ECE250: Lab Project 4

Due Date: Saturday, December 3, 2016 – 11:00PM

1 Project Description

In this project, you implement the Minimum Spanning Tree (MST) of a weighted undirected graph, using the Kruskal's algorithm. We consider the nodes in the graph to be numbered from 0 to n – 1. This means a graph with 4 nodes, has nodes named 0, 1, 2 and 3. Each edge has a weight (a positive number and of double type) associated with it.

2 How to Represent a Graph

You can represent your graph using adjacency matrix, with the rows representing source vertices and columns representing destination vertices. The cost for one edge is stored between each pair of vertices. As an alternative you can decide to represent your graph using an adjacency list, in which vertices are stored as objects, with every vertex storing a list of adjacent vertices.

3 How to Test Your Program

We use drivers and tester classes for automated marking, and provide them for you to use while you build your solution. We also provide you with basic test cases, which can serve as a sample for you to create more comprehensive test cases. You can find the testing files on the course website.

4 How to Submit Your Program

Once you have completed your solution, and tested it comprehensively, you need to build a compressed file, in tar.gz format, which should contain the file:

- Weighted_graph.h
- Disjoint_sets.h

Build your tar file using the UNIX tar command as given below:

- *tar –cvzf xxxxxxxx_pn.tar.gz* Weighted_graph.h Disjoint_sets.h

where *xxxxxxxx* is your UW user id (ie. jsmith), and *n* is the project number which is 4 for this project. All characters in the file name must be lowercase. Submit your tar.gz file using LEARN, in the drop box corresponding to this project.

5 Class Specifications

In this project, you will implement two classes: *Weighted_graph.h* that represents a weighted undirected graph, and *Disjoint_sets.h* that represents the disjoint sets data structure using linked list.

5.1 Weighted_graph.h

The Weighted_graph class represents a weighted undirected graph. One of its methods obtains the minimum spanning tree of a graph, using Kruskal's algorithm.

Member Variables

Depending on how you wish to represent your graph, you need to include here a member variable to hold an adjacency matrix, or those variables required to represent an adjacency list. You will also need other member variables to fully represent the properties of this graph (such as the degree of each node).

Note: If you choose to represent your graph as an adjacency matrix, you will need to define a two-dimensional array in C++, and use dynamic memory allocation, since the number of nodes is variable (given as an input parameter in the constructor). Therefore, your member variable definition will define the matrix using an array of pointers to pointers: double **graph_matrix;

Constructor

Weighted_graph (int n = 10) - Constructs a weighted undirected graph with *n* vertices (by default 50). Assume that initially there are no connections in the graph (edges will be inserted with the "insert" method). Throw an *illegal_argument* exception if the argument is less than 0.

Destructor
~ *Weighted_graph ()* - Cleans up any allocated memory.

Accessors
The class has three accessors:

- *int degree(int n) const* - Returns the degree of the vertex n. Throw an *illegal_argument* exception if the argument does not correspond to an existing vertex.
- *int edge_count() const* - Returns the number of edges in the graph.
- *std::pair<double, int> minimum_spanning_tree() const* - Uses Kruskal's algorithm to find the minimum spanning tree. It returns the weight of the minimum spanning tree and the number of edges that were tested for adding into the minimum spanning tree.

Mutators
The class has three mutators:

- *bool insert_edge(int i, int j, double w)* - If *i* equals *j* and are in the graph, return false. Otherwise, either add a new edge from vertex *i* to vertex *j* or, if the edge already exists, update the weight and return true. Recall that the graph is undirected. If *i* or *j* are outside the range of [0..n-1] or if the weight *w* is less than or equal to zero, throw an *illegal_argument* exception.
- *bool erase_edge(int i, int j)* - If an edge between nodes *i* and *j* exists, remove the edge. In this case or if *i* equals *j* return true. Otherwise, if no edge exists, return false. If *i* or *j* are outside the range of [0..n-1], throw an *illegal_argument* exception.
- *void clear_edges()* - Removes all the edges from the graph.

5.2 Disjoint_sets.h
To build the minimum spanning tree T, the Kruskal's algorithm adds one edge to the T (initialized with an empty graph) in each step. To make sure that this procedure does not form loops in the tree, we need to keep track of the connected components of T. *Disjoint sets* is a well-known data structure for grouping *n* elements (nodes) into a collection of disjoint sets (connected components). In this project, the goal is to implement the disjoint sets data structure using linked list (you can read information on disjoint sets from Chapter 21 of CLRS book).

Member Variables
This class has at least the following member variables:

- *ll_entry* nodes*: An array of pointers that keeps a pointer to each node entry in the linked lists.
- *set_info* sets*: An array of pointers that keeps the information for each set. This information includes the pointers to head and tail of the set as well as an integer that keeps the size of the set.
- *int set_counter*: A variable that saves the current number of sets.
- *int initial_num_sets*: A variable that saves the initial number of sets.

Constructor
Disjoint_sets(int n) - Constructs a disjoint sets data structures with *n* sets each containing one element (therefore the total number of elements in *n*).

Destructor
~Disjoint_sets() - Cleans up any allocated memory.

Accessors
This class has two accessors:

- *int num_sets()* - Returns the number of sets.
- *int find_set(int iitem)* - Returns the representative of the set that the node *item* belongs to.

Mutator
This class has this mutator:

- *void union_sets(int node_index1, int node_index2)* - Unites the dynamic sets that contain *node_index1* and *node_index2.*