COL351 Assignment 1

Viraj Agashe, Vishwas Kalani

TOTAL POINTS

60 / 60

QUESTION 1

Minimum Spanning Tree 20 pts

1.1 Unique MST 5 / 5

✓ **+ 5 pts Correct**

+ 0 pts Incorrect/Unattempted

+ 2 pts Some insights stated, but incorrect argument.

1.2 Fault resilient 15 / 15

✓ **+ 15 pts Correct**

+ 0 pts Incorrect/Unattempted

+ 6 pts Only algorithm written

+ 7 pts Correctness

+ 2 pts Time complexity

+ 6 pts Algorithm with larger time complexity + correctness proof

1 Didn't claim edge-fault resilient on edges which don't belongs to the MST.

QUESTION 2

2 Interval covering 15 / 15

✓ **+ 15 pts Correct**

+ 7 pts algorithm

+ 6 pts correctness

+ 2 pts Time complexity analysis

+ 0 pts Incorrect

QUESTION 3

Bridge edges 25 pts

3.1 Transitive relation 3 / 3

✓ **+ 3 pts Correct**

+ 0 pts Incorrect

3.2 Equivalence classes 5 / 5

✓ **+ 5 pts Correct**

+ 2 pts Algorithm

+ 2 pts Correctness

+ 1 pts Time complexity analysis

+ 0 pts Incorrect

3.3 Matrix A 12 / 12

✓ **+ 7 pts Correct Algorithm**

✓ **+ 5 pts Proof of Correctness**

+ 0 pts Incorrect/Not Attempted

3.4 Set E0 5 / 5

✓ **+ 3 pts Algorithm**

✓ **+ 2 pts Proof of Correctness**

+ 0 pts Incorrect/Not Attempted

1 Minimum Spanning Tree

1.1 Unique MST

Theorem 1. *A graph $G = (V, E)$ has a unique MST if all of its edge weights are unique.*

Proof. We will proceed to prove by contradiction. Suppose that the MST for the graph G is not unique. Consider two minimum spanning trees T_1 and T_2 of G such that $(x, y) \in T_1$ but $(x, y) \notin T_2$. Let $e = wt(x, y)$.

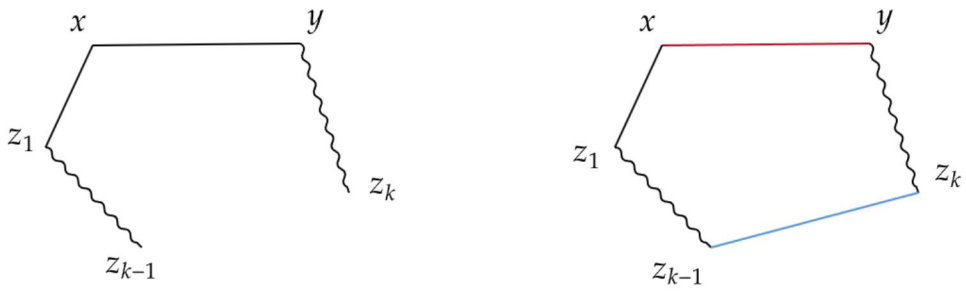

Figure 1: A tree T_1 containing the edge (x, y) and T_2 which does not contain it

Since T_2 is a spanning tree of G, there exists a path $z_0 z_1 ... z_k$, where $z_0 = x$ and $z_k = y$ from x to y in T_2.

Lemma 2. *$wt(z_i, z_{i+1}) < e$ for all $z_i, z_{i+1} \in z_0, z_1, ...z_k$*

Proof. Since edge weights are distinct, $wt(z_i, z_{i+1}) \neq e$. If $wt(z_i, z_{i+1}) > e$, we may remove the edge (z_i, z_{i+1}) from T_2 and add the edge (x, y) to T_2 to get T' by cycle property.

Lemma 3. *$\exists$ (z_i, z_{i+1}) such that on removing (x, y) from T_1, z_i, z_{i+1} lie in different connected components.*

Proof. If such (z_i, z_{i+1}) do not exist, then x and y lie in the same connected component of T_1 after the removal of (x, y), i.e. $\exists$ another path from x to y in T, and therefore T is not a tree, which is a contradiction.

Now, by cycle property, remove the edge (x, y) from T_1 and connect (z_i, z_{i+1}) in T_1 to obtain T_1', which a smaller weight sum.

We arrive at a contradiction. Hence, there cannot exist an edge (x, y) such that $(x, y) \in T_1$ and $(x, y) \notin T_2$ i.e. $T_1 = T_2$. Thus there must exist a unique MST. □

4 Appendix

This section contains some proofs, definitions, etc. which are standard, but are used several times throughout the document.

4.1 Cycle Property of Spanning Tree

Theorem 17. *Let T be any spanning tree of the graph G. For any edge $e \notin T$, there exists a spanning tree containing e.*

Proof. Let the edge be (x, y). Since T is a spanning tree, $\exists$ a path P from x to y in T. Now, consider any edge $e' \in P$ and consider $T' = T \setminus \{e'\} \cup \{e\}$. We claim T' is a spanning tree of G.

- Note that T' has the same number of edges as T, i.e. $|V| - 1$.
- T' is connected. To see this, note that removal of an edge from a tree results in two connected components, and x, y lie in different components (otherwise, T would not have been a tree). Now consider any two vertices u, v. If the path from u to v did not use the edge e', they remain connected. If not, w.l.o.g. assume that $u \in C_x$ and $v \in C_y$, where C_z represents the connected component to which x belongs. After the addition of the edge e, a path from u to v can be construct by appending the path from u to x with the edge e and then appending the path from y to v.

We have, T' is connected and has $|V| - 1$ edges. Thus, T' is a tree, and hence a spanning tree of G ■

Corollary 18. ***(Cycle Property of MST)*** *Let T be an MST of a graph G. Let e be an edge of G that is not in T and C let be the cycle formed in $T \cup \{e\}$. Then for every edge $e' \in C$, $wt(e') \leq wt(e)$.*

Proof. Note that if $wt(e) < wt(e')$ for any edge in C, then by cycle property of spanning tree, $T \setminus \{e'\} \cup \{e\}$ is a spanning tree of lower edge weight sum, and so T is not an MST, which is a contradiction. Hence, the claim follows. ■

4.2 Alternate Algorithm for Edge-Fault Resilient Graph

Definition 19. For an edge $e = (x, y)$, $dist(x, y)$ is defined as the *minimum* of the maximum edge cost encountered on over the set of all paths P from x to y, excluding the direct edge path e in the graph $G = (V, E)$, i.e.

$$dist(x, y) = \min_{p \in P \setminus \{xy\}} \left(\max_{e \in p} (wt(e)) \right)$$

4.2.1 Algorithm

We propose the following algorithm to check whether a graph $G = (V, E)$ is edge-fault-resilient:

1. For every vertex $v \in V$, compute $dist(v, x)$ for all $x \in V$
2. For each edge $(x, y) \in E$: .
   - If $wt(x, y) < dist(x, y)$, then the graph is not edge-fault-resilient.
   - Continue otherwise.
3. If for all edges $wt(x, y) \geq dist(x, y)$, the graph is edge-fault-resilient.

4.2.2 Calculation of Dist

For the calculation $dist(v, x)$ from a vertex v for all vertices $x \in V$, we can use a slightly modified version of Djikstra's Algorithm, and run it for each vertex.

1.1 Unique MST 5 / 5

✓ **+ 5 pts** **Correct**

+ 0 pts Incorrect/Unattempted

+ 2 pts Some insights stated, but incorrect argument.

1.2 Edge-Fault Resilient Graphs

1.2.1 Algorithm

We propose the following algorithm to check if a graph $G = (V, E)$ is edge-fault resilient

1. Consider any edge e of the graph G and construct an MST T of the graph $G \setminus \{e\}$.
2. Check if T is an MST of G. If not, then the G is not edge-fault-resilient.
3. Otherwise, T is also an MST of G. Now for each $e \in T$:
   - Construct using T an MST of $G \setminus \{e\}$.
   - If such an MST cannot be constructed, G is not edge-fault resilient.
   - If the weight of this MST is not the same as the weight of T, G is not edge-fault resilient.
4. If for all edges $e \in T$, an MST of $G \setminus \{e\}$ is an MST of G, the graph is edge-fault resilient.

We elaborate on some portions of the algorithm:

Checking if T is an MST of G: We can check if the MST T of $G \setminus \{e\}$ is an MST of G as follows:

- Let $e = (x, y)$. Using DFS, we find the path between x, y (using parent pointers). This takes $O(n)$ time, since an MST is a tree, and contains $|E| = n - 1$ edges.
- We check the weight of each edge on this path. If there exists any edge e' such that $wt(e') > wt(e)$, then T is not an MST of G. Otherwise, T is an MST of G.

Theorem 4. *In a graph $G = (V, E)$ and an edge $e \in E$, an MST T of graph $G \setminus \{e\}$ is also an MST of graph G if and only if the path P from x to y in T satisfies for all $e' \in P$, $wt(e') \leq wt(e)$.*

Proof. [$\Rightarrow$] Consider an MST T of $G \setminus \{e\}$. Suppose to the contrary that $\exists e' \in P$ such that $wt(e') > wt(e)$. Then, we may replace e' by e by the cycle property to obtain an MST of smaller weight sum, which is a contradiction. Hence, we must have $wt(e') \leq wt(e)\ \forall\ e' \in P$.

[$\Leftarrow$] Suppose the path P satisfies $wt(e') \leq wt(e)\ \forall\ e' \in P$. Suppose the MST T of $G \setminus \{e\}$ which is not an MST of G. If T is not a spanning tree of G, the MST of G must contain e. Let this MST be T^*. Now, recall that in the path P from x to y we have for all $e' \in P'$, $wt(e') \leq wt(e)$. If $\exists e'$ with $wt(e') < wt(e)$ then by cycle property, $T' = T^* \setminus \{e\} \cup \{e'\}$ is a spanning tree of lesser weight sum, which contradicts that T is an MST. Thus, all $e' \in P$ must satisfy $wt(e') = wt(e)$. So, we may replace e with e' in T^* to get another MST, which is nothing but T . This is a contradiction, and for the edge $e \in E$, the MST of graph $G \setminus \{e\}$ is also an MST of graph G. □

Constructing an MST of $G \setminus \{e\}$ using an MST of G: If such a tree can be constructed, we can construct it in $O(m)$ time. The algorithm is as follows:

- Remove the edge $e = (x, y)$ from the tree T, which splits it into 2 connected components. Now perform a DFS on the resultant forest in $O(n)$ time to label the connected component of each vertex, since all vertices which are visited during $DFS(x)$ lie in C_x, i.e. the component containing x.
- Find the minimum weight edge (other than e) connecting C_x and C_y. We do this by considering for all edges (u, v) for each $u \in C_x$, except (x, y). If $v \in C_y$, we compare the edge weight with the existing minimum and update the minimum if $wt(u, v)$ is lower. Since we consider all edges incident on u for $u \in C_x$, the time taken by this step is given as:

$$T = \sum_{u \in C_x} deg(u) \leq \sum_{u \in G} deg(u) = O(m)$$

Note that if no such edge connecting C_x and C_y exists apart from (x, y), then $G \setminus \{e\}$ is not connected and cannot have an MST.

If such edges exist and we add the minimum of these (say e_1) to the forest, we make the following claim:

Claim 5. *$T \setminus \{e\} \cup \{e_1\}$ is an MST of $G \setminus \{e\}$.*

Proof. We consider two cases:

Case 1: Exists an MST T' of $G \setminus \{e\}$ such that $e_1 \in T'$. Define $w(C)$ as the sum of all edges present in the connected component C. Let the connected components of x and y in $T' \setminus \{e_1\}$ be C'_x and C'_y. Note there cannot be any other edge from C'_x to C'_y. If $T \setminus \{e\} \cup \{e_1\}$ is not an MST, we must have $w(C'_x) + w(C'_y) < w(C_x) + w(C_y)$. However, this contradicts that T is an MST of G, as we can then find an MST of lower weight sum by replacing e_1 by e in T'. Thus, $T \setminus \{e\} \cup \{e_1\}$ is an MST.

Case 2: For all MSTs T' of $G \setminus \{e\}$, $e_1 \notin T'$. Since T' is a spanning tree, there must exist a path $P = z_1 ... z_k$ from x to y in T', where $z_1 = x$ and $z_k = y$. We state the following lemma:

Lemma 6. *$\exists\ (z_i, z_{i+1})$ such that $z_i \in C_x$, $z_{i+1} \in C_y$.*

Proof. Note that here, C_x, C_y are the connected components containing x, y in $T \setminus \{e\}$. Suppose no such edge (z_i, z_{i+1}) exists, i.e. w.l.o.g we have $z_j \in C_x \forall j$. Then, x and y are still connected in T after the removal of $e = (x, y)$, i.e. T is cyclic, which is a contradiction. □

Using the above lemma, note that $w(z_i, z_{i+1}) \geq w(e_1)$, since e_1 was the minimum of all edges connecting C_x and C_y. Thus we can exchange (z_i, z_{i+1}) with e_1 to get another MST T'' which contains e_1. So this case is never possible. Therefore, $T \setminus \{e\} \cup \{e_1\}$ is an MST of $G \setminus \{e\}$■

1.2.2 Proof of Correctness

We now prove the correctness of our algorithm.

- Firstly, note that if we prove *any* MST of $G \setminus \{e\}$ is an MST of G, since the edge weight sum is the same, *all* MSTs of $G \setminus \{e\}$ are MSTs of G.
- If for a particular edge e, the MST T of $G \setminus \{e\}$ is not an MST of G, then the graph is clearly not edge-fault resistant by definition. Thus, step 1 and 2 are correct.
- We need not check any edges not in this MST T. For any edge $e' \notin T$, T is a spanning tree of $G \setminus \{e'\}$ as well as G.
- For all edges e present in the MST, we explicitly construct an MST of $G \setminus \{e\}$. If we cannot, on removing that edge we must get a forest for which an MST is not possible, by the above proof.
- For this constructed MST, if the weight sum is higher than the weight sum of the MST of G, i.e. if $wt(e_1) > wt(e)$, then it cannot be an MST of G. Further, *all* MSTs of $G \setminus \{e\}$ must have the same weight sum and cannot be MSTs of G. Thus, the graph cannot be edge-fault resilient. So step 3 is correct.
- If for all edges we have that an MST of $G \setminus \{e\}$ is an MST of G, by definition the graph is edge-fault resilient. Thus, step 4 and hence the algorithm is correct.

1.2.3 Time Complexity

We claim that this algorithm is $O(mn)$. We argue this as follows:

- Construction of the initial MST T takes $O(m \log n)$ time using Kruskal's algorithm.
- Checking if T is an MST of G takes $O(n)$ time.
- Constructing an MST of $G \setminus \{e\}$ takes $O(m)$ time, as proved above.
- Further, we need to do this step once for every edge in the MST T. Since there are $n - 1$ edges, the total time taken is $O(mn)$.

Thus, the total time complexity of the algorithm is $O(m \log n + n + mn) = O(mn)$.

4 Appendix

This section contains some proofs, definitions, etc. which are standard, but are used several times throughout the document.

4.1 Cycle Property of Spanning Tree

Theorem 17. *Let T be any spanning tree of the graph G. For any edge $e \notin T$, there exists a spanning tree containing e.*

Proof. Let the edge be (x, y). Since T is a spanning tree, $\exists$ a path P from x to y in T. Now, consider any edge $e' \in P$ and consider $T' = T \setminus \{e'\} \cup \{e\}$. We claim T' is a spanning tree of G.

- Note that T' has the same number of edges as T, i.e. $|V| - 1$.
- T' is connected. To see this, note that removal of an edge from a tree results in two connected components, and x, y lie in different components (otherwise, T would not have been a tree). Now consider any two vertices u, v. If the path from u to v did not use the edge e', they remain connected. If not, w.l.o.g. assume that $u \in C_x$ and $v \in C_y$, where C_z represents the connected component to which x belongs. After the addition of the edge e, a path from u to v can be construct by appending the path from u to x with the edge e and then appending the path from y to v.

We have, T' is connected and has $|V| - 1$ edges. Thus, T' is a tree, and hence a spanning tree of G ■

Corollary 18. ***(Cycle Property of MST)*** *Let T be an MST of a graph G. Let e be an edge of G that is not in T and C let be the cycle formed in $T \cup \{e\}$. Then for every edge $e' \in C$, $wt(e') \leq wt(e)$.*

Proof. Note that if $wt(e) < wt(e')$ for any edge in C, then by cycle property of spanning tree, $T \setminus \{e'\} \cup \{e\}$ is a spanning tree of lower edge weight sum, and so T is not an MST, which is a contradiction. Hence, the claim follows. ■

4.2 Alternate Algorithm for Edge-Fault Resilient Graph

Definition 19. For an edge $e = (x, y)$, $dist(x, y)$ is defined as the *minimum* of the maximum edge cost encountered on over the set of all paths P from x to y, excluding the direct edge path e in the graph $G = (V, E)$, i.e.

$$dist(x, y) = \min_{p \in P \setminus \{xy\}} \left(\max_{e \in p} (wt(e)) \right)$$

4.2.1 Algorithm

We propose the following algorithm to check whether a graph $G = (V, E)$ is edge-fault-resilient:

1. For every vertex $v \in V$, compute $dist(v, x)$ for all $x \in V$
2. For each edge $(x, y) \in E$: .
   - If $wt(x, y) < dist(x, y)$, then the graph is not edge-fault-resilient.
   - Continue otherwise.
3. If for all edges $wt(x, y) \geq dist(x, y)$, the graph is edge-fault-resilient.

4.2.2 Calculation of Dist

For the calculation $dist(v, x)$ from a vertex v for all vertices $x \in V$, we can use a slightly modified version of Djikstra's Algorithm, and run it for each vertex.

4.2.3 Time Complexity Analysis

We claim that the algorithm is $O(n^3)$. We argue this as follows:

- For a vertex $v \in V$, $dist(v, x)$ for all vertices can be computed in $O(n^2)$ time using Djikstra's algorithm.
- Therefore, for all vertices in V, this quantity can be computed with $O(n^3)$ amount of work.
- Finally, checking if $wt(x, y) < dist(x, y)$ for each edge requires $O(1)$ for each edge and so takes $O(m)$ in total.

Thus, the time complexity of the algorithm is $O(n^3)$.

4.2.4 Proof of Correctness

We introduce the mathematical basis of the algorithm in the form of the following lemmas and theorems, which will help us argue the correctness of the algorithm.

Lemma 20. *For a graph $G = (V, E)$ and an edge $e \in E$, exists an MST which does not contain e if and only if $\exists$ a path P from x to y such that for all $e' \in P$, $wt(e') \leq wt(e)$.*

Proof. [$\Rightarrow$] Suppose $\exists$ a path $P = xz_1...z_ky$ from x to y such that for all $e' \in P$, $wt(e') \leq wt(e)$. Consider an MST T containing the edge e. If $wt(e) > wt(z_i, z_{i+1}) \forall i$, by cycle property (proof later), exists an MST of smaller weight sum and T is not an MST. Consider an edge e' on this path with $wt(e') = wt(e)$. Again by cycle property, $T' = T \setminus e \cup e'$ is an spanning tree with the same weight sum. Thus T' is the required MST.

[$\Leftarrow$] Suppose there exists an MST which does not contain e. We will attempt to prove by contradiction. Suppose for all paths from x to y in G, exists an edge e' on the path such that $wt(e') > wt(e)$. Consider any MST T of G. Then exists a path P from x to y in T, say $xz_1...y$. Now we know exists $e' = (z_i, z_{i+1}) \in P$ such that $wt(z_i, z_{i+1}) > wt(e)$. By cycle property, we know $T \setminus \{e'\} \cup e$ is a spanning tree of G with smaller edge weight sum, which contradicts that T is an MST. Thus, there must be atleast one path with all edges satisfying $wt(e') \leq wt(e)$. □

Theorem 21. *In a graph $G = (V, E)$ we have for all edges $e = (x, y) \in E$ that, $\exists$ a path P from x to y such that for all $e' \in P$, $wt(e') \leq wt(e)$, if and only if for all edges $e \in E$, an MST of graph $G \setminus \{e\}$ is also an MST of graph G.*

Proof. [$\Rightarrow$] Suppose an MST T of graph $G \setminus \{e\}$ is also an MST of graph G, then T is an MST of G which does not contain e. By the above lemma, $\exists$ a path P from x to y such that for all $e' \in P$, $wt(e') \leq wt(e)$.

[$\Leftarrow$] Suppose $\exists$ a path $P = xz_1...z_ky$ from x to y such that for all $e' \in P$, $wt(e') \leq wt(e)$. We proceed by contradiction. Suppose there exists an MST T of $G \setminus \{e\}$ which is not an MST of G. Then note the following lemma:

Lemma. *The path P' from x to y in T must satisfy that for all $e' \in P'$, $wt(e') \leq wt(e)$*

Proof. If $P' = P$, we are done. If not, note that P and P' form a cycle in G, and so exists at least one $e_1 \in P$ which is not in the MST (as it cannot contain cycles). Then, if there is any edge $e' \in P'$ such that $wt(e') > wt(e)$, we have that $wt(e') > wt(e_1)$ and so by cycle property, $T \setminus \{e'\} \cup \{e_1\}$ is a spanning tree of smaller weight sum, which is a contradiction, so we must have for all e', $wt(e') \leq wt(e)$.

If T is not a spanning tree of G, the MST of G must contain e. Let this MST be T^*. Now, recall that in the path P' from x to y we have for all $e' \in P'$, $wt(e') \leq wt(e)$. If $\exists e'$ with $wt(e') < wt(e)$ then by cycle property, $T' = T^* \setminus \{e\} \cup \{e'\}$ is a spanning tree of lesser weight sum, which contradicts that T is an MST. Thus, all $e' \in P$ must satisfy $wt(e') = wt(e)$. So, we may replace e with e' in T^* to get another MST, which is nothing but T . This is a contradiction, and so for all edges $e \in E$, an MST of graph $G \setminus \{e\}$ is also an MST of graph G. □

4.2.5 Correctness Argument

Note that if for an edge $e = (x, y)$, $dist(x, y) \leq wt(x, y)$ implies that there $\exists$ a path P from x to y such that for all $e' \in P$, $wt(e') \leq wt(x, y)$ (since $dist(x, y)$ is the maximum edge on this path).

If this condition is true for all edges e, then by Theorem 6, for all edges $e \in E$, an MST of graph $G \setminus \{e\}$ is also an MST of graph G, and the graph G is edge-fault-resilient as desired. Otherwise, if this is violated for even one edge, the graph is not edge-fault-resilient, which is what the algorithm outputs. Therefore, the algorithm is correct. ■

1.2 Fault resilient 15 / 15

✓ **+ 15 pts Correct**

+ 0 pts Incorrect/Unattempted

+ 6 pts Only algorithm written

+ 7 pts Correctness

+ 2 pts Time complexity

+ 6 pts Algorithm with larger time complexity + correctness proof

1 Didn't claim edge-fault resilient on edges which don't belongs to the MST.

2 Interval Covering

2.1 Algorithm

1. Sort the intervals in the increasing order of their start time.
2. Begin with the starting point of the first interval s_0.
3. For all the intervals including the value s_0 with finish time $> s_0$, include the interval with the maximum finish time and set the next starting point as the end of the chosen interval.
4. If there are no intervals including a given point p_i with finish time $> p_i$ then make the next starting time s_i as the new point and continue the process.
5. Stop when there are no intervals remaining.

2.2 Psuedocode

Algorithm 1 Interval Covering Algorithm

Input: Set $X = \{[s_i, t_i] \; \forall \; 1 \leq i \leq n\}$ of n interval on the real line
Output: Smallest covering Y of X

```
Y = { }
Sort intervals in X by starting time
p ← s_0                                          ▷ X[0] = [s_0, t_0]
i ← 0
k ← 0                                            ▷ X[k] = max interval containing p
while i < n do
    if p ∈ X[i]  or t_i < p then                 ▷ X[i] = [s_i, t_i]
        if t_i > t_k then
            k ← i
        end if
        i ← i + 1
    else
        Add X[k] to Y
        if s_i > t_k then
            p ← s_i
            k ← i
        else
            p ← t_k
        end if
    end if
end while
```

2.3 Explanation of Pseudocode

To implement the algorithm mentioned above in $O(nlogn)$ time, we used the above mentioned data structures and logic. We are beginning from the starting point of the first interval of X sorted as per the starting point and maintaining an interval with index k to be pushed in our greedy solution set Y. At any instance, we maintain the invariant that $X[k]$ is the max interval containing p considered till now. The loop implemented keeps the following 3 cases in consideration :

- *If the current interval is containing p and has higher finish time* : In this case we change k and increment i
- *If the current interval becomes disjoint from the current best interval* : In this case we change our point p to the start point of current disjoint interval and k to the current index.
- *If the current interval might contain our next p*: In this case we change our point p to the end point of the max interval $X[k]$

2.4 Time Complexity Analysis

We argue that the algorithm takes $O(n \log n)$ time to execute. We argue as follows:

- The time taken to sort the list of intervals is $O(n \log n)$.
- In the main loop of the algorithm, we are iterating through the list and performing some $O(1)$ operations inside. We can do these operations for any index i maximum 2 times because for any index i, if we enter the **else** section of the loop, then since $k \leq i$ after performing the $O(1)$ operations of the **else** section and the intervals are sorted as per the start time, therefore we will definitely enter the **if** section in the next iteration where we will be incrementing i.

The most computationally expensive step is $O(n \log n)$, and therefore the time complexity of the algorithm is $O(n \log n)$ which is a **polynomial time algorithm**.

2.5 Correctness using Exchange Argument

Consider the solution produced by our algorithm as $p_0, p_1 \cdots p_{j-1}$.

Claim 7. *All the real points covered by the intervals of X which are less than the start point of any p_i are covered by the intervals $p_0, p_1 \cdots p_{i-1}$*

Proof. **Proof by induction**
Base case:
When i=0 then there are no points covered by intervals of X before its start time because the algorithm begins from the earliest starting point.
Induction hypothesis:
Let the claim be true for $i = k$ that is all the real points x <startpoint(p_k) are covered by the intervals $p_0, p_1 \cdots p_{k-1}$
Induction step:
Now we need to prove the claim for $i = k+1$. Since all the points before the starting point of p_k are already covered as per the induction hypothesis. We need to consider only those points (x) covered by intervals of X which satisfy startpoint(p_k)< x <startpoint(p_{k+1}). As per our algorithm, after reaching the startpoint of p_k, we chose the interval with maximum finish time. The next start point is either the end point of p_k or the start point of next interval which doesn't overlap with p_k. Therefore all possible x are covered.Hence the claim is true.

Claim 8. *There exists an optimal solution containing the intervals of our greedy solution that is $p_0, p_1 \cdots p_{j-1}$*

Proof. **Proof by Induction**
Base case:
Consider any optimal solution $q_0, q_1 \cdots q_{k-1}$ sorted as per the start time which covers all the real values of the interval X. Now consider p_0 as the interval with maximum finish time which includes the starting point of all intervals plotted on real line. We get another optimal solution if we swap p_0 with q_0 as we already have an optimal solution for the range from finish time of q_0 to the end of all the intervals and it will also be a solution for the range from finish time of p_0 to the end because finishtime(p_0)≥finishtime(q_0).

Induction hypothesis:
Consider there exists an optimal solution containing $p_0, p_1 \cdots p_{k-1}$
Induction step:
Let the optimal solution containing $p_0, p_1 \cdots p_{k-1}$ be of form $O = p_0, q_{\alpha_1}, \cdots p_1, q_{\alpha_2}, \cdots p_{k-1}, \cdots q_{\alpha_n}$. As per Claim 7, since it contains $p_0, p_1 \cdots p_{k-1}$, it covers all the points covered by intervals of X before the starting point of p_k. Now since O is an optimal solution, it must contain an interval q_{α_m} which covers the starting point of p_k. We can replace such q_{α_m} with p_k and maintain the optimality of solution as we have not changed the size of the set and all the real points are covered because as per our greedy algorithm p_k is the interval with maximum finish time which covers its starting point.

Final argument: As per Claim 8, we have an optimal solution that contains all the intervals of our greedy solution. That is, $\exists$ optimal set O such that $G = p_0, p_1 \cdots p_{j-1}$ and $G \subseteq O$ (thus sizeof(G)$\leq$sizeof(O)). Since G covers all the points covered by intervals of X, thus G is an optimal solution. ■

2 Interval covering 15 / 15

✓ **+ 15 pts Correct**

+ 7 pts algorithm

+ 6 pts correctness

+ 2 pts Time complexity analysis

+ 0 pts Incorrect

3 Bridge Edges

3.1 Transitivity of Relation

Relation : xRy if and only if there exists an x-y path in G not containing bridge edges.

Proof. Let xRy so there exists a path $x, u_1, u_2 \cdots u_m, y$ between x and y which doesn't contain bridge edges. Similarly let us assume that yRz so there exists a path $y, v_1, v_2 \cdots v_n, z$ between y and z which doesn't contain any bridge edges. Let us consider two sets $A = \{x, u_1, u_2 \cdots u_m, y\}$ and $B = \{y, v_1, v_2 \cdots v_n, z\}$.Consider v_j be the last vertex in the set B which is common in both set A and set B so no vertex other than v_j is common in the paths $P_1 = \{v_j, v_{j+1}, \cdots z\}$ and $P_2 = \{x, u_1, u_2, \cdots v_j\}$, therefore we can append them to get $x, u_1, u_2, \cdots v_j, v_{j+1}, \cdots z$ and thus we get a path between x and z which implies xRz.

3.1 Transitive relation 3 / 3

✓ **+ 3 pts** **Correct**

+ 0 pts Incorrect

3.2 Computing Equivalence Classes

Algorithm Proposed: Let us say there are k bridge edges in the graph $G = (V, E)$ and they are $\{e_1, e_2, \cdots, e_k\}$.

1. Remove all the bridge edges one by one. For instance removing first bridge edge e_1 will increase the number of connected components in G by 1. And we will continue the process until we have removed all the bridge edges. Let us say finally after removing all the bridge edges we have j connected components $C_1, C_2, \cdots, C_j$ in G where $j \geq k+1$ as each bridge edge increases the number of connected components by 1 (*Claim 9*).

2. The vertices in each connected component with no further bridge edge in each component form **equivalence class** with respect to the given relation (*Claim 10*), and they can be found by running **Depth first search** in the graph.

3. Begin **DFS** from any node of the graph and keep appending nodes to a list which are unvisited during dfs. Return back from nodes which are already visited. Once dfs from a node finishes with all the vertices in its connected component in its list, begin dfs from next unvisited node forming next equivalence class.

Time Complexity Analysis:

1. *Computation and removal of bridge edges :* Using the **dfs** algorithm discussed in class $O(m + n)$

2. *Finding vertices in each connected component :* Since we are using **dfs** therefore $O(m + n)$

3. *Total time complexity:* $O(m + n)$

Claim 9. *Removal of a bridge edge increases the number of connected components in graph by 1*

Proof. An edge (x, y) is a bridge edge if there is no path from x to y in $G - (x, y)$. Let us say that there are q connected components $C_1, C_2, \cdots C_q$ in a G initially. Since there is an edge between x and y before removal of the bridge edge therefore they are in same connected component C_j initially with $i \leq j \leq q$. After removal of (x, y), they have to be in different connected components since there is no path between them now breaking C_j in 2 components C_j^1 and C_j^2. As there has been removal of just 1 edge of the graph, the number of connected components increase by 1.

Claim 10. *R is an equivalence relation*

Proof. Proving the reflexivity,symmetricity and transitivity of the relation

1. **Reflexive:** Consider a path containing the vertex x, it contains no edge thus thus there is no bridge edge. Thus xRx.

2. **Symmetric:** if xRy then this bridgeless path is a bridgeless path between y and x thus yRx

3. **Transitive:** Proved in the **3a**

Claim 11. *The vertices in each connected component after removal of all bridge edges form an equivalence class with respect to the given relation*

Proof. Consider any two vertices v_1 and v_2 belonging to the same connected component C_j. Since they belong to the same connected component there exists a path between v_1 and v_2. and as there are no bridge edges in the graph now therefore they have a path without any bridge edge in it and thus $v_1 R v_2$.
Since the different connected components of a graph are disjoint and partition all the vertices of the graph, hence removal of all the bridge edges from the graph partitions the graph in equivalence classes. ■

3.2 Equivalence classes **5 / 5**

✓ **+ 5 pts Correct**

+ 2 pts Algorithm

+ 2 pts Correctness

+ 1 pts Time complexity analysis

+ 0 pts Incorrect

3.3 Witness Matrix

Definition 12. Let C_i and C_j be any two connected components of a graph $G = (V, E)$ after the removal of its bridge edges (assuming they exist). We define witness(i, j) as the bridge edge connecting C_i and C_j.

3.3.1 Algorithm

To compute the $n \times n$ witness matrix for G, we propose the following algorithm:

1. Compute all the bridge edges of the graph $G = (V, E)$.
2. Remove all bridge edges from the graph.
3. Perform a DFS which sets the connected component of each vertex of the graph.
4. Create an auxiliary graph $G' = (V', E')$ in which each $v \in V'$ represents a connected component of G and each $e \in E'$ represents a bridge edge between connected components.
5. For each $v \in V'$ do the following:
   - Perform a DFS traversal of G'
   - For each vertex x encountered during the traversal, we set the witness$(v, x) = e$, where e is the last edge traversed to reach x.
6. For any two vertices x and y unrelated by R, suppose their connected components are C_x and C_y (already computed by DFS). Then the corresponding entry in the witness matrix A is given by:

$$A[x][y] = \text{witness}(C_x, C_y)$$

3.3.2 Time Complexity Analysis

We claim that this algorithm is $O(n^2)$. We argue this as follows:

- All bridge edges of a graph can be computed in $O(m + n)$ time.
- In a connected graph, there can be atmost n components. By Claim 9, there can be atmost $n - 1$ bridge edges. So removal of bridge edges takes $O(n)$ time.
- The DFS traversal of G takes $O(m + n)$ time.
- Creation of the auxiliary graph G' takes $O(|V'| + |E'|)$ time. Since no. of connected components is atmost n and no of bridge edges is atmost $n - 1$, we have $|V'| \leq n$ and $|E'| \leq n - 1$ and hence this step is $O(n)$.
- We perform a DFS traversal of the auxiliary graph $O(|V'|) = O(n)$ times, and each traversal takes $O(|V'| + |E'|) = O(n)$ time. The total time for this step is hence $O(n^2)$.
- Filling up of the witness matrix A takes $O(1)$ time for each entry and is hence takes a total of $O(n^2)$ time.

Note that $m = |E| \leq \frac{n(n-1)}{2}$ (for a fully connected graph). Thus $O(m + n) = O(n^2)$ in the worst case. The most computationally intensive step is $O(n^2)$ and hence the algorithm is $O(n^2)$.

3.3.3 Proof of Correctness

We need to prove that each entry $A[x][y]$ of the witness matrix contains an edge e satisfying that all paths from x to y in the graph use the edge e. To prove this, consider the following theorem:

Theorem 13. *The auxillary graph $G' = (V', E')$ is a tree.*

Proof. We make the following observations:

- We know that by **Claim 9**, the removal of a bridge edge increases the number of connected components of a graph by 1, i.e. the number of bridge edges must be the number of connected components of the graph after the removal of all edges minus one. But this is the very definition of V' and E'. Thus, we must have $|E'| = |V'| - 1$.
- Suppose that there exists a cycle $C_iC_{i+1}...C_j$ in the graph G'. Then, suppose we remove an edge e present in this cycle, connecting C_k and C_{k+1} (say). For all walks in the graph which used the edge e, there exists a walk containing $C_{k+1}...C_jC_i...C_k$ instead of C_kC_{k+1}. Thus any components which were connected earlier are still connected, which contradicts that e is a bridge edge. Thus, G' must be acyclic.

We know that any acyclic graph satisfying $|E| = |V| - 1$ is a tree. Therefore, G' is a tree. □

Corollary 14. *For any two vertices x, y lying in different equivalence classes C_x, C_y of R, all paths from x to y contain every edge from the path from C_x to C_y in G'.*

Proof. Since G' is a tree, between any two components C_x, C_y, there is a unique path P between them, i.e. they use a unique set of bridge edges. Thus, the paths between any vertices $x \in C_x$ and $y \in C_y$ must contain all the bridge edges in this path P. □

3.3.4 Correctness Argument

Using the above corollary, it is clear that $A[x][y]$ can contain any of the bridge edges from the path from C_x to C_y in G. The algorithm finds this path using DFS, and makes a valid choice, by choosing the last traversed edge on this path. The correctness of the components of the algorithm, including DFS, have been argued in class. Thus, the algorithm is correct. ■

3.3 Matrix A 12 / 12

✓ **+ 7 pts** Correct Algorithm

✓ **+ 5 pts** Proof of Correctness

+ 0 pts Incorrect/Not Attempted

3.4 Making a Graph Bridge-Free

3.4.1 Algorithm

We can make a slight modification in our algorithm of part 3*c* for construction of an *Auxiliary graph* and use the construction of that graph to make our graph bridge less.

1. Compute all the bridge edges of the graph $G = (V, E)$.
2. Remove all bridge edges from the graph.
3. Perform a DFS which sets the connected component of each vertex of the graph. Maintain an array which sets the *representative vertex* of each connected component as the vertex where the DFS ends in a connected component.
4. Create an auxiliary graph $G' = (V', E')$ in which each $v \in V'$ represents a connected component of G and each $e \in E'$ represents a bridge edge between connected components. From our array of representative vertices, we can access the representative vertex of any component as $A[v_i] = r_i$.
5. Search for the leaf in the auxiliary graph(which is a tree) and connect the Representative vertex of the leaf of the graph ($A[v_j] = r_j$) with the representative vertex of all the vertices of the graph except the neighbour of the leaf. Add these edges to the set E_0
6. Adding these initially non-existent edges in the graph will make our graph bridge less.

3.4.2 Time Complexity Analysis

We claim that this algorithm is $O(m + n)$. We argue this as follows:

- All bridge edges of a graph can be computed in $O(m + n)$ time.
- In a connected graph, there can be atmost n components. By **Claim 9**, there can be atmost $n - 1$ bridge edges. So removal of bridge edges takes $O(n)$ time.
- The DFS traversal along with maintaining a representative vertex of different components of G takes $O(m + n)$ time.
- Creation of the auxiliary graph G' takes $O(|V'| + |E'|)$ time. Since no. of connected components is atmost n and no of bridge edges is atmost $n - 1$, we have $|V'| \leq n$ and $|E'| \leq n - 1$ and hence this step is $O(n)$.
- Since we are adding maximum $n - 2$ edges to the graph among the known representative vertices, this step takes $O(n)$ time.

3.4.3 Proof of Correctness

We already know from **Theorem 13** that our auxillary graph is a tree.

Claim 15. *Addition of edges between a leaf of a tree and all other vertices except the neighbour of that leaf will result in 2 paths between any 2 vertices of the tree.*

Proof. Let the tree T have $v_1, v_2, \cdots, v_n$ as its vertices with v_j as one of the leaf of a tree (a tree always contains atleast 2 leaves). Let v_k be the only neighbour of the leaf v_j. Let us now add the following $n - 2$ egdes to our tree :

$$(v_j, v_i), 1 \leq i \leq n, i \neq j, i \neq k$$

For any vertex $v_i \neq v_k$, we had a path from v_i to v_j in the tree which was not a direct edge but now we have a direct edge path as well resulting in 2 paths from $v_i \neq v_k$ to v_j. For the vertex v_k, we had an edge

in the original tree and now we have new path between v_k and v_j passing through any of the neighbours of v_k (v_k will definitely have a neighbour other than v_j because we can't have two connected leaves in a tree). Consider any two random vertices v_p and v_q where $p, q \neq j$. We had a path between them in the original tree which was not passing through v_j because it was a leaf and now we have a new path $v_p v_j v_q$ between them. Therefore we have 2 paths between any two vertices in our new graph. □

Claim 16. *The set E_0 generated by the algorithm satisfies $E_0 \cap E = \phi$ where E is the original edge set of the graph.*

Proof. We are adding edges between representative vertices of two different connected component C_i and C_j represented by v_i and v_j in the *Auxiliary graph* where one of v_i or v_j is a leaf. Since leaf is originally having just one bridge edge to it neighbour and the connected components of the graph which form the *Auxiliary graph* don't have any edge other than bridge edges between them as per **Claim 11** thus all the edges added by the algorithm are new. □

3.4.4 Correctness Argument

The edges which were not bridge edges in the original graph $G = (V, E)$ can't become bridge edges by addition of new edges. We are adding maximum $n - 2$ new edges as shown in **Claim 15** depending on the number of vertices in the *Auxiliary graph*. Consider the set $E' = B \cup E_0$ where B represents the set of bridge edges in G. The set E' is the set of edges of the *Auxiliary graph* after we have added the edges of E_0 to it. By **Claim 15**, we have that any two vertices of *Auxiliary graph* have at least two paths between them . Appending those paths paths to the internal path within each connected component which doesn't include any edge from the set E', we get two paths between any two vertices which are end points of $e \in E'$ and thus each edge of E' is contained in a cycle in $G_0 = (V, E \cup E_0)$. Thus there are are no bridge edges in G_0 as an edge is a bridge edge if and only if it is not contained in a cycle. ■

3.4 Set E0 **5 / 5**

✓ **+ 3 pts** **Algorithm**

✓ **+ 2 pts** **Proof of Correctness**

+ 0 pts Incorrect/Not Attempted

COL351: Assignment 1

Vishwas Kalani (2020CS10411)
Viraj Agashe (2020CS10567)

August 2022

Contents

1 Minimum Spanning Tree **2**
1.1 Unique MST . . . 2
1.2 Edge-Fault Resilient Graphs . . . 3
1.2.1 Algorithm . . . 3
1.2.2 Proof of Correctness . . . 4
1.2.3 Time Complexity . . . 5

2 Interval Covering **6**
2.1 Algorithm . . . 6
2.2 Psuedocode . . . 6
2.3 Explanation of Pseudocode . . . 6
2.4 Time Complexity Analysis . . . 7
2.5 Correctness using Exchange Argument . . . 7

3 Bridge Edges **9**
3.1 Transitivity of Relation . . . 9
3.2 Computing Equivalence Classes . . . 10
3.3 Witness Matrix . . . 11
3.3.1 Algorithm . . . 11
3.3.2 Time Complexity Analysis . . . 11
3.3.3 Proof of Correctness . . . 11
3.3.4 Correctness Argument . . . 12
3.4 Making a Graph Bridge-Free . . . 13
3.4.1 Algorithm . . . 13
3.4.2 Time Complexity Analysis . . . 13
3.4.3 Proof of Correctness . . . 13
3.4.4 Correctness Argument . . . 14

4 Appendix **15**
4.1 Cycle Property of Spanning Tree . . . 15
4.2 Alternate Algorithm for Edge-Fault Resilient Graph . . . 15
4.2.1 Algorithm . . . 15
4.2.2 Calculation of Dist . . . 15
4.2.3 Time Complexity Analysis . . . 16
4.2.4 Proof of Correctness . . . 16
4.2.5 Correctness Argument . . . 17