CS512: Design and Analysis of Algorithms

Assignment #2

Name: Kunal Wanikar

Roll No: 204101070

Question 1)

Given: Undirected graph $G = (V, E)$ and a source vertex *'s'* which belongs to G

To find: The value which the num[v] array holds at the end of execution of the algorithm and the time complexity of the algorithm.

Solution: Let us consider a graph $G = (V, E)$ with two vertices $u, v \in V$. Now we can see that we enqueue the source vertex *s* into the empty queue. After this the algorithm enters while loop and enqueues and dequeues several vertices. Assume that *u* is the vertex, which is being explored before *v*, hence *u* should be enqueued in the queue before *v*.

Statement: Cost array hold the length of the shortest path from the source vertex to that vertex. Num array stores the number of ways in which we can reach the vertex from the source having minimum path length.

Working of the Algorithm:

- We are given input an undirected graph $G = (V, E)$ and a source vertex $s \in V$ and we need to output for every vertex *v*, the value of num[v]. We divide the algorithm in two parts.
  - **A. Initialization:** The num array and the cost array is set to zero and infinity respectively. Now the num and cost of the source vertex is initialized to 1 and 0 respectively. An empty queue is created, and the source vertex is inserted into the queue.
  - **B. Updation:**
    - **a.** In this part the algorithm checks if the queue is not empty. If the condition is true, then deletes the front element from the queue and explores its adjacent vertices.
    - **b.** Now for the element, which is deleted, it checks all the vertices directly connected to it via an edge. If the cost of that vertex $cost[v] = \infty$ then we say that the vertex is not discovered. Now we mark the vertex as discovered and set the cost of that vertex as cost of exploring vertex (parent) plus 1 $(cost[v] = cost[u] + 1)$.
    - **c.** This is how the cost array is being updated and if we move to the next vertex then the cost value gets added by one and we can see that the cost

stores the minimum distance of the vertex from the source. In the next step we store the num value of the exploring node (parent) into the index of num array for that vertex. Now we insert that unexplored vertex into the queue.

d. If the vertex is already being explored i.e. *cost[v]* $\neq \infty$ then the algorithm will go into the else condition and check whether the given vertex is the child descendent of the exploring vertex. If yes that means the child vertex can be reached from the source with another path. Now we add the num values of the 2 vertices (parent and child) and store in the index of num array of the child vertex.

e. Hence, we can conclude that the num array stores the number of possible ways we can reach a particular vertex from the source with minimum length of path.

- The above gives the exact working of the algorithm and we observe that the num array is being updated by the number of times the parent is visiting the child. The num[v] array stores the number of ways we can visit the vertex v from the source with minimum path length.
- The algorithm depicts the working of **Breadth First Search algorithm** with slight modifications as we can see that whenever a new node is explored it is being added to the next layer of the parent node and then inserted into the queue.
- **Time Complexity analysis:** We implement the graph using adjacency list hence for inserting vertices and edges into the adjacency list we need **O(V+E)** time. Now observing the algorithm, we can find that for every vertex which is inserted in the queue, we must traverse the edges connected to that vertex. This will take O(V+E) time. Although we can optimize this time by considering two cases.

i) **Graph is connected:** If we know that the input graph is always connected then time complexity will be **O(E).**

- **Best Case:** When the graph is a tree with V vertices then number of edges will be minimum and equal to V-1. From this we can get **O(E) = O(V).**
- **Worst Case:** When the graph is a complete graph with V vertices then number of edges will be maximum and equal to V(V-1)/2. From this we can get **O(E) = O(V^2).**

Hence, if the graph is connected then **O(E)** will suffice the time complexity.

ii) **Graph is either connected or disconnected:** If we don't know that the input graph is connected or not then the time complexity will be **O(V+E).**

- **Best Case:** When the graph is disconnected graph with zero vertices. Then to create the adjacency list representation we need O(V) time. Hence the algorithm time complexity will be **O(V).** Another best case can be that the graph is a tree with V vertices and V-1 edges.

- **Worst Case:** When the graph is a complete graph i.e. dense graph, at that time the number of edges will be V(V-1)/2. From this we can get **O(E) = O(V^2).**

Hence if it is not given that the graph is connected or not then time complexity of the algorithm will be **O(V+E).**

Conclusion: The given algorithm is the **Breadth First Search Algorithm** with slight modifications of adding an array num[v]. The num array stores the **number of ways** we can reach vertex v from the source with **minimum path length**. The time complexity of the algorithm for connected graph will be **O(E).**

Question 2)

Given: Undirected graph $G = (V, E)$ and a source vertex *'s'* which belongs to G

To find: The value which the *IsGood[v]* array holds at the end of execution of the algorithm and the time complexity of the algorithm.

Solution: Let us consider a graph $G = (V, E)$ with edge weights $l_e > 0$ for all $e \in E$. Now we can see that we enqueue the source vertex *s* into the empty queue. After this the algorithm enters while loop and enqueues and dequeues several vertices. Assume that *u* is the vertex, which is being explored before *v*, hence *u* should be enqueued in the queue before *v*.

Working of the Algorithm:

- We are given input an undirected graph $G = (V, E)$ and a source vertex $s \in V$ and we need to output for every vertex *v*, the Boolean value of IsGood[v]. We divide the algorithm in two parts.
  - **A. Initialization:** We initialize each element of cost array and IsGood array as ∞ and TRUE respectively. Now make the cost of the source i.e. $cost[s] = 0$. Now we create a priority queue using the min heap data structure and insert the vertices with the cost of that vertex as in the cost array. In the priority queue the *cost[v]* is the key value.
  - **B. Updation:**
    - **a.** In this part the algorithm checks if the priority queue is not empty. If true, then deletes the min key value from the queue and explores that node i.e. the node with shortest distance from the source. Now we check for each edge which is leaving the exploring vertex (say *u*).
    - **b.** If the cost of the visited vertex is greater than the cost of vertex u from source plus weight of the edge $(cost[v] > cost[u] + l_e)$, then we relax the edge from u to v with the less weight i.e. $cost[u] + l_e$ and store it in the cost array. Hence, we can see that the cost array stores the least possible weighted path length from the source to that vertex.

c. After this step the *DecreaseKey* is called which changes the priority of the vertex *v* to the updated value of *cost[v]*, now the *IsGood[v]* is updated to the value of *isGood[u]* because if the vertex *u* is not in the shortest path then the IsGood value of vertex *v* is updated to true/false depending on the value of *IsGood[u].*
d. Otherwise the *IsGood[v]* value is kept to false that means the vertex *v* is included in the shortest path from source.

- The above gives the exact working of the algorithm and the IsGood array stores false if the vertex is being considered in the shortest path from the source. Although this does not hold true if there exists a pendant vertex in the graph.
- The algorithm depicts the working of **Dijkstra's Algorithm** with slight modifications as we see that the algorithm chooses the smallest cost from the source and explores that node completely till all the edges are relaxed (if possible).
- **Time Complexity Analysis:** We have implemented the graph using adjacency list hence for insertion of vertices and edges we need **O(V+E)** time. To insert vertices into the priority queue it takes **O(V)** time. For delete min it takes **O(logV)** time multiplied by the number of vertices i.e. **O (VlogV).** For each edge it takes **O(logV)** time for decrease key. This means **O(ElogV)** for all the E edges. So, the overall time complexity will be **O (V+ E + V+ VlogV + ElogV)** = O (VlogV + ElogV) = **O((V+E) logV)**. If we consider graph is connected, then in the best-case O(E) = O(V) and in the worst-case **O(E) = O(V^2).** Hence total time complexity will be **O(ElogV).**

Conclusion: The given algorithm is the **Dijkstra's algorithm** with slight modifications of adding and array *IsGood[v]*. The isGood array marks the vertices as false which are lying on the shortest path. The time complexity of the algorithm for connected graph will be **O(ElogV).**