Name: Param Somane **Access ID:** pss5256

Problem 1 **Points:**

1. We run Dijkstra's algorithm on the given graph. We denote NULL by ϕ. We utilize three data structures for this algorithm, namely the priority queue PQ, $dist[]$ array, $prev[]$ array, and $S := \{v \in V \mid distance(s,v)$ has been found$\}$.

dist *prev* *PQ minimum*

Initialization

s	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	∞

s	a	b	c	d	e	f
s	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

s – $distance(s,s) = 0$

Iteration 1
Delete s from PQ and $S = \{s\}$

s	a	b	c	d	e	f
0	6	8	∞	∞	∞	∞

s	a	b	c	d	e	f
s	*s*	*s*	ϕ	ϕ	ϕ	ϕ

a – $distance(s,a) = 6$

Iteration 2
Delete a from PQ and $S = \{s,a\}$

s	a	b	c	d	e	f
0	6	8	11	26	∞	∞

s	a	b	c	d	e	f
s	*s*	*s*	*a*	*a*	ϕ	ϕ

b – $distance(s,b) = 8$

Iteration 3
Delete b from PQ and $S = \{s,a,b\}$

s	a	b	c	d	e	f
0	6	8	11	26	∞	25

s	a	b	c	d	e	f
s	*s*	*s*	*a*	*a*	ϕ	*b*

c – $distance(s,c) = 11$

Iteration 4
Delete c from PQ and $S = \{s,a,b,c\}$

s	a	b	c	d	e	f
0	6	8	11	20	12	25

s	a	b	c	d	e	f
s	*s*	*s*	*a*	*c*	*c*	*b*

e – $distance(s,e) = 12$

Iteration 5
Delete e from PQ and $S = \{s,a,b,c,e\}$

s	a	b	c	d	e	f
0	6	8	11	19	12	16

s	a	b	c	d	e	f
s	*s*	*s*	*a*	*e*	*c*	*e*

f – $distance(s,f) = 16$

Iteration 6
Delete f from PQ and $S = \{s, a, b, c, e, f\}$

s	a	b	c	d	e	f
0	6	8	11	19	12	16

s	a	b	c	d	e	f
s	*s*	*s*	*a*	*e*	*c*	*e*

$d - distance(s, f) = 19$

Iteration 7
Delete d from PQ and $S = \{s, a, b, c, e, f, d\}$

s	a	b	c	d	e	f
0	6	8	11	19	12	16

s	a	b	c	d	e	f
s	*s*	*s*	*a*	*e*	*c*	*e*

ϕ

2. Hence we obtain $prev = [s, s, s, a, e, c, e]$ which yields the shortest-path tree for the graph as depicted in Figure 1.

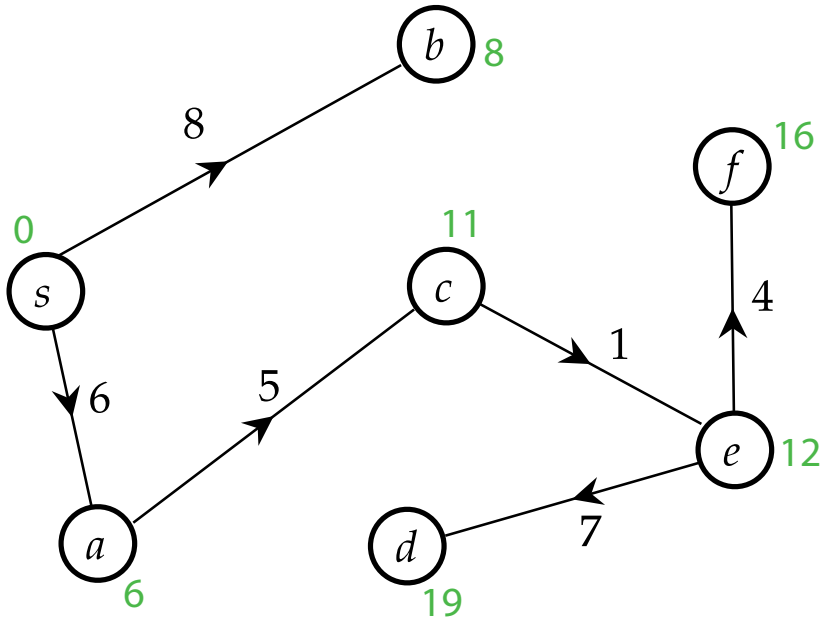

Figure 1: Shortest-path tree of the given graph.

Problem 2 **Points:**

1. Observe that the given graph G does not have a negative cycle but there is, indeed, a cycle of length zero, namely $c \to d \to e \to c$. We maintain the dynamic programming table $dist(k, v)$ and another table to store the corresponding $pre(k, v)$ values for $v \in V$ and $0 \leq k \leq |V| - 1 = 5$. The resulting dynamic programming tables are given below in Table 1 and Table 2. Note that we get $dist = [0, 4, 7, 15, 8, 1]$ and $prev = [s, s, e, c, f, b]$.

2. The shortest-path tree for the G is portrayed in Figure 2 below.

	s	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	0	4	9	∞	∞	∞
2	0	4	9	17	∞	1
3	0	4	9	17	8	1
4	0	4	7	17	8	1
5	0	4	7	15	8	1

Table 1: $dist(k,v)$

	s	b	c	d	e	f
0	s	ϕ	ϕ	ϕ	ϕ	ϕ
1	s	s	s	ϕ	ϕ	ϕ
2	s	s	s	c	ϕ	b
3	s	s	s	c	f	b
4	s	s	e	c	f	b
5	s	s	e	c	f	b

Table 2: $prev(k,v)$

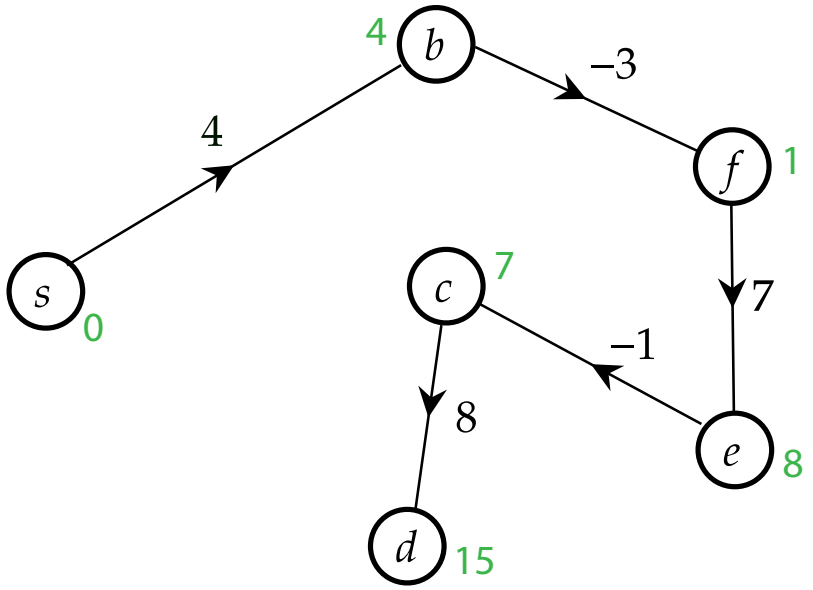

Figure 2: Shortest-path tree of G.

Problem 3 **Points:**

1. Let $l\colon V \to \{0,1,\cdots,9\}$ be the given label map. Observe that if there is a path $s = v_0 \to v_1 \to \cdots \to v_{n-1} \to v_n = t$ satisfying the 465465... pattern, then $l(v_i) = \begin{cases} 4 & if\ i \equiv 0\ (mod\ 3) \\ 6 & if\ i \equiv 1\ (mod\ 3) \\ 5 & if\ i \equiv 2\ (mod\ 3) \end{cases}$ for all $0 \leq i \leq n$. In particular, $l(s) = l(v_0) = 4$ for such a path. Moreover, a path satisfying this pattern in G corresponds to a unique path in the digraph $G' = (V', E')$, constructed by letting $V' = V$ and $E' = \{e = (u,v) \in E \mid (l(u), l(v)) \in \{(4,6), (6,5), (5,4)\}\}$, the new path merely being the exact same path projected onto G'. Ergo, we first construct G' and then verify that $l(s) = 4$ and t is reachable from s in this new graph. Note that this condition is sufficient because if we find a path from s to t in G' satisfying $l(s) = 4$, then the inclusion map from $G' \to G$ gives us the required path in G satisfying the 465465... pattern by construction of G'.

Algorithm PATH-465 ($G = (V,E)$, $l(v)$ $(\forall v \in V)$, $s,t \in V$)
  if $l(s) \neq 4$: return false;
  init $G' = (V',E')$ with $V' = V$ and $E' = \phi$;
  for each edge $(u,v) \in E$
    if $(l(u),l(v)) \in \{(4,6), (6,5), (5,4)\}\}$: add (u,v) to E';
  end for;
  init $visited[\,] = [0] * |V|$
  explore(G',s);
  if $visited[t] = 1$: return true;
  return false;
end algorithm;

Running time: Construction of G' takes $O(|V|+|E|)$ time because $V' = V$ and forming E' requires iterating through all edges in E. Moreover, explore(G',s) runs in $O(|V'|+|E'|) = O(|V|+|E|)$ time as $|E'| \leq |E|$. Thus, the running time of the algorithm is $O(|V|+|E|)$.

2. The crux of this problem is to perform a breadth first search on G while examining all walks originating from s that follow the 465465... pattern along the edges, until either we find a walk that passes through t or we run out of possible paths. The only issue arises when there is a $3n$-cycle in G of the form $a \to b \to c \to \cdots \to c' \to a$ with $\ell(a,b) = 4$, $\ell(b,c) = 6, \cdots, \ell(c',a) = 5$ (for example, $c = c'$ when $n = 1$) which is reachable from s and there does not exist a walk from s to t satisfying the 465465... pattern. In this case, we need to prevent our algorithm from getting stuck iterating in this cycle. This can be done by making the remarkable observation that if a walk from s to t follows the 465465... pattern, then although it may contain repeated vertices, it cannot contain repeated edges. This is because if an edge e occurs twice in such a walk, say at v_i and at v_j, then the subwalk $v_{i+1} \to \cdots \to v_j$ between those two occurrences can be omitted and we will still maintain the 465465... pattern. Hence, all we need to do is keep track of the traversed edges and prevent our algorithm from traversing over them again. Note that we use the formula for $l(v_i)$ given in part 1 to check if our walk satisfies the required pattern and we store vertices v in the queue for BFS along with the label of the edge (u,v), while examining which we had inserted v into the queue. By finding the 'junction' in the queue where the value of these labels changes to the next term in the sequence 465465..., we can know when to proceed to the next level of the graph and thereby the expected label of the following vertex.

Running time: Since the algorithm below is a modified version of BFS, with the modifications running in constant time, and as every edge is traversed exactly once due to the $traversed[\,]$ array, its running time is $O(|V|+|E|)$.

```
Algorithm WALK-465 (G = (V,E), l(e) (∀e ∈ E), s,t ∈ V)
    init pattern[0···2] = {4,6,5};
    init level = 0;
    init traversed[ ] = [0] * |E|;
    init an empty queue Q;
    insert(Q, [s,0]);
    while empty(Q) = false
        a = find-earliest(Q);
        delete-earliest(Q);
        init u = a[0]; //earliest vertex
        init label = a[1]; //label of edge associated with vertex u
        for each edge (u,v) ∈ E
            if traversed[(u,v)] = 1: continue;
            if l(u,v) = pattern[level (mod 3)]
                if v = t: return true;
                insert(Q, [v, l(u,v)])
            end if;
            traversed[(u,v)] = 1;
        end for;
        if empty(Q) = false
            init b = find-earliest(Q);
            if label ≠ b[1]: level++
        end if;
    end while;
    return false;
end algorithm;
```

Note: Alternatively, we can attempt this problem by constructing the line graph (also called the conjugate graph) G' of G by interchanging the roles of edges and vertices. More precisely, $G' = (V', E')$, where $V' = E$ and $E' = \{(e, e') \mid e = (u,v)$ and $e' = (v,w)$ for some $(u,v), (v,w) \in E$ that share a common vertex $v \in V\}$. Then, by the construction of G', a walk $W : s = v_0 \to v_1 \to \cdots \to v_n = t$ from s to t, where the v_i's are not necessarily distinct, exists in G if and only if there is a path $P : (s, v_1) \to (v_1, v_2) \to \cdots \to (v_{n-1}, t)$ in G'. Also, such a path P is well-defined because although the vertices in W may repeat, we can suppose that the edges in W are distinct because otherwise we can omit the subwalk between two consecutive occurrences of the same edge. Lastly, W will follow the 465465... pattern along its edges in G if and only if P follows the 465465... pattern along its vertices in G', thereby reducing this problem to calling the algorithm PATH-465$(G' = (V', E'), l(e)\ (\forall e \in E), (s,x), (y,t) \in E)$ for all possible out-edges $(s,x) \in E$ with $l(s,x) = 4$ and in-edges $(y,t) \in E$. Although this algorithm works, its running time is not $O(|V| + |E|)$. Constructing V' takes $O(|E|)$ time. For constructing E', we iterate over all the edges

$(u,v) \in E$ and for each edge, we examine the array corresponding to v in the adjacency list and add all edges of the form $((u,v),(v,w))$ to E' for every (v,w) present in E. This will take $O(|E|^2)$ if for each edge (u,v), we iterate over all the edges and select the out-edges from v. This suggests that the construction of G' should be done in $O(|E|^2)$ time. Moreover, if we look at the worst case scenario of an $n-$clique (with directed edges both ways between any two vertices), then we have $|V| = n$ and so $|E| = \binom{n}{2} \cdot 2! = n(n-1)$; thus, by looking at all the possible permutations of $((u,v),(v,w))$, where u and w can repeat, we will have $n(n-1)^2$ edges in E'. Adding each such edge to E' takes constant time and so adding all of them will take at least $O(n(n-1)^2) = O(n^3)$ time regardless of how we add them to E'. But $O(|E|^2) = O(n^4)$ and $O(|V|+|E|) = O(n^2)$. This suggests that an algorithm more efficient than $O(|E|^2)$ might exist but a $O(|V|+|E|)$ time algorithm is likely not possible.

Problem 4 **Points:**

Recall that the optimal substructure property of directed graphs holds for graphs with possibly negative edge lengths. Let $a \in V$ be fixed and $u,v \in V$ be arbitrary. Then by the optimal substructure property, if P: $u \to \cdots \to a \to \cdots \to v$ is the shortest path from u to v passing through a, then $u \to \cdots \to a$ and $a \to \cdots \to v$ must be the shortest paths from u to a and from a to v respectively, because if there were a shorter path from u to a or from a to v, then by replacing this new subpath in P, we would obtain a path from u to v passing through a that is shorter than P, which is absurd. Hence, $distance_a(u,v) = distance(u,a) + distance(a,v)$, where $distance(x,y)$ is the length of the shortest path from x to y. We first call DP-shortest-path(G, $l(e)$ $(\forall e \in E)$, $a \in V$) which yields the shortest distance $distance(a,v)$ from a to v for all $v \in V$. Recall that the DPS-shortest-path algorithm runs in $O(|V| \cdot |E|)$ time. Next, we compute G_R, the reverse of graph G which takes $O(|V|+|E|)$ time. Note that if $(x,y) \in E$, then $(y,x) \in E_R$ and $l(x,y) = l_R(y,x)$. Hence, $u \to v_1 \to \cdots \to v_{n-1} \to a$ is the shortest path from u to a in G if and only if $a \to v_{n-1} \to \cdots \to v_1 \to u$ is the shortest path from a to u in G_R because given any path $P = (e_1, e_2, \cdots, e_m)$ in G, the corresponding path in G_R is uniquely given by $P_R = (e'_m, \cdots, e'_2, e'_1)$ (where $e = (a,b) \in E \iff e' = (b,a) \in E_R$) and so the lengths of the two paths are the same because

$$\sum_{e_i \in P} l(e_i) = l(e_1) + l(e_2) + \cdots + l(e_m) = l_R(e'_m) + \cdots + l_R(e'_2) + l_R(e'_1) = \sum_{e'_i \in P_R} l_R(e'_i)$$

Therefore, $distance(u,a) = distance(a,u)|_{G_R}$ for all $u \in V$. Thus, we now call the algorithm DP-shortest-path(G_R, $l(e')$ $(\forall e' \in E_R)$, $a \in V_R$) which yields $distance(a,u)|_{G_R}$ for all $u \in V$, which in turn gives us $distance(u,a)$ for all $u \in V$, again in $O(|V_R| \cdot |E_R|) = O(|V| \cdot |E|)$ time since $|V| = |V_R|$ and $|E| = |E_R|$. Lastly, we iterate through all the $|V|^2$ pairs (u,v) (note that u and v can be possibly equal and one of them can be possibly equal to a) and compute $distance_a(u,v)$ as $distance(u,a) + distance(a,v)$. This takes $O(|V|^2)$ time. Subsequently, the time complexity of this algorithm is $O(2 \cdot |V| \cdot |E| + |V| + |E| + |V|^2) = O(|V| \cdot (|V| + |E|))$ as $|V|, |E| << |V|^2, |V| \cdot |E|$.

Problem 5 **Points:**

The main idea of this problem is to regard the weight of every vertex as an extension of the lengths of each of its in-edges. To be explicit, given an edge $(u,v) \in E$, the idea is to replace $l(u,v)$ in Dijkstra's algorithm with $l(u,v) + w(v)$. Note that both $l(e)$ and $w(v)$ are positive; thus, $l(e) + w(v)$ is positive and so the invariant of having $distance(s,v) = dist[v]$ once we set $dist[v]$ to a finite value

is maintained. Also note that $dist[s] = 0 + w(s) = 0$. Another way to view this problem is to split every weighted vertex v into unweighted vertices v' and v'', and add an edge (v', v'') to E with $l(v', v'') = w(v)$, such that all in-edges of v connect to v' and all out-edges of v emerge from v''. Then the problem is reduced to running the usual Dijkstra's algorithm on this new graph. Since the length of a path in this modified graph without weighted vertices is equal to the length of the corresponding path in G defined as the sum of lengths of all edges and weights of all vertices in the path, it is clear that this alternate way is essentially equivalent to implementing Dijkstra's algorithm with the weight of each vertex considered as an extension of the lengths of each of its in-edges. The following pseudo-code illustrates the slight modification made to Dijkstra's algorithm.

```
Algorithm Dijkstra-Mod (G = (V,E), l(e) (∀e ∈ E), w(v) (∀v ∈ V), s ∈ V)
    dist[v] = ∞, for any v ∈ V
    init an empty priority queue PQ;
    for each v ∈ V: insert(PQ, v), where the priority of v is ∞;
    dist[s] = 0;
    decrease-key(PQ, s, 0);
    while empty(PQ) = false
        u = find-min(PQ)
        delete-min(PQ)
        for each edge (u, v) ∈ E
            if dist[v] > dist[u] + l(u, v) + w(v)
                dist[v] = dist[u] + l(u, v) + w(v);
                decrease-key(PQ, v, dist[v]);
            end if;
        end for;
    end while;
end algorithm;
```

Running time: Running time of the algorithm is the same as Dijkstra's algorithm since the number of iterations made remains unchanged. So Dijkstra-Mod runs in $O((|V| + |E|) \cdot log|V|)$ time.

Problem 6 **Points:**

In the dynamic programming algorithm for the shortest path problem, observe that when $dist(k, v) > dist(k-1, u) + l(u, v)$, we update $dist(k, v)$ to reflect the current shortest path(s) from s to v using at most k edges; thus, there will be two or more shortest paths from s to v if and only if there are two or more shortest paths from s to u, or equivalently if $multiple[u] = 1$ (this is because we can traverse from u to v only along the edge (u, v) and so the possibility of multiple shortest paths depends on the scenario between s and u). Hence, we need to set $multiple[v]$ to $multiple[u]$, should $dist(k, v)$ be updated. Moreover, if $dist(k, v) = dist(k-1, u) + l(u, v)$ and $prev[v] \neq u$, then we have found at least two distinct shortest paths from s to v, one (class of paths) being the current path(s) $s \to \cdots \to prev[v] \to v$ associated with the previous update of $dist(k, v)$ and another (class) being

the join of the shortest path(s) from s to u and the edge (u,v). By checking that $prev[v] \neq u$, we ensure that we do not count the current shortest path twice. But then this raises the query about what happens when there are multiple shortest paths of the form $s \to \cdots \to u \to v$. This is dealt with in the previous case wherein we set $multiple[v] = multiple[u]$ while updating $dist(k,v)$, so all the shortest paths from s to u carry over to the scenario between s and v. This exhausts all possibilities for shortest paths from s to v. Lastly, our choice of setting $multiple[v] = 0$ for all $v \in V$ at the beginning of the algorithm below is justified because all vertices $v \in V$ are reachable from s and so either $multiple[v] = 0$ (unique shortest path) or $multiple[v] = 1$ (multiple shortest paths). Ergo, the state of the binary array $multiple$ at the end of the algorithm is as desired.

```
Algorithm DP-multi-shortest-path (G = (V,E), l(e) (∀e ∈ E), s ∈ V)
    init a 2D array dist of size |V| × |V|;
    dist[0,s] = 0;
    dist[0,v] = ∞, for all v ∈ V \ {s};
    prev[v] = null, for all v ∈ V;
    multiple[v] = 0, for all v ∈ V;
    for k = 1 to |V| − 1
        for v ∈ V
            dist(k,v) = dist(k−1,v);
            for each edge (u,v) ∈ E
                if dist(k,v) = dist(k−1,u) + l(u,v) and prev[v] ≠ u
                    multiple[v] = 1;
                else if dist(k,v) > dist(k−1,u) + l(u,v)
                    dist(k,v) = dist(k−1,u) + l(u,v);
                    prev[v] = u;
                    multiple[v] = multiple[u];
                end if;
            end for;
        end for;
    end for;
    dist[|V| − 1,v] gives distance(s,v), for each v ∈ V;
end algorithm;
```

Running time: Since we have only made slight modifications to the DP-shortest-path algorithm, and these modifications run in constant time, so the running time of the algorithm is the same as the dynamic programming algorithm for shortest path, namely $O(|V| \cdot |E|)$.

Problem 7 **Points:**

Suppose that $P = (e_1, e_2, \cdots, e_n)$ is a path in G with edges $e_i \in E$. Then the probability that P will not fail (reliability) is given by $Pr(P) := \prod_{1 \leq i \leq n} r(e_i)$ because all the given probabilities are independent. We want to find the most reliable path from s to t, namely the path P from s to t for

which $Pr(P)$ is maximum. However, Dijkstra's algorithm can only work with graphs wherein the lengths of paths are computed additively and the lengths of all edges are positive. Now, observe that maximizing $Pr(P) = \prod_{1 \leq i \leq n} r(e_i)$ is equivalent to maximizing

$$ln(Pr(P)) := ln\left(\prod_{1 \leq i \leq n} r(e_i)\right) = \sum_{i=1}^{n} ln(r(e_i))$$

because the natural logarithm $ln(\cdot)\colon (0,\infty) \to \mathbb{R}$ is an increasing monotonic function. Now, for every $e \in E$, we have $r(e) \in (0,1) \implies ln(r(e)) \in (-\infty, 0) \implies -ln(r(e)) \in (0,\infty)$. Moreover, maximizing $\sum_{i=1}^{n} ln(r(e_i))$ is equivalent to minimizing $\sum_{i=1}^{n} -ln(r(e_i))$. Now, we define a map $\ell\colon E \to (0,\infty)\colon (u,v) \mapsto -ln(r(u,v))$ and ℓ is well-defined because given any $e \in E$, $-ln(r(e)) \in (0,\infty)$. Ergo, we can now call Dijkstra(G, $\ell(e)$ $(\forall e \in E)$, $s \in V$) because all the new edge lengths $\ell(e)$ are positive and the length of a path $P = (e_1, e_2, \cdots, e_n)$ in G is computed additively as $\sum_{1 \leq i \leq n} \ell(e_i)$. This yields the shortest distances from s to all vertices in V and the $pre[\,]$ array, which tells us the path $\gamma : s = v_0 \to v_1 \to \cdots \to v_{n-1} \to v_n = t$ with the minimum length $\sum_{0 \leq i \leq n-1} \ell(v_i, v_{i+1}) = \sum_{0 \leq i \leq n-1} -ln(r(v_i, v_{i+1}))$, and since this corresponds to the maximum value of the product $\prod_{0 \leq i \leq n-1} r(v_i, v_{i+1})$, we conclude that γ is the most reliable path from s to t.

Running time: Constructing the map ℓ takes $O(|E|)$ time and so the running time of Dijkstra's algorithm dominates in the algorithm. Hence, the time complexity of the algorithm is the same as Dijkstra's algorithm, that is $O((|V| + |E|) \cdot log|V|)$.