Project 04 - Graphs

July 4, 2016

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.

It is strictly prohibited to use the structures and/or algorithms defined in the C++ standard library (STL). So, if your design requires a list, queue, stack, hash table, or a tree, you can only use your own implementations. 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 report that explains the design of your solutions.
- 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.

If your solution requires a list, queue, stack, hash table, or a tree, you can only use your own implementations. Also, any array needed must be dynamically allocated.

1.2 Data Members

Specify all the data members your implementation needs.

1.3 Member Functions

Constructors (1 point)

Defines constructor. Minimum graph size is 20.

Destructor (1 point)

Defines destructor. Clean up any allocated memory.

Accessors

bool empty() (1 point) Returns true if the graph is empty, false otherwise.

int degree(string v)(1 point) Returns the degree of the vertex v. Throw an illegal argument exception if the argument does not correspond to an existing vertex.

int edgeCount(string v)(1 point) Returns the number of edges of the vertex v.

int edgeCount()(1 point) Returns the number of edges in the graph

bool isConnected() (1 point) Determines if the graph is connected.

double adjacent(string u, string v) (2 points) 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 infinity (your representation of infinity). Throw an illegal argument exception if the arguments do not correspond to existing vertices.

DFS(string v) (5 points) Performs BFS traversal starting on vertex v. Reset vertices after the traversal.

BFS(string v) (5 points) Performs BFS traversal starting on vertex v. Reset vertices after the traversal.

MST(string v) (20 points) Returns the minimum spanning tree of those vertices which are connected to vertex v. Throw an illegal argument exception if the arguments do not correspond to existing vertices.

Mutators

void buildGraph() (5 points) Reads structure from a text file and builds a undirected weighted graph.

clear() (5 points) Removes all the elements in the undirected weighted graph

reset() (5 points) Iterates over all vertices in the graph and marks them as unvisited.

insert(string u, string v, double w) (10 points) 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.

del(string v) (5 points) Removes vertex v from the graph, and updates connections in the graph.

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.

If your solution requires a list, queue, stack, hash table, or a tree, you can only use your own implementations. Also, any array needed must be dynamically allocated.

2.2 Data Members

Specify all the data members your implementation needs.

2.3 Member Functions

Constructors (1 point)

Defines constructor. Minimum graph size is 20.

Destructor (1 point)

Defines destructor. Clean up any allocated memory.

Accessors

bool empty() (1 point) Returns true if the graph is empty, false otherwise.

int indegree(string v)(1 point) 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)(1 point) Returns the outdegree of the vertex v. Throw an illegal argument exception if the argument does not correspond to an existing vertex.

int edgeCount()(1 point) Returns the number of edges in the graph

double adjacent(string u, string v) (2 points) 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 infinity (your representation of infinity). Throw an illegal argument exception if the arguments do not correspond to existing vertices.

DFS(string v) (5 points) Performs DFS traversal starting on vertex v. Reset vertices after the traversal.

BFS(string v) (5 points) Performs BFS traversal starting on vertex v. Reset vertices after the traversal.

shortPath(string u, string v) (20 points) Returns the shortest path 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)(10 points) 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 infinity.

Mutators

void buildGraph() (5 points) Reads structure from a text file and builds a directed weighted graph.

clear() (5 points) Removes all the elements in the undirected weighted graph

reset() (5 points) Iterates over all vertices and marks them as unvisited.

insert(string u, string v, double w) (10 points) If the weight w $\leq$ 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.

del(string v) (5 points) Removes vertex v from the graph, and updates connections in the directed weight graph.

Friends

Defines friends for this class.

3 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. Vertices will store *doubles* in their data field. The TA will choose one member of your group to defend the demo, and the same grade will be assigned to all of the members of the group.

4 Rubric

This is how your project is going to be graded.

1. Report 40%.
2. Demo 60%. 30% for each problem.

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.