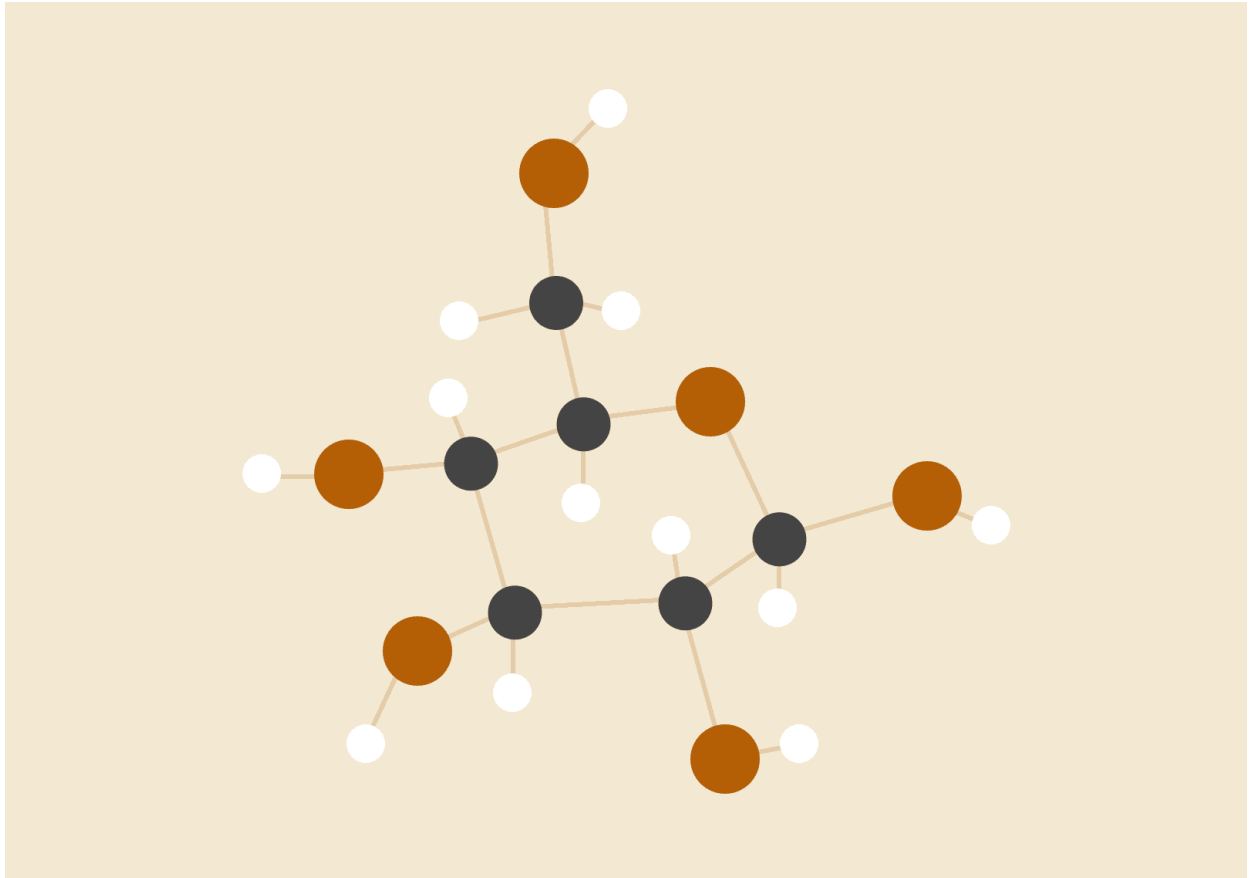


Computer Networks Lab 4



Alisetti Sai Vamsi

17.09.2021

111801002

1. Dijkstra's Algorithm:

Initialization:

Class PriorityQueue: This class maintains a python dictionary which is used as a priority queue. This priority queue is used to hold <key: value> pairs of <vertex: distance>. The priority is on the distance value, i.e the less the distance value the more priority it gets. This queue supports the functions like put(id, distance), get(id), empty(), extract_min().

Class Vertex: This class acts as a structure for each vertex, which stores some essential data like vertex id, distance, parent, python dictionary of adjacent vertices with <key: value> pairs as <vertex id: weight>, visited flag. The distance is initially set to the maximum integer size in python. Vertex(id) creates a vertex with vertex id set to id.

Class Graph: This class provides the graph structure which is implemented through an adjacency map. The graph structure maintains a dictionary of vertices with <key: value> pairs as <vertex id: Vertex(id)>. An edge from a vertex with id u is created by referencing the dictionary with the vertex id “u” and get the corresponding Vertex object. Now this vertex object stores a dictionary of all the adjacent vertices to it. So to create the edge we have to enter the vertex id of the other end of the edge in this adjacent vertices dictionary as key and the value being the weight of the edge. It also supports other functions.

Algorithm:

My algorithm follows these steps which is implemented as a static method in the Graph Class:

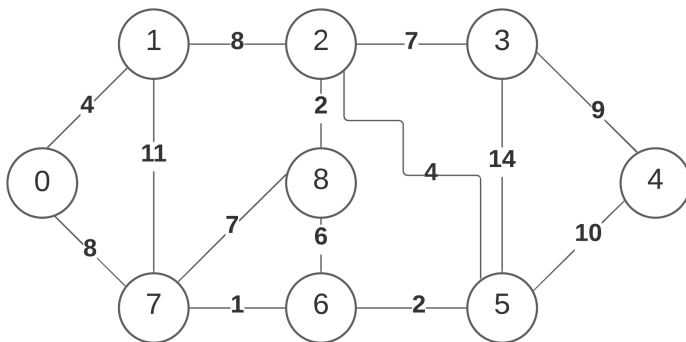
1. Build the graph from the given input
2. Initialize
 - a. Dictionary distance_map = {}
 - b. PriorityQueue queue
 - c. Set the distance of the source vertex to be 0

3. Populate the priority queue with all the vertices and their corresponding distance values which are stored in their structure.
4. While the queue is not empty
 - a. `current_vertex_id, current_distance = Extract the <key: value> pair i.e <vertex id: distance> from the queue with the minimum distance.`
 - b. `current_vertex = graph.vertexSet[current_vertex_id] // returns the corresponding vertex object`
 - i. Set the `current_vertex`'s distance field to `current_distance`
 - ii. Set the current vertex's visited flag to True
 - c. Append the extracted `<vertex: distance>` pair into `distance_map`
 - d. Iterate through each neighbor of current vertex
 - i. Let the neighbor vertex be `v`
 - ii. Current distance of `v` from source vertex = `queue.get(v)`
 - iii. Expected distance = `get_edge_weight(current vertex, v) + current_vertex.distance`
 - iv. If `v` is not visited and current distance of `v` from source > expected distance
 1. `New_distance = expected distance`
 2. Put the new distance in the queue corresponding to the neighbour vertex `v`.
 3. Update the neighbour vertex `v`'s parent to `current_vertex`.
 - e. Return `distance_map`
5. Finding the shortest distance (the `shortest_path` function takes an argument which is the destination vertex object):
 - a. Create an empty list and store the destination vertex id in it.
 - i. While the parent field in the vertex object is not None
 1. `parent_vertex = graph.vertexSet[parent field]`
 2. Append `parent_vertex.id` into the list
 3. `parent_field = parent_vertex.parent`
 - b. Reverse the list and convert it into space separated strings
 - c. Return this string.

Input Format:

Input format is (u, v, w) which is (vertex1, vertex2, weight). An assumption made is that the input will always be an undirected graph and because of this the build_graph function adds both (u, v, w) and (v, u, w) to the graph.

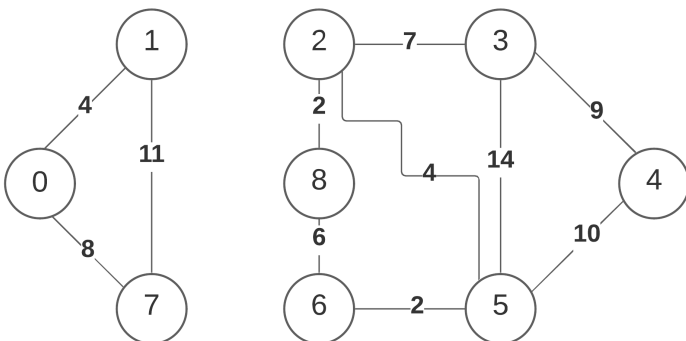
Input graph (input1.txt):



```
Computer Networks/Lab/Lab4 via v3.7.12 (Algorithms)
> python Q1.py 9 0 input1.txt
```

Vertex	Cost	Path
0	0	0
1	4	0 1
2	12	0 1 2
3	19	0 1 2 3
4	21	0 7 6 5 4
5	11	0 7 6 5
6	9	0 7 6
7	8	0 7
8	14	0 1 2 8

Input graph (input2.txt):



```
Computer Networks/Lab/Lab4 via v3.7.12 (Algorithms)
> python Q1.py 9 0 input2.txt
```

Vertex	Cost	Path
0	0	0
1	4	0 1
2	infinity	No Route
3	infinity	No Route
4	infinity	No Route
5	infinity	No Route
6	infinity	No Route
7	8	0 7
8	infinity	No Route

2. Distance Vector Algorithm

Initialization:

Class Vertex: This class acts as a structure for each vertex, which stores some essential data like vertex id, distance, parent, python dictionary of adjacent vertices with <key: value> pairs as <vertex id: weight>, visited flag. The distance is initially set to the maximum integer size in python. Vertex(id) creates a vertex with vertex id set to id.

Class Graph: This class provides the graph structure which is implemented through an adjacency map. The graph structure maintains a dictionary of vertices with <key: value> pairs as <vertex id: Vertex(id)>. An edge from a vertex with id u is created by referencing the dictionary with the vertex id “u” and get the corresponding Vertex object. Now this vertex object stores a dictionary of all the adjacent vertices to it. So to create the edge we have to enter the vertex id of the other end of the edge in this adjacent vertices dictionary as key and the value being the weight of the edge. It also supports other functions.

Algorithm:

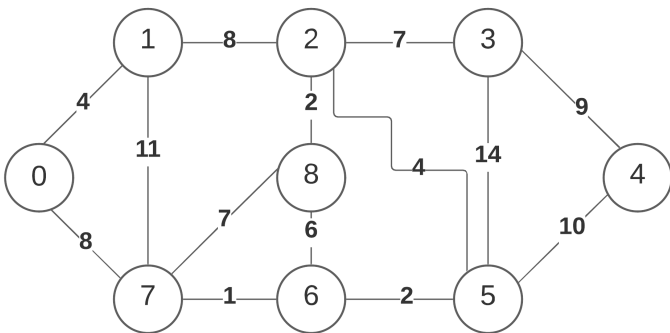
My algorithm follows these steps which is implemented as a static method in the Graph Class:

1. Build the graph from the given input
2. Initialize distance_vector as 2D numpy array of size n x n where n is the number of nodes parsed from the command line arguments.
 - a. Each row of distance_vector variable indicates the distance vector corresponding to the particular vertex i.e 0th index of the distance_vector variable indicates the n length distance vector corresponding to vertex 0..
 - b. Initialize the 2D arrays with zeros.
 - c. Go through all the vertices and fill the distance vector of each vertex with the weights of its corresponding neighbours.
 - d. Initialize next_hop_nodes as similar sized 2D arrays and fill the next hop


nodes arrays with their neighbouring nodes.

3. Initialize a `prev_distance_vector` with a similar sized 2D array
4. While `prev_distance_vector` is not equal to `current_distance_vector`
 - a. `vertex_set = get_vertices()`
 - b. Shuffle `vertex_set`
 - c. For each vertex `u` in `vertex_set`
 1. `current_distance_vector = distance_vector[u.id]`
 2. `neighbours = find_neighbours_of_vertex(u)`
 3. Randomly shuffle the list of neighbours.
 4. For each vertex `v` in `neighbours`
 - a. `edge_weight = get_edge_weight(u,v)`
 - b. `neighbour_distance_vector = distance_vector[v.id]`
 - c. for `j` in `length(current_distance_vector)`:
 - i. If `current_distance_vector[j] > neighbour_distance_vector[j] + edge_weight`:
 1. `current_distance_vector[j] = neighbour_distance_vector[j] + edge_weight`
 2. `next_hop_nodes[j] = v.id`
5. Return `distance_vector`

Input Graph (input1.txt):



5

```
Computer Networks/Lab/Lab4 via  v3.7.12 (Algorithms)
```

```
> python Q2.py 9 input1.txt
```



```
Vertex 0
```

Vertex	Distance	Vector	Hop	Node
0	0		0	
1	4		1	
2	12		1	
3	19		1	
4	21		7	
5	11		7	
6	9		7	
7	8		7	
8	14		1	


```
Vertex 1
```

Vertex	Distance	Vector	Hop	Node
0	4		0	
1	0		1	
2	8		2	
3	15		2	
4	22		2	
5	12		2	
6	12		7	
7	11		7	
8	10		2	

Vertex 2		
Vertex	Distance Vector	Hop Node
0	12	1
1	8	1
2	0	2
3	7	3
4	14	5
5	4	5
6	6	5
7	7	5
8	2	8

Vertex 3		
Vertex	Distance Vector	Hop Node
0	19	2
1	15	2
2	7	2
3	0	3
4	9	4
5	11	2
6	13	2
7	14	2
8	9	2

Vertex 4		
Vertex	Distance Vector	Hop Node
0	21	5
1	22	5
2	14	5
3	9	3
4	0	4
5	10	5
6	12	5
7	13	5
8	16	5

Vertex 5		
Vertex	Distance Vector	Hop Node
0	11	6
1	12	2
2	4	2
3	11	2
4	10	4
5	0	5
6	2	6
7	3	6
8	6	2

Vertex 6		
Vertex	Distance Vector	Hop Node
0	9	7
1	12	7
2	6	5
3	13	5
4	12	5
5	2	5
6	0	6
7	1	7
8	6	8

Vertex 7		
Vertex	Distance Vector	Hop Node
0	8	0
1	11	1
2	7	6
3	14	6
4	13	6
5	3	6
6	1	6
7	0	7
8	7	8

Vertex 8		
Vertex	Distance Vector	Hop Node
0	14	2
1	10	2
2	2	2
3	9	2
4	16	2
5	6	2
6	6	6
7	7	7
8	0	8