Informatics II, Spring 2024, Exercise 12

Publication of exercise: May 20, 2024
Publication of solution: May 27, 2024
Exercise classes: May 27 - May 31, 2024

Learning Goal

- Understand how to build a minimum spanning tree.
- Understand how Dijkstra's Algorithm works.
- Learn how to handle different representations of graph.

Task 1 [Medium]

In this task, you need to generate a Minimum Spanning Tree (MST) for an undirected connected graph. Given an undirected connected graph, there are ***Num_V*** vertices and ***Num_E*** edges in the graph, and all valid edges are provided by ***(from, to, weight)*** pairs. You need to generate the minimum spanning tree and return the total weight of the MST.

Note1: It's assumed that all vertices are represented by integers: $0, 1, 2, \cdots$, and the MST starts from vertex 0 by default.

Note2: You can define the maximum number of vertices and set a large value for the weight of non-existent edges.

```c
#define MAX_VERTICES 20
#define INF 1000
```

Below are the example input and output. The input starts from **Num_V, Num_E** and is followed by edge pairs **from, to, weight**.

Input

```
9, 14
0, 1, 4
0, 7, 8
1, 2, 8
1, 7, 11
2, 3, 7
2, 5, 4
2, 8, 2
3, 4, 9
3, 5, 14
4, 5, 10
5, 6, 2
6, 7, 1
6, 8, 6
7, 8, 7
```

Output

```
The total weight of the MST is 37
```

After obtaining the graph from the provided edge pairs, the adjacency matrix representing the graph should be like this below (x means no edge). You can print it for a sanity check.

```
0 4 x x x x x 8 x
4 0 8 x x x x 11 x
x 8 0 7 x 4 x x 2
x x 7 0 9 14 x x x
x x x 9 0 10 x x x
x x 4 14 10 0 2 x x
x x x x x 2 0 1 6
8 11 x x x x 1 0 7
x x 2 x x x 6 7 0
```

Here is the graph and its minimum spanning tree corresponds to the example. The vertices in the figure are represented by letters for clearer representation ($a- > 1, b- > 2, ...$).

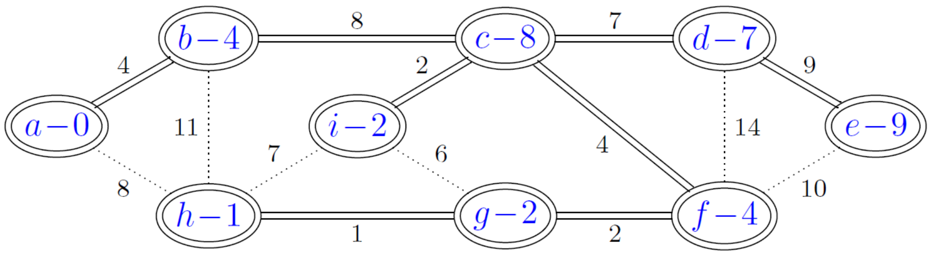

Figure 1: The minimum spanning tree of the example

Task 2 [Medium]

In this task, you need to implement Dijkstra's algorithm for a directed graph with nonnegative edge weights. Given a directed graph, there are ***Num_V*** vertices and ***Num_E*** directed edges in the graph, and all valid edges are provided by ***(from, to, weight)*** pairs. You need to find the shortest path from the source vertex to all other vertices and show the distance.

Note1: It's assumed that all vertices are represented by integers $0, 1, 2, \cdots$, and the source vertex is 0 by default.

Note2: You can define the maximum number of vertices and set a large value for the weight of non-existent edges.

```
#define MAX_VERTICES 20
#define INF 1000
```

Below are the example input and output. The input starts from **Num_V, Num_E** and is followed by edge pairs **from, to, weight** in arbitrary order.

The output shows the minimum distance from the source vertex to other vertices. You also need to show which vertex is unreachable because the graph is not guaranteed to be connected.

Input

```
5, 10
0, 1, 10
0, 2, 5
1, 2, 2
1, 3, 1
2, 1, 3
2, 3, 9
2, 4, 2
3, 4, 4
4, 0, 7
4, 3, 6
```

Output

```
Shortest distances from vertex 0:
Vertex 0: 0
Vertex 1: 8
Vertex 2: 5
Vertex 3: 9
Vertex 4: 7
```

Here is the corresponding graph. For a clearer representation of the vertices and the corresponding distance, the vertices in the figure are represented by letters.

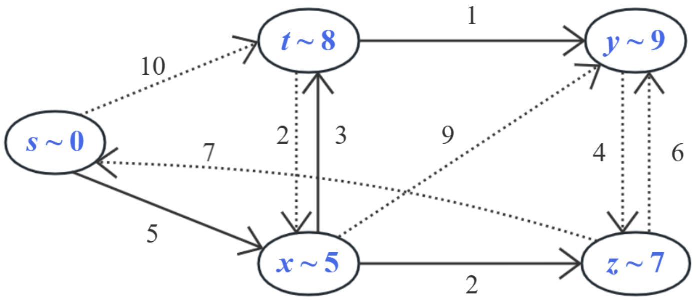

Figure 2: Graph and Results of the algorithm

Task 3 [Hard]

In this task, you will implement Shortest Path Faster Algorithm (SPFA), which is a faster version of Bellman Ford Algorithm.

In Bellman Ford Algorithm, we need to relax every edge in each iteration; therefore, the time complexity of Bellman Ford Algorithm is $\mathcal{O}(|V| \cdot |E|)$. Howevere, there are a lot of unnecessary relaxations in each iteration. In SPFA, we **perform relaxation for egdes start from one vertex in each iteration and put the destinaton vertex into the exploring list only if we update its distance**.

The average time complexity of SPFA is $\mathcal{O}(|E|)$ on random graphs, and the worst-case time complexity is $\Omega(|V| \cdot |E|)$, which is equal to that of Bellman-Ford algorithm. What's more, SPFA can accept negative weights as input and detect the negative weight cycle.

For your implementation, you can use a BFS-based (use queue) or a DFS-based (recursive) method. You can find a more detailed description of SPFA and the pesudo code in Wiki page.

Moreover, how could we check if there is a negative weight cycle?

- For DFS-based method: There is a negative weight cycle when the algorithm tries to push a vertex that already exists in the stack.
- For BFS-based method: A vertex can only be updated by a maximum of N-1 neighbor vertices; a negative weight cycle occurs if a vertex enters the queue more than N-1 times.

The problem description is mainly the same as in Task 2, except that the graph is a connected graph with negative weights. You need to show the minimum distance from the source vertex to all other vertices or show that there is a negative weight cycle.

Note1: Whether the SPFA can detect a negative weight cycle is source-vertex-dependent.
Note2: You can define the maximum number of vertices and set a large value for the weight of non-existent edges.
Note3: We provide an implementation of the array-simulated adjacency list. Use it for faster searching.

```c
#define MAX_VERTICES 20
#define MAX_EDGES 1000

struct Edge {
    int dest; // destination of the edge, initialize to -1
    int weight;
};

struct Graph {
    int num_vertices;
    // idx_1: current vertex
    // idx_2: index of edges start from current vertex
    struct Edge array[MAX_VERTICES][MAX_VERTICES];
};

// how to iterate the edges
int edge_idx = 0;
while(graph->array[u][edge_idx].dest != -1) {
    int v = graph->array[u][edge_idx].dest;
    ...
    edge_idx++;
}
```

Below are the example input and output. The original input corresponding to the graph in Figure 3. If you modify the input data according to the comment, then it becomes the graph in Figure 4.

The output shows the minimum distance from the source vertex to other vertices. If there is a negative weight cycle, show a message instead.

Input

```
// set edge (3->2, 5) to (2->3, -5) to test negative weight cycle
5, 8
0, 1, -1
0, 2, 4
1, 2, 3
1, 3, 3
1, 4, 2
3, 1, 1
3, 2, 5
4, 3, -3
```

Output

```
Shortest distances from vertex 0:
Vertex 0: 0
Vertex 1: -1
Vertex 2: 2
Vertex 3: -2
Vertex 4: 1
OR - Graph contains a negative weight cycle
```

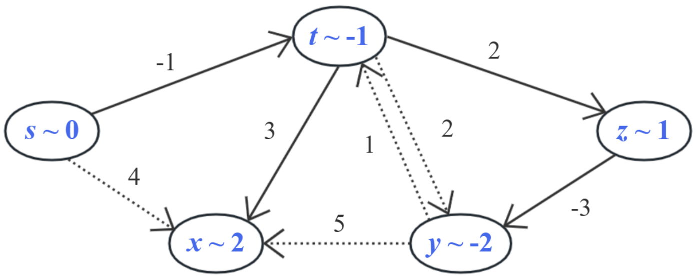

Figure 3: Graph and Results of the algorithm

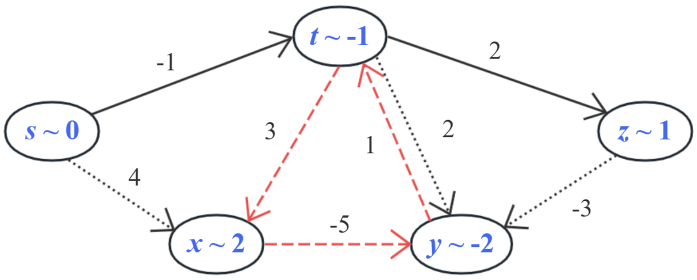

Figure 4: Graph and Results of the algorithm, with negative weight cycle