Assignment 8

Yu-Chieh Kuo B07611039†

†Department of Information Management, National Taiwan University

Problem 1

Ordinary array approach: In the case of the implementation for an ordinary array, for each vertex v, we need to find the unvisited adjacent vertex w with the minimal weight of edge, which takes $O(|V|^2)$. In addition, updating the value of SP after picking w takes $O(|E|)$. Hence, the total complexity by an ordinary array approach is

$$O\left(|V|^2\right) + O(|E|) \in O\left(|V|^2\right).$$

Min-heap approach: Firstly, building the heap takes $O(|V|)$. For each vertex v, the unvisited adjacent vertex with the minimal length w is the root of heap. Retrieving the heap root and re-heapifing the min-heap takes $O(|V| \log |V|)$. Moreover, updating the value of SP after picking w takes $O(|E| \log |V|)$ in the heap approach. Hence, the total complexity by an ordinary array approach is

$$O(|V| \log |V|) + O(|E| \log |V|) \in O((|V| + |E|) \log |V|).$$

Problem 2

We prove it by contradiction. Suppose there exists two distinct minimum cost spanning trees (MCST), say S and T. Edges in S and T sorted by the order of costs are

$$e_1^S, e_2^S, \cdots, e_n^S \quad \text{and} \quad e_1^T, e_2^T, \cdots, e_n^T.$$

Assume that e_i^S is the minimum cost edge in S but not in T, and reversely e_i^T is the minimum cost edge in T but not in S. Suppose $e_i^S < e_i^T$ WLOG, the graph G from $T \cup \{e_i^S\}$ contains a cycle.

Now, let e_k^G be the maximum cost edge of the cycle, which indicates e_k^G is not in any MCST. However, e_k^G is in G, which is built from $T \cup \{e_i^S\}$. That is, T is a MCST, which results in a contradiction.

Problem 3

3.(a)

A simple example is described as below. Consider a desired squared graph G with four vertices v, w_1, w_2, w_3 and the weights of corresponding existing edge $(v, w_1) = (v, w_2) = (w_2, w_3) = \ell$, and $(w_1, w_3) = 3\ell + k$. The minimum cost spanning tree of such a graph G is the same as the shortest-path tree rooted at v, where the edge (w_1, w_3) will be excluded.

3.(b)

Consider a desired triangle graph G with three vertices v, w_1, w_2 with the corresponding weights of edge $(v, w_1) = W_2, (v, w_2) = W_1, (w_1, w_2) = V$ following the order $W_2 > V > W_1$. Thus, the minimum cost spanning tree of such a graph G is different from the shortest path tree rooted at v, where the edge (v, w_1) will be exclude in the former, and (w_1, w_2) in the latter.

To examine whether two trees can be completely disjoint, we separate the discussion for the case of the vertex v with only one edge and more than one edges. In addition, we denote T_m and T_s by the minimum cost spanning tree and the shortest path tree root at v of the graph G for convenience. The idea for proofs comes from building contradictions.

Only one edge: The only edge must be both in T_m and T_s clearly; otherwise v is disjoint from T_m and T_s, a contradiction.

More than one edge: Let (v, u) be the edge rooted at v with the minimal edge weight. If (v, u) does not belong to T_m, then we could substitute (v, u) for any other edge (v, u') in T_m to make T_m be with lower weight. Hence, (v, u) must be in T_m.

If (v, u) does not belong to T_s, the shortest path from v to u contains other edge with total weight ℓ. However, $(v, u) < \ell$ for sure since G is a weighted graph. Hence, (v, u) must be in T_s

In conclusion, we state that the MCST and the shortest path tree cannot be completely disjoint.

Problem 4

Suppose there exists n vertices and m edges in a given graph G. To present the algorithm in suitable pseudocode utilizing the two operations of the Union-Find data structure, we first define Union-Find and its two operations $Find(\cdot)$ and $Union(\cdot, \cdot)$ formally.

We define a Union-Find over a set of n elements $X = \{x_1, x_2, \cdots, x_n\}$ and a collection of disjoint subsets $S_1, S_2, \cdots, S_k$ the elements in X belong to, where $1 \leq k \leq n$. Two operations supported by a Union-Find are defined as

- $Find(x)$: return S_i where $x \in S_i$.
- $Union(S_i, S_j)$: replace S_i and S_j with $S_i \cup S_j$.

A simple pseudocode is described as Algorithm 1

Algorithm 1 Kruskal's algorithm by Union-Find

```
function KRUSKAL'SALGORITHMBYUNIONFIND(G=(V,E))
    Union-Findify all vertices v in G
    for (u, v) ∈ E do
        if Find(u) ≠ Find(v) then
            Union(Find(u), Find(v))
        end if
    end for
end function
```

Now we analyze the complexity of Algorithm 1. The first stage, to sort edges by their weights, takes $\mathcal{O}(m \log m)$. Note that

$$m \leq n^2 \iff \log m \leq 2 \log n \implies \mathcal{O}(m \log m) \in \mathcal{O}(m \log n).$$

The second stage is to traverse all m edges in G and execute *Find* operation, which requires at most $2m$ operations. As a tree implementation of the Union-Find data structure that uses union-by-depth with depth d contains at least 2^d elements (that is, $n \geq 2^d \iff \log n \geq d$, the complexity of *Find* requires $\mathcal{O}(\log n)$. Consequently, this stage uses $\mathcal{O}(2m \log n) \in \mathcal{O}(m \log n)$.

The last stage is to unify two disjoint subsets ($Union(Find(u), Find(v))$). We execute at most n union process, and the function $Union(\cdot, \cdot)$ requires a linear time $\mathcal{O}(1)$. Hence, the time complexity is $\mathcal{O}(n)$.

In summary, the total complexity of Kruskal's algorithm by Union-Find is

$$\mathcal{O}(m \log n) + \mathcal{O}(m \log n) + \mathcal{O}(n) \in \mathcal{O}(m \log n).$$

Problem 5

The algorithm is described as Algorithm 2.

Algorithm 2 MCST Determinator

```
function MCST Determinator(G=(V,E), T)
    if (u, v) := increasing and (u, v) ∈ T then
        Remove (u, v) from T
        Run DFS on T from u and mark as 1
        Run DFS on T from v and mark as 2
        for (u', v') ∈ E and u'.mark ≠ v'.mark do
            if (u', v') < (u, v) then
                newEdge := (u', v')
            end if
        end for
        Add newEdge to T
    else if (u, v) := decreasing and (u, v) ∉ T then
        Add (u, v) to T
        C ← cycle in T
        for (u', v') ∈ C do
            if (u', v') > (u, v) then
                removeEdge := (u', v')
            end if
        end for
        Remove removeEdge from T
    end if
end function
```

Denote $T = (V', E')$ where $|V'| = |V|$ and $|E'| = |V| - 1$, traversing all edges in G takes $\mathcal{O}(|E|)$ and running *DFS* takes $\mathcal{O}(|V'| + |E'|) \in \mathcal{O}(|V|)$. In the second case of $(u, v) = \textit{decreasing}$, searching all cycles in T takes $\mathcal{O}(|V'| + |E'|) \in \mathcal{O}(|V|)$, and traversing all edges in C takes $\mathcal{O}(|E'|) \in \mathcal{O}(|V|)$. In conclusion, the total complexity is $\mathcal{O}(|V| + |E|)$.