CSE319: Modern Algorithm Design

Homework 1 Solutions

Submitted By: Divyajeet Singh (2021529) **Discussion Partner:** Siddhant Rai Viksit (2021565)

Solution 1.

Part (a)

We are given two weight functions w_1 and w_2 such that

$$w_1(e) \leq w_1(e') \iff w_2(e) \leq w_2(e') \quad \forall\, e, e' \in E \tag{1}$$

Since the given implication 1 is birectional, we can prove for any one direction and it will hold for the other direction without loss of generality.

Claim 1. *Any MST of a graph $G = (V, E)$ with weight function w_1 is also an MST of G with weight function w_2.*

Proof. Let T_1 be an MST of G w.r.t w_1. Assume for the sake of contradiction that T_1 is not an MST of G w.r.t w_2. Let $T_2 \neq T_1$ be an MST of G w.r.t w_2. Then, there exists an edge $e_2 \in T_2 \setminus T_1$ such that

$$w_2(e_2) < w_2(e_1) \implies w_1(e_2) < w_1(e_1) \quad \text{(By 1)} \tag{2}$$

for some edge $e_1 \in T_1 \setminus T_2$. Adding e_2 to T_1 creates a cycle in T_1 containing both e_1 and e_2. Then, $T^* = T_1 \setminus \{e_1\} \cup \{e_2\}$ is a spanning tree with weight

$$w_1(T^*) = w_1(T_1) - w_1(e_1) + w_1(e_2) < w_1(T_1) \tag{3}$$

which contradicts the assumption that T_1 is an MST w.r.t. w_1. □

Corollary 1. *Any MST of a graph $G = (V, E)$ with weight function w_2 is also an MST of G with weight function w_1.*

Proof. Without loss of generality in 1 and by Claim 1, the corollary holds by swapping w_1 and w_2. □

Part (b)

We need to show that any MST of a graph G has the weight $n - W + \sum_{i=1}^{W-1} \kappa_i$, where κ_i is the number of components in the subgraph G_i of G having edges of weight at most i, and the edge weights are integers in the range $\{1, 2, \ldots, W\}$.

Claim 2. *The sequence $\kappa_0, \kappa_1, \ldots, \kappa_{W-1}, \kappa_W$ is monotonically non-increasing.*

Proof. We use κ_0 to denote the subgraph G_0 of G with edges having weight at most 0, i.e. no edges. This means G_0 has no edges, i.e. G_0 has n components. Then, $\kappa_0 = n$. If W is the weight the heaviest edge in G, then G_W contains all edges of G, i.e. G_W has only and exactly 1 component.
Suppose there is no edge in the graph with weight $w \in \{1, 2, \ldots, W\}$. Then, G_w contains no extra edges as compared to G_{w-1}, i.e. $\kappa_{w-1} = \kappa_w$.
Finally, in the general case for any G_i and G_{i+1} having κ_i and κ_{i+1} components respectively, we have $\kappa_i \geq \kappa_{i+1}$, because some edge of weight $i + 1$ may or may not connect some components in G_i. If all edges of weight $i + 1$ are inside the components already present in G_i, then $\kappa_{i+1} = \kappa_i$. Otherwise, they will connect some components, making $\kappa_{i+1} < \kappa_i$. □

Claim 3. *The weight of any MST of G is exactly $\sum_{i+1=1}^{W} (i + 1)(\kappa_i - \kappa_{i+1})$.*

Proof. We can prove the claim by showing that any correct MST-producing algorithm, say Kruskal's, produces an MST with the given weight. This is simple to see. In the sorted order of edges on the i^{th} iteration, the algorithm considers all edges of weight at most i if all edge weights are in the range $\{1, 2, \ldots, W\}$. $\kappa_i - \kappa_{i+1}$ represents the number of edges that are required to connect the components in G_i to form G_{i+1}. The weight of these edges is exactly $i+1$. So, Kruskal's algorithm adds $\kappa_i - \kappa_{i+1}$ many edges of weight $i+1$ to make the MST for every weight i. Since Kruskal's algorithm always produces a correct MST, the weight of any MST of G is exactly the given expression. □

Claim 4. *The weight of any MST of G is exactly $n - W + \sum_{i=1}^{W-1} \kappa_i$.*

Proof. We show that the expression in Claim 3 is equal to the given expression.

$$\begin{aligned}
\sum_{i+1=1}^{W} (i+1)(\kappa_i - \kappa_{i+1}) &= \sum_{i=0}^{W-1} (i+1)(\kappa_i - \kappa_{i+1}) \\
&= \sum_{i=0}^{W-1} i\kappa_i + \sum_{i=0}^{W-1} \kappa_i - \sum_{i=0}^{W-1} (i+1)\kappa_{i+1} \\
&= \sum_{i=0}^{W-1} i\kappa_i + \sum_{i=0}^{W-1} \kappa_i - \sum_{i=1}^{W} i\kappa_i \qquad (4) \\
&= n - W + \sum_{i=1}^{W-1} i\kappa_i + \sum_{i=1}^{W-1} \kappa_i - \sum_{i=1}^{W-1} i\kappa_i \\
&= n - W + \sum_{i=1}^{W-1} \kappa_i
\end{aligned}$$

where we crucially use the fact that $\kappa_0 = n$ and $\kappa_W = 1$. □

Part (c)

Given all edge weights in a graph $G = (V, E)$ are distinct. For the sake of contraction, let us assume that there are two MSTs T_1 and T_2 of G such that $T_1 \neq T_2$. Let $e_1 \in T_1 \setminus T_2$ be the lightest edge in T_1 that is not in T_2. Adding e_1 to T_2 creates a cycle C in T_2, since T_2 is a spanning tree. Consider an edge $e_2 \in C \setminus T_1$. Such an edge must exist, since T_2 is a spanning tree and e_1 is not in T_2. Since $e_2 \notin T_1$, then $w(e_1) < w(e_2)$, as e_1 must have been chosen, even when competing with e_2. Then, the tree $T^* = T_2 \setminus \{e_2\} \cup \{e_1\}$ is a spanning tree of G having

$$w(T^*) = w(T_2) - w(e_2) + w(e_1) < w(T_2) \qquad (5)$$

which contradicts the assumption that T_2 is an MST of G. □

Solution 2.

Part (a)

We show an implementation of the graph contraction algorithm in $O(m)$ time in Algorithm 1.
In the given implementation, we use a union-find data structure to keep track of the connected components of the graph. The union-find disjoint set is one such data structure, using which, we can find the root/parent of each connected component in near constant time.
The bottleneck of the algorithm are the loops over the edge-set E of the graph. We first loop over E to identify all connected components, which is done in $O(m)$ time (since each step takes roughly $O(1)$ time). The new vertex set V' is constructed in $O(n)$ time (we can construct V' by going over E as well). The new edge set E' is initialized with infinite weights, and then updated to the minimum edge weight between each pair of connected components, which is done in $O(m)$ time.
So, the overall time complexity[1] of the CONTRACT subroutine is $O(m)$.

[1]The actual time complexity of the algorithm is $O(m+n)$, since we incur $O(n)$ work to create the new vertex set. But I believe we assume for all practical purposes $n \leq m$. This additional work can be avoided while using the subroutine in Boruvka's algorithm if we contract on the fly, since we can obtain the new vertex set by finding the roots of vertices from the edges.

Algorithm 1 The CONTRACT subroutine in $O(m)$ time.

```
1: procedure CONTRACT(G = (V, E)):
2:     UFDS ← [v_1, v_2, ..., v_n]                    ▷ A union-find data structure for all vertices v ∈ V
3:     for (u, v) ∈ E do
4:         if (u, v) is BLUE then
5:             UFDS-UNION(u, v)
6:         end if
7:     end for
8:     V' ← {UFDS-FIND(v) | v ∈ V}
9:     E' ← {(UFDS-FIND(u), UFDS-FIND(v)) : ∞ | (u, v) ∈ E}    ▷ A mapping of new edges to weights
10:    for (u, v) ∈ E do
11:        r_u ← UFDS-FIND(u)
12:        r_v ← UFDS-FIND(v)
13:        if r_u ≠ r_v then
14:            E'[r_u, r_v] ← min{E[u, v], E'[r_u, r_v]}    ▷ Assuming a mapping of the form E[u, v] = w_uv
15:        end if
16:    end for
17:    return G' = (V', E')
18: end procedure
```

Part (b)

The solution to this problem requires showing a *bad example* for Boruvka's algorithm, where the number of edges in G remains $\Omega(m)$ for $\Omega(\log n)$ rounds. Consider a line graph of n vertices, having $m = n - 1$ edges.

Figure 1: A line graph with n vertices with $m = n - 1$ weighted edges.

We set the weights of the edges such that

$$w_i = \begin{cases} 1 & \text{if } i \in \{2k+1 \mid k = 0, 1, 2, \dots\} \\ 2 & \text{if } i \in \{4k+2 \mid k = 0, 1, 2, \dots\} \\ 4 & \text{if } i \in \{8k+4 \mid k = 0, 1, 2, \dots\} \\ \vdots & \vdots \\ 2^j & \text{if } i \in \{2^j \cdot (2k+1) \mid k = 0, 1, 2, \dots\} \end{cases} \tag{6}$$

In essence, we start with an unweighted graph. Weights are assigned to the edges in a fashion similar to the Sieve of Eratosthenes. We assign weight 2^j to alternate unweighted edges in the j-th round until all edges are marked. Now, we model the behavior of Boruvka's algorithm on the above graph.
In the first round, each vertex has exactly one edge weighted 1 incident on it. The other edge incident on the vertex will be heavier, since only alternate edges have weight 1. These are the $\frac{m}{2}$ edges which are contracted in the first round. We enter the second round with $\frac{m}{2} = \Omega(m)$ edges (and $\frac{n}{2}$ vertices). In the second round, each contracted vertex has exactly one edge weighted 2 incident on it. The other edge incident on the vertex will be heavier, since only alternate of alternate edges have weight 2. These edges will be contracted in the second round.
It is easy to see that in general, the number of edges in the graph in the k-th round will be $\frac{m}{2^k} = \Omega(m)$. Clearly, there will be exactly $\log_2 n = \Omega(\log n)$ rounds of the algorithm, since the number of vertices is halved in each round.

Part (c)

We propose an $O(m \log \log n)$ time algorithm to find the MST of a graph. The algorithm is a hybrid of Boruvka's and Prim/Jarnik's algorithms. The algorithm is described in Algorithm 2.
Let us analyze the time complexity of the algorithm. We crucially use the following facts

1. Each step of the Boruvka's algorithm takes $O(m)$ time.

Algorithm 2 An $O(m \log \log n)$-time algorithm to find the MST of a graph.

procedure MINIMUM-SPANNING-TREE($G = (V, E)$):
  for $i \leftarrow 1$ to $\log \log n$ **do**
    $G \leftarrow$ BORUVKA-ROUND(G) ▷ Color the lightest edge of each vertex and contract the graph
  end for
  $T \leftarrow$ PRIM-JARNIK(G)
  return T
end procedure

2. Boruvka's algorithm halves the number of vertices in each round. So, the number of vertices after k rounds on a graph G is $\frac{n}{2^k}$.

3. Prim/Jarnik's algorithm takes $O(m + n \log n)$ time to find the MST of a graph when implemented using Fibonacci heaps.

By fact 1, the $\log \log n$ rounds of Boruvka's algorithm take $O(m \log \log n)$ time. By this time, the contracted graph can contain at most

$$\frac{n}{2^{\log \log n}} = \frac{n}{\log n} \qquad \text{(By fact 2)} \tag{7}$$

vertices. For simplicity, we say that the contracted graph can still contain at most m edges. Then, the final application of Prim/Jarnik's algorithm takes (by fact 3)

$$O\left(m + \frac{n}{\log n} \log \frac{n}{\log n}\right) = O\left(m + \frac{n}{\log n} \log n - \frac{n}{\log n} \log \log n\right) = O(m + n) \tag{8}$$

time. The dominating factor thus, is the time taken by the steps of the Boruvka's algorithm. Formally,

$$O(m \log \log n + m + n) = O(m \log \log n) \tag{9}$$

Thus, the overall time complexity of the proposed Algorithm 2 is $O(m \log \log n)$.

Solution 3.

Part (a)

Algorithm 3 describes the usual Boruvka's algorithm to find the MST of a graph.

Algorithm 3 Boruvka's algorithm to find the MST of a graph.

procedure BORUVKA($G = (V, E)$):
  $T \leftarrow \emptyset$
  UFDS $\leftarrow [v_1, v_2, \ldots, v_n]$ ▷ A union-find data structure for all vertices $v \in V$
  while $|T| < n - 1$ **do**
    for $v_i \in V$ **do**
      $v_j \leftarrow \arg\min_{v:(v_i,v)\in E} \{w(v_i, v) \mid \text{UFDS-FIND}(v_i) \neq \text{UFDS-FIND}(v)\}$
      $T \leftarrow T \cup \{(v_i, v_j)\}$
      UFDS-UNION(v_i, v_j)
    end for
  end while
  return T
end procedure

Since the number of vertices gets halved in each round of Boruvka's algorithm, the algorithm requires $O(\log n)$ rounds to find the MST of a graph with n vertices. Note the following

1. In each round, the algorithm takes, in total, $O(m + n)$ to identify the lightest edge incident on each vertex. This is because for each vertex, we might need to scan all its adjacent edges to find the lightest one.

2. When implemented smartly, we can contract the graph on the fly. The union-find disjoint set data structure can be used to keep track of the connected components of the graph in near constant time. Its initialization takes $O(n)$ time.

So, the time complexity of the Boruvka's algorithm is $O((m+n)\log n) = O(m \log n)$. Now, we are given that the edges adjacent to each vertex are stored in increasing order of weights. This fact helps us in the following ways

1. The lightest edge incident on each vertex can be identified in $O(1)$ time. For each vertex, this edge is the leftmost edge in its adjacency list not considered so far. This can be simply achieved through pointers for each vertex.

2. The initialization of the union-find data structure still takes the initial $O(n)$ time and each of the union and find operations take near constant time. Thus, we have reduced the total time to identify the lightest edge incident on each vertex to $O(n)$.

3. The algorithm will see each edge only and exactly once (as opposed to going over them multiple times to find the minimum). This scanning of each of the m edges adds another $O(m)$ time to the algorithm.

Thus, we have seen that we require $O(n)$ time for each of the $\log n$ rounds of the Boruvka's algorithm required to find the MST, with an additional overhead of $O(m)$ to scan all the edges. So, if the adjacency list is sorted in increasing order of weights, the time complexity of the Boruvka's algorithm is $O(m + n \log n)$.

Part (b)

Given a list of N numbers and a parameter k, we want to partition the list into k buckets of size at most $\lceil \frac{N}{k} \rceil$ such that each bucket contains elements smaller than the elements in the bucket to its right. The algorithm is described in Algorithm 4.

Algorithm 4 An $O(N \log k)$ algorithm to partition a list into k buckets.

```
1:  procedure k-PARTIAL-SORT(A[1 : N], k):
2:      PIVOTS ← { cN/k | c = 1, 2, ..., k − 1 }
3:      for i ← 1 to log k do
4:          for j ← 0 to 2^i − 1 do
5:              p ← ⌈k/2^i⌉
6:              l ← PIVOTS[j · p]
7:              r ← PIVOTS[min{(j + 1) · p, N}]
8:              m_j ← QUICKSELECT(A[l : r], p)
9:              PARTITION(A[l : r], m_j)
10:         end for
11:     end for
12: end procedure
```

Before analysing the running time of the proposed algorithm, we make the following points.

1. The algorithm modifies the input list in-place, resulting in a k-partially sorted array. We can instead choose to return the set of the final $k-1$ pivots, or return the groups g_1, g_2, ..., g_k as a nested list.

2. We use two standard subroutines in the algorithm.

   (a) The QUICKSELECT subroutine finds the K^{th} smallest element of a list of n numbers in $O(n)$ time. We use the quickselect algorithm on portions/sub-sections of the original array.

   (b) The PARTITION subroutine partitions the list around a pivot, such that all elements smaller than the pivot are on the left, and all elements greater than (or equal to) the pivot are on the right. This can be done in $O(n)$ time for a list of n elements.

Here is an informal description of the algorithm. We identify $k-1$ evenly spaced pivot indices in the list[2]. We first identify the element that should sit at the $\lceil \frac{N}{2} \rceil$ position in the sorted list and partition the list around it. Then,

[2]These are the first $k-1$ multiples of $\lceil \frac{N}{k} \rceil$

we repeat the same procedure for the two halves formed after the partition - identify the element that should be at the $\lceil \frac{N}{4} \rceil^{th}$ position in the first half, and in the second half (separately). We then repeat this procedure for $\log k$ rounds[3].

Claim 5. *In the i^{th} iteration, the size of each group is at most $\lceil \frac{N}{2^i} \rceil$.*

Proof. The size of each group in the *active area* in the list is roughly halved in each iteration. This is because the pivots are evenly spaced throughout the array. In each iteration, we select the central pivot of the current sub-array. In fact, if the size of the array is a power of 2, the array will be halved exactly in each iteration. In the i^{th} iteration, the size of each group, thus, becomes at most $\lceil \frac{N}{2^i} \rceil$. □

Claim 6. *In the i^{th} iteration, we get a 2^i-partially sorted array.*

Proof. We can prove the claim using induction on i. The base case is $i = 1$, where we want to divide the array into 2 groups. We partition around the element which should sit in the middle of the sorted array (i.e. its median). Clearly, the size of the 2 groups is at most $\lceil \frac{N}{2} \rceil$, and by the partitioning property, the elements of the first group are smaller than those of the second.
Assume that the claim holds for some $i = I$. We prove the claim for $i = I + 1$. We partitioned the array into 2^I groups in the I^{th} iteration. In the $(I+1)^{th}$ iteration, we partition each of these 2^I groups into 2. We partition each bucket on the pivot location in its middle. Partitioning property guarantees the left-right sorted nature of the sub-buckets. This, in-turn, means that finally, the elements of any group are smaller than the elements of the groups to its right. Moreover, since the size of each group in the I^{th} iteration was at most $\lceil \frac{N}{2^I} \rceil$, the size of each group in the $(I+1)^{th}$ iteration is at most $\lceil \frac{N}{2^I} \cdot \frac{1}{2} \rceil = \lceil \frac{N}{2^{I+1}} \rceil$ by Claim 5. Thus, the claim holds for $i = I + 1$. □

Corollary 2. *The algorithm paritions the list into a k-partially sorted array.*

Proof. In the $(\log k)^{th}$ iteration, the algorithm partitions the list into k groups. By claim 6, the output list is k-partially sorted. □

The time complexity of Algorithm 4 is $O(N \log k)$. It is clear that the outer loop runs for $\log k$ iterations. Thus, we only need to show that the inner loop runs in $O(N)$ time for each i. Let $T(n)$ denote the time taken by the inner loop for a list of size n. It is clear that the time taken is proportional to the size of the list. Then,

$$T(n) = cn \quad (c > 0), \quad \text{i.e. } T(n) = O(n) \tag{10}$$

In the i^{th} iteration, the inner loop runs for 2^i iterations on lists of size at most $\lceil \frac{N}{2^i} \rceil$. So, the total work done in the i^{th} iteration is

$$2^i \cdot T\left(\frac{N}{2^i}\right) = 2^i \cdot \frac{cN}{2^i} = cN = T(N) = O(N) \tag{11}$$

Since we have $\log k$ iterations, the total time complexity of the algorithm is $O(N \log k)$.

Part (c)

If the edges adjacent to each vertex are only k-partially sorted, then we can achieve the following runtime.

1. The time taken to find the lightest edge incident on each vertex is $O\left(\frac{m}{k}\right)$, since we only always need to scan the first $\frac{m}{k}$ edges of each vertex to identify the lightest edge [4]. We incur an extra $O(m)$ effort once, in total, by scanning all edges of the graph.

2. We need to do this operation for each of the n vertices for each of the $\log n$ rounds.

This makes the final runtime of the algorithm $O\left(m + n \log n + \frac{m}{k} \log n\right)$.

Algorithm 5 An $O(m \log \log n)$-time algorithm to find the MST of a graph.

```
1: procedure MINIMUM-SPANNING-TREE(G = (V, E)):
2:     for i ← 1 to log log n do
3:         G ← BORUVKA-ROUND(G)
4:     end for
5:     for v ∈ V do
6:         G[v] ← k-PARTIAL-SORT(G[v], log |V|)        ▷ (log n)-partially sort the edges adjacent to each vertex
7:     end for
8:     T ← BORUVKA(G)
9:     return T
10: end procedure
```

Part (d)

Now we design another $O(m \log \log n)$-time algorithm to find the MST of a graph. The algorithm is described in Algorithm 5.
This algorithm is very similar to Algorithm 2 in **Problem 2 Part (c)**. We first run $\log \log n$ rounds of Boruvka's algorithm. Then, we partially sort the edges adjacent to each vertex to a degree of $k = \log n$. Finally, we run Boruvka's algorithm on the partially sorted graph. Combined with partial sorting, this run of Boruvka's algorithm behaves like a blackbox for any $O(m + n \log n)$-time MST algorithm.
Let us analyse the full time complexity of the algorithm.

1. As analysed earlier, the time taken to run $\log \log n$ rounds of Boruvka's algorithm is $O(m \log \log n)$.
2. After these rounds, if m_v edges are incident on each vertex v, then the time taken for partially sorting them is $O(m_v \log k)$. The total time taken to partially sort the edges adjacent to all vertices is thus
$$O\left(\sum_{v \in V} m_v \log k\right) = O(m \log \log n) \quad \text{Since} \sum_{v \in V} m_v \leq m \tag{12}$$
3. At this stage, the graph has at most $\frac{n}{\log n}$ vertices and m edges, as proved in **Problem 2 Part (c)**. It is also $(\log n)$-partially sorted. From **Problem 3 Part (c)**, Boruvka's algorithm runs in $O(m + n \log n)$ time on a $(\log n)$-partially sorted graph. So, the time taken to run Boruvka's algorithm on the partially sorted graph
$$O\left(m + \frac{n}{\log n} \log \frac{n}{\log n}\right) = O(m + n) \tag{13}$$

So, the overall time complexity of this MST-finding algorithm is $O(m \log \log n)$.

Solution 4.

Part (a)

Consider the following graph with 7 vertices and 7 edges.
Table 1 lists the steps of Dijkstra's algorithm for the above graph. It is easy to see that the shortest distance from A to G is 3, but Dijkstra's algorithm computes it as 8.

Part (b)

We prove the given claim (slightly modified to ease of understanding).

Claim 7. *If a node v is such that the shortest path from the source s to v contains at most K negative-length edges, then running Dijkstra's algorithm $K + 1$ times computes the length of this shortest s-v path.*

[3] In the final iteration, p becomes 1, so we partition the sub-array of size $\left\lceil \frac{2N}{k} \right\rceil$ around each pivot

[4] It is obvious that if you delete the first element from any k-partially sorted array, the resulting array is still k-partially sorted. This is because this *pop* operation essentially reduces the size of the first sorted bucket by 1. After multiple deletions, it will still be sufficient to scan the first $\frac{m}{k}$ elements to find the current required minimum.

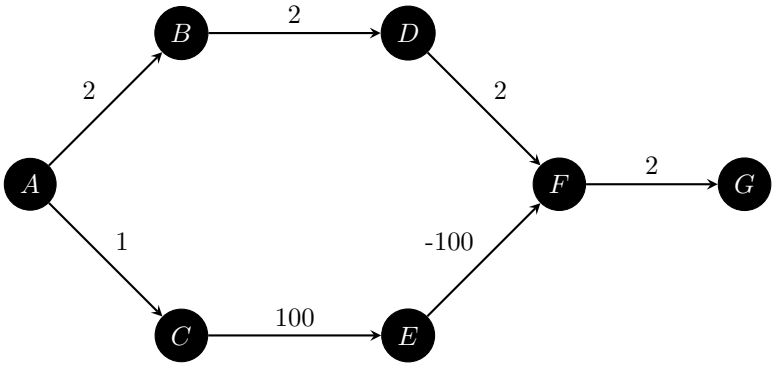

Figure 2: A graph with 1 negative weight edge.

Iteration	Node	A	B	C	D	E	F	G
0		0	∞	∞	∞	∞	∞	∞
1	A	0	2	1	∞	∞	∞	∞
2	C	0	2	1	∞	100	∞	∞
3	B	0	2	1	4	100	∞	∞
4	D	0	2	1	4	100	6	∞
5	F	0	2	1	4	100	6	8
6	G	0	2	1	4	100	6	8
7	E	0	2	1	4	100	1	8

Table 1: Table showing the computed *shortest* distances at each step.

Proof. The claim can be proven using induction on K. The base case, $K = 0$, is trivially true, since if the shortest s-v path has no negative-length edges, then Dijkstra's algorithm computes the correct shortest path length in 1 iteration.
We assume that the claim holds true for some $K = k$, and prove the claim for $K = k + 1$.
Let the shortest path between s and v be $P_{sv} = s \to v_{1:n} \to v = s \to v_1 \to v_2 \to \cdots \to v_n \to v$. Let P_{sv} contain $k+1$ negative edges. Let us say the $(k+1)^{th}$ negative edge is between v_i and v_{i+1}. By induction hypothesis, the $(k+1)^{th}$ iteration produces the correct distance for v_i. In fact, it also produces the correct distance for v_{i+1}, since the algorithm would update the path-lengths of v_i's neighbors. Note that the labels of $v_{i+2:n}$ and v may or may not be correct. This is because if v_{i+1} was already marked, then the labels of $v_{(i+2):n}$ and v would not be updated to their correct value. Then, one additional iteration is sufficient to update the distance labels of $v_{(i+1):n}$ and v, thereby computing the length of the shortest s-v path. □

Appendix

Problem 2 (b)

The example given as a solution to this problem may not be correct. The line graph of n vertices is a painfully obvious planar graph with $n - 1 \leq 3n - 6$ edges. This means that Boruvka's algorithm runs in $O(n)$ time, which sounds reasonable, as the effort required in each round goes down by half. As n decreases, m decreases along with it. However, a confusion arises when we think that $\frac{m}{2^k} = \Omega(m)$.