EE 205 Data Structures and Algorithms **Fall 2018**

Project 4: Graph Structure & Algoritms

Lecturer: Yung Yi *TA: Joonki Hong, Taeyoung Lee*

Introduction

In this project, you are expected to solve the famous problems in graph theroy with the graph strucure and graph algorithms learned in the class. These two famous problems are finding the shortest path and MST (Minimum Spanning Tree).

In various applications (e.g. web mapping, road networks, ethernet networks, ...), finding the shortest path from the one node to the another is extremely important to reduce cost, energy, and complexity. MST also have direct applications in the design of networks, including computer networks, telecommunication networks, transportation networks, and so on. In our scenario, we will take a look into the road networks between the airports all over the world.

Task1: Implement ADT Graph

In the first part of this project, you will implement ADT graph functionalities with adajacency list structure. General undirected graph consists of vertices and edges. In ADT graph data structure, theses vertices and edges are handled as objects, supporting some functionalities. First, the vertices support following functionalities:

incidentEdges():	Return an edge list of the edges incident on u.
isAdjacentTo(v):	Test whether vertices u and v are adjacent.

Functionalities of ADT_Node object.

Next, edges support following functionalities:

endVertices():	Return a vertex list containing e's end vertices.
opposite(v):	Return the end vertex of edge e distinct from vertex v; an error occurs if e is not incident on v.
isAdjacentTo(f):	Test whether edges e and f are adjacent.
isIncidentOn(v):	Test whether e is incident on v.

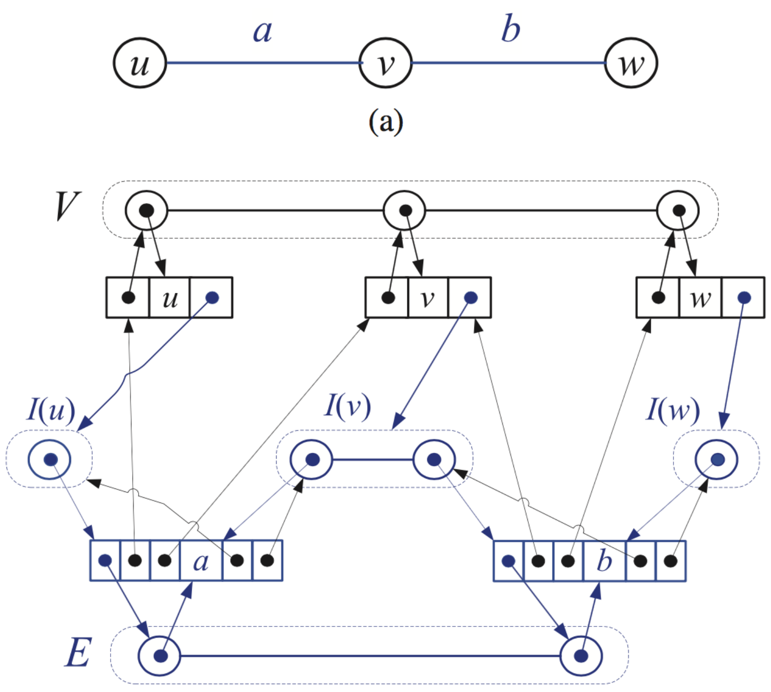
Functionalities of ADT_Edge object.

The ADT graph itself is also implemeted as an object, which contains several verticies and edges. The object of ADT graph should support functions for handling vertices and edges, such as insert or erase, to save the connectivity information of the graph. The functionalities of ADT graph follows:

vertices():	Return a vertex list of all the vertices of the graph.
edges():	Return an edge list of all the edges of the graph.
insertVertex(x):	Insert and return a new vertex storing element x.
insertEdge(v,w,x):	Insert and return a new undirected edge with end vertices v and w and storing element x.
eraseVertex(v):	Remove vertex v and all its incident edges.
eraseEdge(e):	Remove edge e.

Functionalities of ADT_Graph object.

You can implement ADT graph funtionalities in many ways, but in this project, you will implement ADT graph using adjacency list structure. The visualization of adjacency list structure is given bellow:



Visualization of the adjacency list structure.

The implementation of ADT graph with adjacency list srtucture is known to have low computational complexity for various operations of ADT_Edge, ADT_Node, and ADT_Graph. The following figure shows the computational complexity of ADT operations with adajacency list structure.

Operation	***Time***
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
v.incidentEdges()	$O(\deg(v))$
v.isAdjacentTo(w)	$O(\min(\deg(v), \deg(w)))$
isIncidentOn	$O(1)$
insertVertex, insertEdge, eraseEdge,	$O(1)$
eraseVertex(v)	$O(\deg(v))$

Complexity of ADT opeartions.

- Every vertices are saved in vertex list V and every edges are saved in edge list E such that the operation `vertices()` and `edges()` can be done in $O(n)$ and $O(m)$ order.
- Each vertices $(u, v, w, ...)$ saves the iterator that marks its postion in the vertex list and the pointer to the incidence colleciton $I(v)$.
- Incidence collection $I(v)$ is a list that stores references to the edges incidencent on v.
- Similarly, each edges $(a, b, ...)$ has the iterator that marks its position in the edge list.
- As each edge objects has two end vertices, it has iterators that save the position in the first vertex incidence collection and in the second vertex incidnce collection.
- Moreover, each edge objects has pointers to the first vertex object and the second vertex object.

Check textbook p.603 for detailed description.

Hints for Task 1

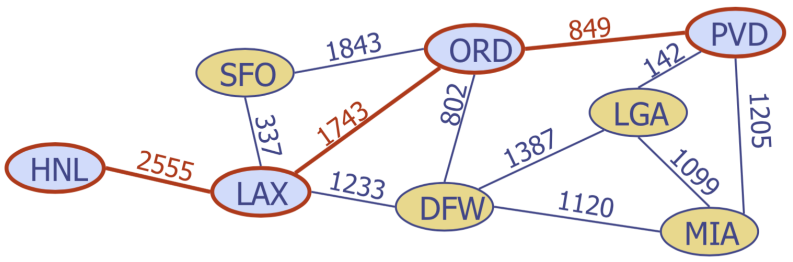
- Refer to the header file and the constructor of each class. First, check the dependency between the classes.
- There are comments and debug prints for you in `'ADT_graph.cpp'`.
- Many of the functions are already complete. Please use those functions to complete the others.

Task2: Graph Algorithms

In the second part of this project, you are expected to solve the shortest path and MST problem using ADT graph data structure that you implemented in the first part.

Shortest Path Problem

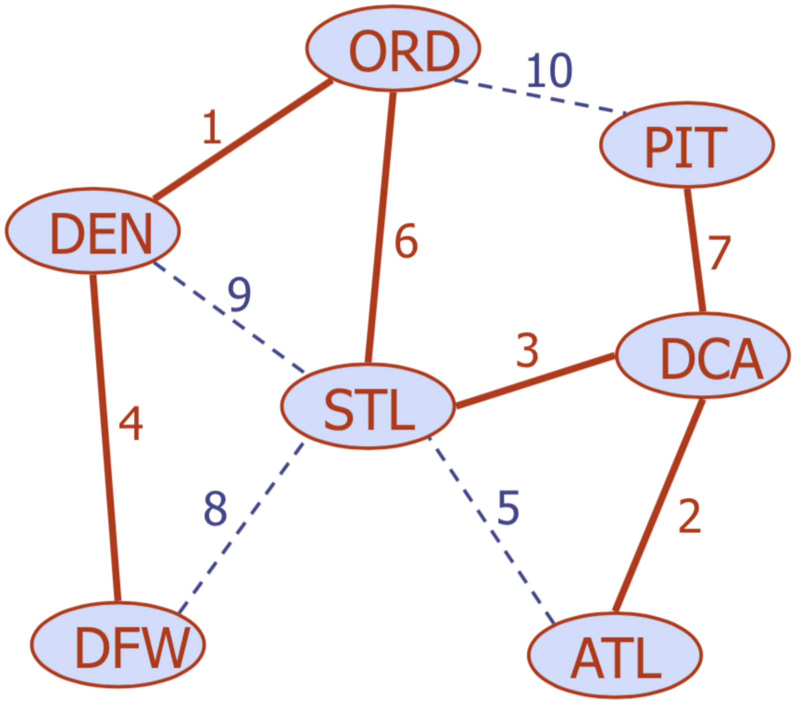
Shortest path problem is to find the shortest path from one vertex to another one. In this project, you need to find the shortest path from one airprot to another airport (e.g. HNL to PVD). You can use any other algorithms to find the shortest path, but we recommend you to use Dijistra's algorithm in this project.



Shortest path problem.

MST Problem

MST is to find the spanning tree that has the smallest sum of the branch lengths. In this project, you will be given road networks between airports and you should find MST for the given road networks. You can use any other algorithms to find MST, but we recommend you to use Kruskal's algorithm in this project.



Minimum spanning tree problem.

Check lecture slides for detailed description.

Guideline

Skeleton Code

We will provide a skeleton code for implementing ADT graph data structure and testing the operations for ADT graph. `'ADT_graph.h'` includes the declarations of classses and functions for adjacency list structure. `'ADT_graph.cpp'` includes the function defiintions for adjacency list structure. We will provide some of the imeplementations for you, but you should implement rest of it by yourself. `'main.cpp'` will help you check your implementation is successful or not. There are various tests to check the operations of ADT_Node, ADT_Edge, and ADT_Graph.

Task1

For Task1, you should not change anything `'ADT_graph.h'`. We will grade every operations for ADT_Node, ADT_Edge, and ADT_Graph with modified `'main.cpp'` file.

Task2

For Task2, you can add some variables or fucntions to `'ADT_graph.h'` and `'ADT_graph.cpp'` if you want. For Dijstra's algorithm you may need priority queue data structure, and for Kruskal's algorithm, you may need set data structure. If you need another data structure except ADT graph, you can use STL library. However, you should use your own implementation for ADT graph data structure.

For Task2, we will give you `'graph.txt'` and `'shortest_path_task.txt'` as input files. Then your main file `'task2.cpp'` should output `'ans_SP.txt'` and `'ans_MST.txt'`. You should generate your output files with following commands.

```
g++ task2.cpp ADT_graph.cpp -o task2_solver

./task2_solver graph.txt shortest_path_task.txt ans_SP.txt ans_MST.txt
```

I will provide you sample input files and output files for your debugging, but the evaluation will be done by modified input files. Rules for input files and output files are explained in `'README.txt'`. You can search for the algorithms in the internet but don't just copy and paste the code.

Readme

Please turn in a Readme file along with your code which states any specific consideration we should take to grade your code properly or anything that you thinks to be ambiguous in interpreting this project (e.g. I have discussed with Suho Shin / I changed the input format / I use set function instead of constructor...).

Submission Format

You should submit seperate directories for Task1 and Task2.

- For Task1, make `'project4_task1'` directory and put your `'ADT_graph.cpp'` file.

- For Task2, make `'project4_task2'` directory and put your main file for the shortest path problem and MST problem as `'task2.cpp'`. Also include your `'ADT_graph.cpp'` and `'ADT_graph.h'` in the folder. These `'ADT_graph.cpp'` and `'ADT_graph.h'` files can be different from Task1 because you can modify these files in Task2, if you want.

  Then you should compress two folders into one file `'proj4_20xxyyyy.zip'`.

Grading Policy

Task1

ADT_Node

`isAdjacentTo(v)`: 0.5

ADT_Edge

`opposite(v)`: 0.5

`isAdjacentTo(f)`: 0.5

`isIncidentOn(v)`: 0.5

ADT_Graph

`vertices()`: 0.5

`edges(v)`: 0.5

`insertVertex(v)`: 0.5

`insertEdges(v)`: 0.5

`eraseVertex(v)`: 0.5

`eraseEdge(v)`: 0.5

Task2

Shortest path: 5

MST: 5