ECE 650 Report 3

Graph API

Group members: Zhao Jun(20683461), Liu Chi(20649042)

1. DESIGN CHOICE

In current design, the map API is composed of three classes: Vertex, Edge, and Graph. In particular, Vertex and Edge are two basic classes, based on which the class Graph is constructed.

1.1 Vertex

For the class Vertex, each vertex has a *name, vertex type* and *(x, y*) coordinates. *name* is in <*string*> format and considered as the object ID. *vertexType = 0* represents the vertex is a point of interest and *vertexType = 1* represents the vertex is an intersection. *(x,y)* coordinates are just used for drawing the graph to demonstrate and test.

1.2 Edge

As to the class Edge, each edge has 2 end vertex names(*v1* and *v2*), *directional*, *speed*, *length* and *eventType*. If *directional = 0*, it means the edge is bidirectional. When *directional = 1* or *2*, the edge is single directional, which means edge direction is from *v1* to *v2* or from *v2* to *v1* respectively. The unit of speed is k*m/h* and the unit of length is *km*. Additionally, e*ventType = 0* represents the edge is open; *eventType =1* means the edge is closed; *eventType =2* represents the event on the edge is hazard.

1.3 Graph

The Graph class is the crucial part of this mapping API. Design details are presented as follows:

(1)Map representation

The vertices and the edges in the graph are represented by a <*string*> ID. Their detailed information is stored in the private *map* data structures, respectively v*ertexMap<string, Vertex>* and *edgeMap<string, Edge>*. Similarly, the computed road information is stored in *roadMap<string, vector<string> >* with a <*string*> name as the road ID.

As to the representation of vertices relationship, the connection information is stored in the adjacency list *adjOutList<string, vector<string> >* and *adjInList<string, vector<string> >*, where *adjOutList* contains the out-neighbor edges of vertices, while *adjInList* keeps the in-neighbor edges of vertices. The “key” entities are represented by the vertex ID and the “value” entities are represented by the edge ID.

Additionally, we also allocate three static variables, *numVertex*, *numEdge* and *numRoad*, to count the numbers of vertices, edges and roads in the Graph object.

(2)Key method implementations

● *addVertex(string label, vertexType type, int x, int y):*

The vertex information is stored in the *vertexMap* and the vertex <*string*> ID is added into *adjOutList* and *adjInList* with an empty <*string*> list.

● *addEdge(string label, string v1, string v2, dirType dir, int speed, int length, eventType type):*

The vertexes, *v1* and *v2*, are first checked whether they are in graph. If yes, the edge information is stored in the *edgeMap*. Then, according to the directional information, the <*string*> ID of edge is appended in the corresponding list of *adjOutList* and *adjInList*.

● *edgeEvent(string label, eventType type):*

Find the edge in *edgeMap* according to its <*string*> ID and set the eventType value.

● *road(string name, vector<string> edges):*

A list of edges is added into *roadMap* with a name as the key value.

● *vertex(string name):*

Check whether the specific name of vertex is in the *vertexMap*. If it exists, return the corresponding vertex pointer. Otherwise, return a null pointer.

● *trip(string fromVertex, string toVertex, string label, tripType type = SHORTEST):*

The Dijkstra algorithm is applied to find the optimal path with the minimum sum of edge priority values. This API function provides two choices for users. When *type = SHORTEST*, the priority value of edge is represented with the edge length to search for the shortest trip; As *type = FASTEST*, the edge priority value is equal to the quotient of edge length and speed to find the fastest path in the map. In order to facilitate the implementation of this algorithm, the data structure "priority queue" is used to select the vertex with minimal priority value in the queue.

● *store(string filename)/retrieve(string filename):*

All the graph information is stored/retrieved in/from a text file in the specific format described in the following session.

(3)Input/output implementation

The input/output language is defined in the following format. The information of vertices, edges and road begins with the key words *"#vertex", "#edge"* and *"#path"*. In the data lines, every information unit is separated by a whitespace. The information block ends with the key word "#end".

Input/output format:

#vertex: name; vertex_type; x; y
CLV 0 2 1
…
#edge: name; v1_name; v2_name; directional; speed; length; eventType
E1 CLV CLN 0 50 1 0
…
#path: name; edge1_name; edge2_name; ...; edgeN_name
Shortest_path E1 E2 E3 E4
...
#end

---

2. RESULTS

In this session, two typical test cases will be presented. The first case compares the trip planning under normal conditions with that under CLOSE event on certain edges, both of which are based on the SHORTEST tripType. The second case compares the shortest path with the fastest path under the same conditions.

2.1 Test case 1

The test code in the *main()* routine is shown as below and the results is presented in Figure 1.

```
Graph m1;
m1.retrieve("./test cases/test1.txt");
m1.edgeEvent("E26",CLOSE);
m1.trip("CLV", "340", "path1", SHORTEST);
m1.edgeEvent("E26",OPEN);
m1.trip("CLV", "340", "path2", SHORTEST);
m1.store("output.txt");
return 0;
```

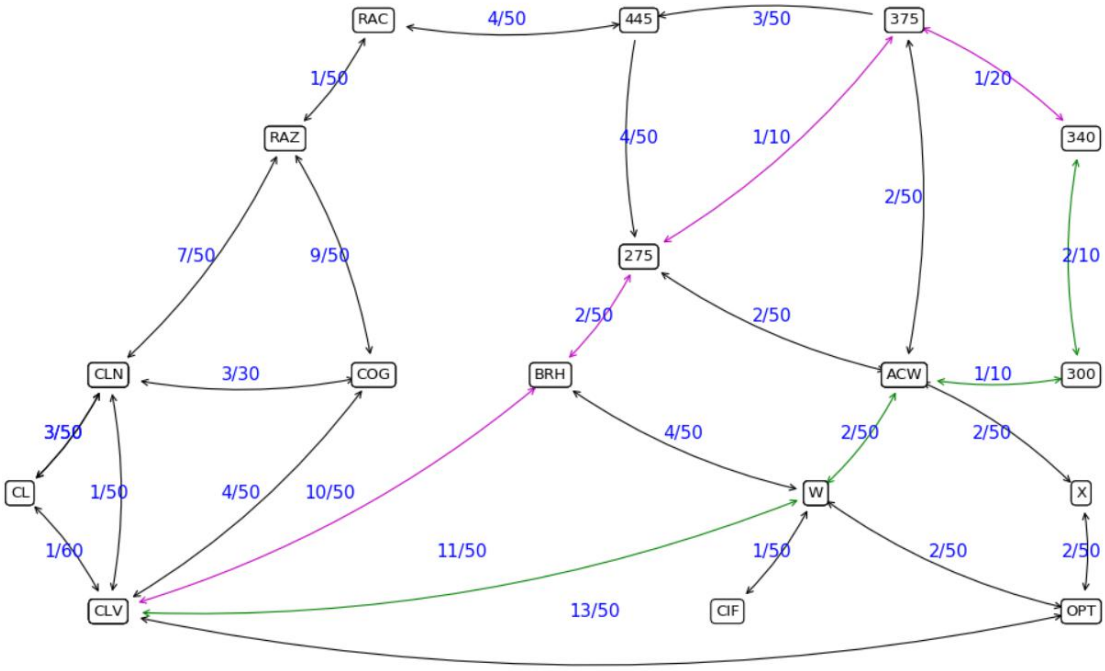

Figure 1. Trip planning of the test case 1 (OPEN vs CLOSE)

Note: The blue annotations(1/50, 3/30, etc) means length(km)/speed(km/h)

As shown in Figure 1, when the edge *275-375* is set to be *CLOSE*. The shortest path from *CLV* to *340* is the green line. As this edge is reset to *OPEN*, the shortest path becomes the purple line. The comparison of length between these two different paths is presented as follows:

Purple line:

$$length = 10 + 2 + 1 + 1 = 14km$$

Green line:

$$length = 11 + 2 + 1 + 2 = 16km$$

2.2 Test case 2:

The test code in the *main()* routine is shown as below and the results is presented in Figure 2.

```
Graph m2;
m2.retrieve("./test cases/test1.txt");
m2.trip("CLV", "375", "path1", SHORTEST);
m2.trip("CLV", "375", "path1", FASTEST);
m2.store("output.txt");
return 0;
```

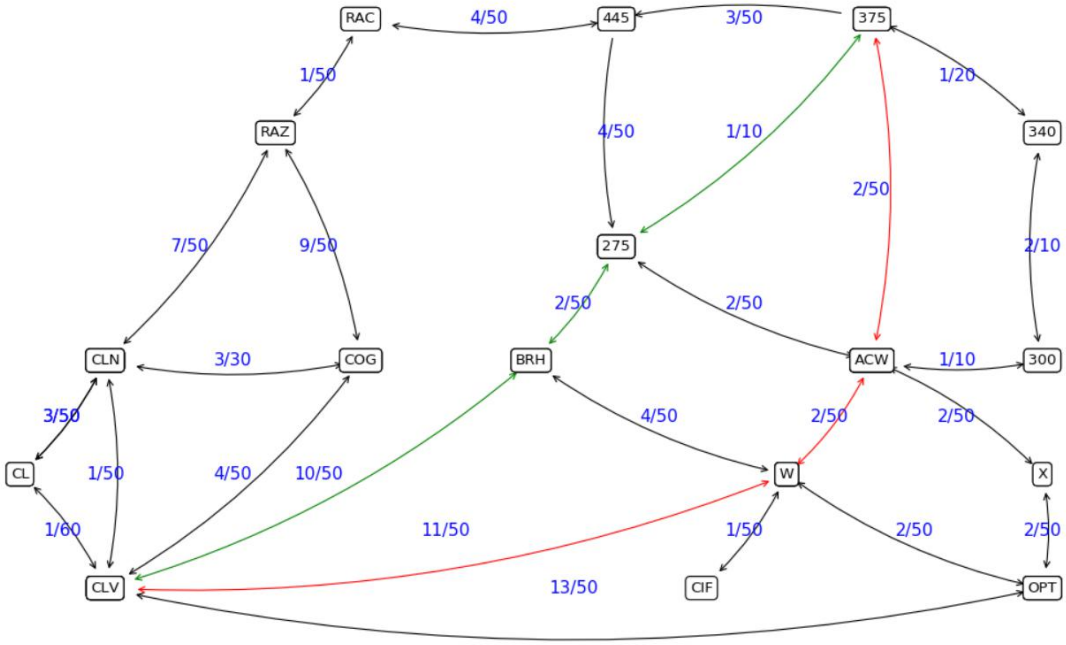

Figure 2. Trip planning of the test case 2 (SHORTEST vs FASTEST)

Note: The blue annotations(1/50, 3/30, etc) means length(km)/speed(km/h)

Figure 2 shows two paths from *CLV* to *375*. The red one is the fastest path,while the green one is the shortest path with consideration of the travel time. The comparison of length and travel time between these two different paths is presented as follows:

Red line:

$$length = 11 + 2 + 2 = 15km$$

$$time = \frac{11}{50} + \frac{2}{50} + \frac{2}{50} = 0.3h = 18\,\text{min}$$

Green line:

$$length = 10 + 2 + 1 = 13km$$

$$time = \frac{10}{50} + \frac{1}{10} + \frac{2}{50} = 0.34h = 20.4\,\text{min}$$