    5.00      --      --      --      --
     1|    5.00       *    4.00      --    1.00      --
     2|      --    4.00       *    3.00      --    2.50
     3|      --      --    3.00       *    2.00    1.50
     4|      --    1.00      --    2.00       *      --
     5|      --      --    2.50    1.50      --       *


*****************************************************************
Elbonian Internet Configuration:
--------------------------------

By Vertex's:
------------
1 - 0  5.00
2 - 5  2.50
3 - 4  2.00
4 - 1  1.00
5 - 3  1.50
```

```
By Elbonian Cities:
-------------------
County - Yellowhead  5.00
Rutigliano - Ogbomosho  2.50
Lille - Rachecourt  2.00
Rachecourt - County  1.00
Ogbomosho - Lille  1.50

Total Cost:  $12.00k

************************************************************
The Elbonian Chancellor thanks you.
ed-vm%
ed-vm%
```