Project 04 - Graphs

Sunday, July 09 at 11:59pm

Introduction

You are required to implement two solutions for problems that are embedded in a graph data structure: Prim's algorithm in a undirected weighted graph, Dijkstra's algorithm in a directed weighted graph; you can only use adjacency lists to represent your graphs. You are also required to create UML diagrams for each of the classes that you implement. Finally, you have to provide the means to test your system by developing a menu program that allows the user to manipulate your structures.

You may use the queue, stack, and list algorithms defined in the C++ standard library (STL). You must use your own implementation for the hash table. Also, any array must be dynamically allocated.

Underflow exceptions might be generated in some of the functions you implement. Make sure exceptions are thrown and caught where appropriate.

Deliverables

- A 1 page report that explains the design of your solutions. Please include what each teammate did and approximate hours spent on the project.
- An implementation of Prim's algorithm.
- An implementation of Dijkstra's algorithm.
- A menu program to test the implemented data structures.

1 Prim's algorithm

In this part of the project, you need to implement three classes, a *Graph Class*, a *Vertex Class*, and an *Edge Class*; create their respective UML diagrams. Calculate the running time of your functions and include them in your report.

1.1 Description

A spanning tree of graph G=(V, E) is another graph T = (V, F) with the same vertices as G, and $|V| - 1$ edges of E that form a tree.

A minimum spanning tree (MST) T of G is a spanning tree whose total weight (summed over all edges of T) is minimal.

For this part of the project you will **implement Prim's algorithm to find an MST in a undirected weighted graph; an adjacency list has to be used to represent the graph**. Vertices are represented by their names in the graph. To ease the execution of some of your operations, your graph has to have a hash table to store all the vertices, where the key is the name of the vertex and the value is the vertex object. In the event you need information (data, edges, etc) about a particular vertex, you will have to hash the vertex's name and find the vertex object in the table (you may use linear probing).

1.2 Data Members

Specify all the data members your implementation needs.

1.3 Member Functions

Constructors

Defines constructor. Maximum graph size is 20 vertices.

Destructor

Defines destructor. Clean up any allocated memory.

Accessors

bool empty() Returns true if the graph is empty, false otherwise.

int degree(string v) Returns the degree of the vertex v. Throw an illegal argument exception if the argument does not correspond to an existing vertex.

int edgeCount() Returns the number of edges in the graph

bool isConnected() Determines if the graph is connected.

double adjacent(string u, string v) Returns the weight of the edge connecting vertices u and v. If the vertices are the same, return 0. If the vertices are not adjacent, return -1 (our representation of infinity). Throw an illegal argument exception if the arguments do not correspond to existing vertices.

DFS(string v) Performs DFS traversal starting on vertex v. Reset vertices after the traversal. Prints the order of vertices visited.

BFS(string v) Performs BFS traversal starting on vertex v. Reset vertices after the traversal. Prints the order of vertices visited.

MST(string v) Returns the minimum spanning tree using Prim's algorithm of those vertices which are connected to vertex v. Throw an illegal argument exception if the argument does not correspond to an existing vertex.

Mutators

void buildGraph() Reads structure from a text file and builds a undirected weighted graph.

clear() Removes all the elements in the undirected weighted graph

reset() Iterates over all vertices in the graph and marks them as unvisited.

insert(string u, string v, double w) If the weight $w < 0$ or $w = \infty$, throw an illegal argument exception. If the weight w is 0, remove any edge between u and v (if any). Otherwise, add an edge between vertices u and v with weight w. If an edge already exists, replace the weight of the edge with the new weight. If the vertices do not exist or are equal, throw an illegal argument exception.

Friends

Defines friends for this class.

2 Dijkstra's algorithm

In this part of the project, you need to implement three classes, a *DirGraph Class*, a *Vertex Class*, and an *Edge Class*; create their respective UML diagrams. Calculate the running time of your functions and include them in your report.

2.1 Description

Dijkstra's algorithm is an algorithm to find the shortest path among vertices in a graph.

For this part of the project you will **implement Dijkstra's algorithm to find the shortest path between a source vertex and the rest of the vertices in a directed weighted graph; an adjacency list has to be used to represent the graph**. Vertices are represented by their names in the graph. To ease the execution of some of your operations, your graph has to have a hash table to store all the vertices, where the key is the name of the vertex and the value is the vertex object. In the event you need information (data, edges, etc) about a particular vertex, you will have to hash the vertex's name and find the vertex object in the table.

2.2 Data Members

Specify all the data members your implementation needs.

2.3 Member Functions

Constructors

Defines constructor. Maximum graph size is 20 vertices.

Destructor

Defines destructor. Clean up any allocated memory.

Accessors

bool empty() Returns true if the graph is empty, false otherwise.

int inDegree(string v) Returns the indegree of the vertex v. Throw an illegal argument exception if the argument does not correspond to an existing vertex.

int outDegree(string v) Returns the outdegree of the vertex v. Throw an illegal argument exception if the argument does not correspond to an existing vertex.

int edgeCount() Returns the number of edges in the graph

double adjacent(string u, string v) Returns the weight of the edge connecting vertices u and v. If the vertices are the same, return 0. If the vertices are not adjacent, return -1 (our representation of infinity). Throw an illegal argument exception if the arguments do not correspond to existing vertices.

DFS(string v) Performs DFS traversal starting on vertex v. Reset vertices after the traversal. Prints the order of vertices visited.

BFS(string v) Performs BFS traversal starting on vertex v. Reset vertices after the traversal. Prints the order of vertices visited.

shortPath(string u, string v) Returns the shortest path using Dijkstra's between vertices u and v. Throw an illegal argument exception if the arguments do not correspond to existing vertices.

double distance(string u, string v) Returns the shortest distance between vertices u and v. Throw an illegal argument exception if the arguments do not correspond to existing vertices. The distance between a vertex and itself is 0.0. The distance between vertices that are not connected is -1 (our representation of infinity).

Mutators

void buildGraph() Reads structure from a text file and builds a directed weighted graph.

clear() Removes all the elements in the undirected weighted graph

reset() Iterates over all vertices and marks them as unvisited.

insert(string u, string v, double w) If the weight w ≤ 0, throw an illegal argument exception. If the weight is w > 0, add an edge between vertices u and v. If an edge already exists, replace the weight of the edge with the new weight. If the vertices do not exist or are equal, throw an illegal argument exception.

Friends

Defines friends for this class.

3 Graph Representation

Format for the text file

The example text file for the buildGraph() method is uploaded to Canvas. Up to twenty vertices (string) will be listed with white spaces between, followed by a new line. Next, each edge will be listed: u (string), space, v (string), space, weight (double), new line. The same format is used for directed and undirected graphs.

Format for printing Prim's MST (edges and total weight)

A B 4
A C 6.4
C F 3
A D 9
Distance: 22.4

Format for printing Dijkstra's shortest path (distance to each node from A)

A 0
B 4
C 3.1
D 6
F 7

4 The Menu Program

In order to test your program, you are required to implement a menu program that provides the means to run each of the functions in your classes (name your executable proj4). The TA will choose one group to demo the project.

Format for the Menu Program

First, prompt the user for the type of graph (char 'd' for directed or char 'u' for undirected). Next, prompt for a file name (using path provided) for the .txt file to load the graph. The format of an example .txt file is provided on Canvas for you to be able to test your work. Next, give the following options for the specific graph (please have them in this **EXACT** order for grading purposes):

Undirected Graph (Prim's)

1. Empty?

2. Degree (v)
3. Edge count
4. Connected?
5. Adjacent (u, v)?
6. DFS (v)
7. BFS (v)
8. Print MST (v)
9. Clear
10. Insert (u, v, w)
11. Exit

Directed Graph (Dijkstra's)

1. Empty?
2. InDegree (v)
3. OutDegree (v)
4. Edge count
5. Adjacent (u, v)?
6. DFS (v)
7. BFS (v)
8. Print Short path (v)
9. Clear
10. Insert (u, v, w)
11. Exit

Submit the following files to Canvas (named **EXACTLY** as shown below **without zipping** for grading purposes: (you may also include implementation files with the header files if you would like)

1. menu.cpp
2. graph.h
3. vertex.h
4. edge.h
5. dirGraph.h
6. makefile (name your executable 'proj4')
7. project4.pdf

The rubric is as follows:

1. A program that does not compile will result in a zero
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Report 10%
5. Functionality 70% (functions were declared and implemented as required)

5 The Project Report

You must include everything you consider relevant in your design, UML diagrams, and algorithm analysis.

6 Acknowledgment

This project was created based on the work shared by the University of Waterloo.