CSC2040 Data Structures, Algorithms and Programming Languages

Practical 13

Graphs

Wednesday 28th February 2018

As described in the lecture notes, a graph can be represented using two different methods: (1) adjacency list representation, and (2) adjacency matrix representation. The lecture notes have detailed the first method. In this practical, we focus on the second method. We will create a C++ class that can represent arbitrary directed and undirected graphs. We will use dynamically allocated 2-D arrays to store their adjacency matrices.

Program 1 Graph Representation

Re-develop the `Graph` class, detailed in the lecture notes, by replacing the `list<Edge>* edges` with `double** edges` (thus, in this new representation, you will not need the Edge class any longer.) In the new presentation, `edges` is a dynamic 2-D matrix to store the edge information of a graph, with the row indices `i` representing the source vertices, column indices `j` representing the destination vertices, and the values `edges[i][j]` representing the weights. The new `Graph` class based on the adjacency matrix representation may take the following structure:

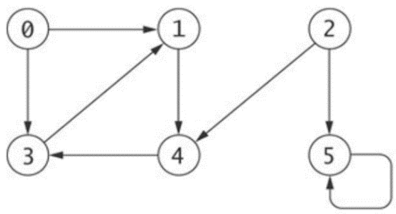
```
#ifndef GRAPH_H
#define GRAPH_H

class Graph
{
public:
    // constructor - create a graph from a file with the given name
    Graph(char* fname);
    // destructor
    ~Graph();

    // return the number of vertices
    int get_num_v() const { return num_v; }
    // return the weight between the given source & dest vertices
    // weight = 0.0 means no edge
    double get_edge(int source, int dest) const;

private:
    // the graph
    int num_v;       // number of vertices
    bool directed;   // direction
    double** edges;  // 2-D adjacency matrix holding weighets for edges
                     // from num_v vertices to num_v vertices
};
#endif
```

1. Write the new constructor `Graph(char* fname)`, which will open and read a **graph definition file** of a format defined in the lecture notes, with the given name `fname`, and use the information to create the adjacency matrix `edges`.
    a. For **directed** graphs, set `directed = true`, and then create a full `num_v` x `num_v` matrix `edges` to store the edge information.
    b. For **undirected** graphs, set `directed = false`. Because the matrix is symmetric, you should only create a triangular matrix to store the edge information, i.e., the first row has one element, the second row has two elements, the third row has three elements, …, and the last row has `num_v` elements. This saves memory.

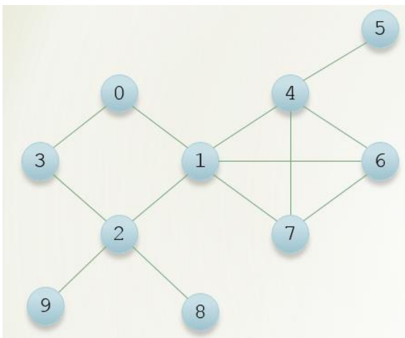
c. Initialise all `edges[i][j] = 0.0`. In our system, we assume that `edges[i][j] = 0.0` indicates that there is no direct edge between source vertex `i` and desination vertex `j`.

d. Then read the edge records from the graph definition file line by line to set the corresponding `edges[source][destination] = weight`. For undirected graphs, if `destination > source`, then set `edges[destination][source] = weight.`

2. Write the new destructor to delete the matrix `edges` properly.
3. Write the new `get_edge` function, which returns the weight of the edge of the specified `source` and `destination`.
4. Using the testing main function in the lecture notes as an example, write a testing `main` function to test your new `Graph` class by using the following two graph examples, one directed and the other undirected. You should produce the same results as shown in the lecture notes.



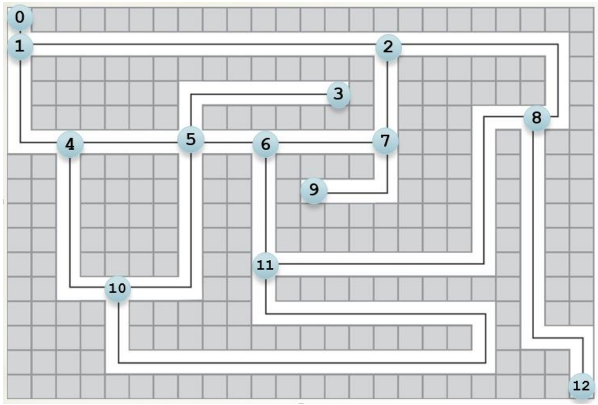

```
6
d
0 1 1.0
0 3 1.0
1 4 1.0
2 4 1.0
2 5 1.0
3 1 1.0
4 3 1.0
5 5 1.0
```



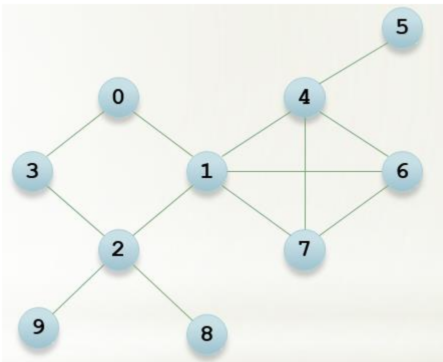

```
10
u
0 1 1.0
0 3 1.0
1 2 1.0
1 4 1.0
1 6 1.0
1 7 1.0
2 3 1.0
2 8 1.0
2 9 1.0
4 5 1.0
4 6 1.0
4 7 1.0
6 7 1.0
```

Program 2 Breadth-First Search

Modify the Breadth-First Search code, given in the lecture notes, for using the new Graph class as described above. Then, using the testing main function in the lecture notes as an example, write a testing main function to test your new BFS algorithm for the following two graphs. You should obtain the same results as shown in the lecture notes.



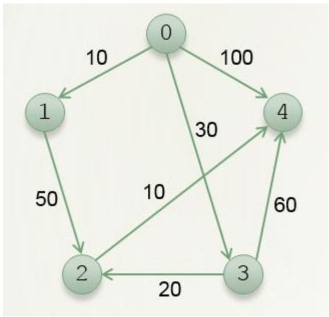

```
13
u
0  1  1.0
1  2  1.0
1  4  1.0
2  7  1.0
2  8  1.0
3  5  1.0
4  5  1.0
4  10 1.0
5  6  1.0
5  10 1.0
6  7  1.0
6  11 1.0
7  9  1.0
8  11 1.0
8  12 1.0
10 11 1.0
```



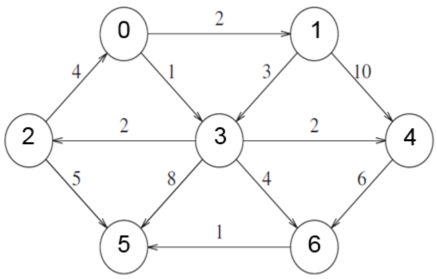

```
10
u
0 1 1.0
0 3 1.0
1 2 1.0
1 4 1.0
1 6 1.0
1 7 1.0
2 3 1.0
2 8 1.0
2 9 1.0
4 5 1.0
4 6 1.0
4 7 1.0
6 7 1.0
```

Program 3 Dijkstra’s Algorithm

Modify the Dijkstra’s algorithm code, given in the lecture notes, for using the new `Graph` class described above. Then, using the testing main function in the lecture notes as an example, write a testing main function to test your new Dijkstra’s algorithm for the following two graphs. You should obtain the same results as shown in the lecture notes.




```
5
d
0 1 10
0 3 30
0 4 100
1 2 50
2 4 10
3 2 20
3 4 60
```




```
7
d
0 1 2
0 3 1
1 3 3
1 4 10
2 0 4
2 5 5
3 2 2
3 4 2
3 5 8
3 6 4
4 6 6
6 5 1
```