Assignment 1

Harshit Mawandia, Tanish Tuteja

August 2022

Q1

(a) **Proof by contradiction**

Suppose $G = (V, E)$ has 2 distinct MSTs $T_1 = (V, E_1)$ and $T_2 = (V, E_2)$. Let the set S contain the edges lying in exactly one of the two trees T_1 and T_2. That is,

$$S = (E_1 \setminus E_2) \cup (E_2 \setminus E_1)$$

Since T_1 and T_2 are distinct, $|S| \geq 2$

Consider the edge $e \in S$ such that e has the smallest weight. Since all the weights are distinct,

$$wt(e) < wt(k) \forall k \in S - \{e\}$$

We know e belongs to either E_1 or E_2.

w.l.o.g assume $e \in E_1$. Consider $G' = (V, E_2 \cup \{e\})$. G' will contain a cycle $C = \{e_1, e_2, ..., e_n\}$ such that $e \in C$.

Claim: $\exists e' \in C$ such that $e' \notin E_1$.
Proof: If there exists no such e', $C \subseteq E_1$. This implies there exists a cycle in T_1. However, T_1 is a tree and thus can't contain a cycle.
Hence, proved $\exists e' \in C$ such that $e' \notin E_1$.

Let $e' \in C$ such that $e' \notin E_1$.

$$\Rightarrow e' \neq e$$

$$\Rightarrow e' \in S$$

$$\Rightarrow wt(e') > wt(e)$$

Now, $T_3 = (V, E_3)$ where $E_3 = E_2 \cup \{e\} \setminus \{e'\}$ is a spanning tree. Also,

$$\sum_{k \in E_3} wt(k) = \sum_{k \in E_2} wt(k) + wt(e) - wt(e')$$

$$\Rightarrow \sum_{k \in E_3} wt(k) < \sum_{k \in E_2} wt(k)$$

This contradicts the fact that $T_2 = (G, E_2)$ is an MST.

Thus, if all edges in a graph G are distinct, it has a unique MST.

(b) **Algorithm to check edge-fault-resistance**

```
T ← MST(V, E)
Visited ← φ

function CHECK-RESISTANCE(G′, x, y, w)
    if x=y then
        return(True)
    end if
    for all z ∈ x.children do
        if z ∉ Visited & (x, z).weight ≤ w then
            Visited ← Visited ∪ {z}
            if CHECK-RESISTANCE(G′, z, y, w) then
                return(True)
            end if
        end if
    end for
    return(False)
end function

function REMOVEEDGE(node)
    for all e ∈ T.Edges do
        G′ ← (V, E\{e}
        (x, y) ← e
        if CHECK-RESISTANCE(G′, x, y, e.wieght) = False then
            return(False)
        end if
    end for
    return(True)
end function

return(REMOVEEDGE(T.root))
```

Proof of Correctness

Claim: $\forall e \in E$, e is part of a cycle $(e_1, e_2, .., e)$ and $wt(e) \geq wt(e_i)$, $\Leftrightarrow \exists$ an MST T such that $e \notin T$.
Proof: Consider any minimum spanning tree T' containing e. Now, removing e from T' results in two disconnected trees. Adding any e_i will again connect these trees. Moreover, $wt(e) \geq wt(e_i)$ and thus $T' - e + e'$ is also a minimum spanning tree.

Conversely, let there exist an MST T such that $e \notin T$. Now, if e was not part of a cycle, e must have been in the MST. Thus, e is part of a cycle. Moreover, if $\exists$ an edge e' in the cycle such that $wt(e') > wt(e)$, then e' cannot be in the MST, which implies e must be in the MST, which is a contradiction. Hence, proved.

Let $G = (V, E)$ be the graph and T be an MST. Now for any edge $e = (x, y) \in T$, if we find a path $(x, a_1, a_2, ..., a_n, y)$ such that $wt(x, a_1), wt(a_i, a_{i+1}), wt(a_n, y) \leq wt(e)$,
$\Rightarrow e$ is part of a cycle $(e_1, e_2, .., e)$ and $wt(e) \geq wt(e_i)$.
$\exists$ an MST of G without e. Moreover, this MST will also be an MST of $G - e$.

However, if for an edge, we find no such path, it implies there is no cycle $(e_1, e_2, .., e)$ such that e has the maximum weight in the cycle. Thus, e is present in all MSTs of G, and the graph is not edge resilient.

Time Complexity

It takes $O(m + n)$ time to find out the MST of Graph $G = (V, E)$
For every edge $e \in MST$ it takes $O(m + n)$ time to check if its possible to find another MST of G by using DFS. Now there are at most $(n - 1)$ edges in an MST.
$\therefore$ total time complexity =

$$T = O(m + n) + O(n) \cdot O(m + n)$$

$$\Rightarrow O(mn + n^2)$$

Now since the graph is connected $m \geq (n - 1)$

$$2mn \geq mn + n^2$$

$$\therefore O(mn + n^2) = O(2mn) = O(mn)$$

Q2

Algorithm

$L' \leftarrow \textsc{SortByStart}(L)$
$I \leftarrow [L'[0]]$
$toAppend \leftarrow L'[0]$
$lastAdded \leftarrow L'[0]$
$i \leftarrow 1$
while $i < L'.length$ **do**
  if $L'[i].end \leq toAppend.end$ **then**
    $i \leftarrow i + 1$
    continue ▷ Do nothing
  else if $L'[i].start < lastAdded.end$ **and** $toAppend.end < L'[i].end$ **then**
    $toAppend \leftarrow L'[i]$
  else if $L'[i].start \geq lastAdded.end$ **then**
    $I.append(toAppend)$
    $lastAdded \leftarrow toAppend$
  end if
  $i \leftarrow i + 1$
end while

Proof of Correctness Consider an optimal solution $I' \subset L$, sorted by start time. The first element of I (from algorithm) and I' have to be same, and be the interval with minimum start time.

Now, if $I[1].start < I[0].end$, $I'[1].start$ must be $< I'[0].end$ as well. We also know $I[1].end \geq I'[1].end$. Thus, $I[1]$ completely overlaps at least as many intervals as $I'[1]$. In case $I'[1].start > I[0].end$, both I and I' must include $I[1]$, and thus $I[1]$ completely overlaps exactly as many intervals as $I'[1]$.
Thus, in any case, we can exchange $I[1]$ and $I'[1]$ without losing optimality. This forms the exchange argument.
Continuing in this way, $I[0], I[1]..., I[i]$ will completely cover at least as many intervals as $I'[0], I'[1], ..., I'[i]$. Thus, to cover the same number of intervals $|L|$, I will contain at most as many intervals as I'.

Time Complexity Sorting the intervals takes $O(nlogn)$ time, and then the iteration takes $O(n)$ time. Thus, the overall time

$$T = O(nlogn) + O(n)$$

$$= O(nlogn)$$

Q3

(a) **Proof by construction**

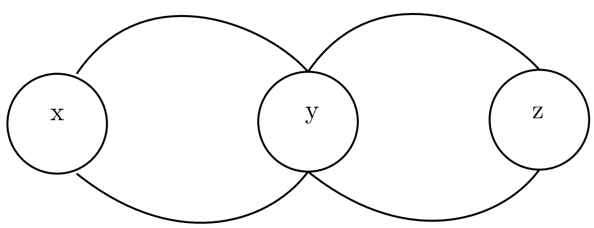
Let's take 3 vertices $x, y, z \in \mathcal{R} : (x\mathcal{R}y)$ & $(y\mathcal{R}z)$

Now, we know that no brigde edge between x and $y \Leftrightarrow \exists$ atleast 2 paths P_1 and P_2 between x and y : there is no common edge in P_1 and P_2.

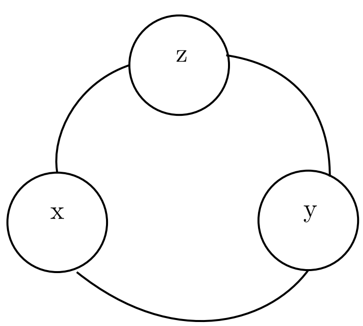
Similarly, $\exists$ atleast 2 paths P_3 and P_4 between y and z : there exists no common edge in P_3 and P_4.

Now there can be 2 cases :

Case 1:



Case 2:



In both cases we can see that there exists 2 paths from x to z that do not have any edge in common. $\therefore$ There exists no bridge edge between x and z. Hence, $x\mathcal{R}z$

(b) From lecture 7, we already know an algorithm to compute the bridge edges of a graph.

Algorithm:

$T = $ DFS-TREE(G)
$B = $ BRIDGE-EDGES(G)
$ConnectingBE(i, j) \leftarrow NULL \forall 0 \leq i, j \leq |B|$
function COMPUTE-EC(x, c, k)
  $x.class \leftarrow c$
  for all $e = (x, y) \in T$ **do**
    if $e \notin B$ **then**
      $k \leftarrow$ COMPUTE-EC(y, c, k)
    else
      $ConnectingBE(c, k) \leftarrow e$
      $k \leftarrow$ COMPUTE-EC$(y, k, k + 1)$
    end if
  end for
  return k
end function
$num - classes \leftarrow$ COMPUTE-EC$(root, 0, 1)$
for all $x \in V$ **do**
  $classes[x.class].append(x)$
end for

Proof of Correctness

Claim: Vertices of an equivalence class are connected in the DFS-Tree of a graph. i.e., $\forall$ equivalence classes C_i, $\forall x, y \in C_i, \exists$ path $(x, a_1, a_2, ..., a_k, y)$ in DFS-Tree T such that $a_j \in C_i \forall 1 \leq j \leq k$.

Proof: If possible, let C_a and C_b be two subsets of an equivalence class C such that C_a and C_b are disconnected in T.
$\Rightarrow \exists x \in C_a, y \in C_b$, path $(x, a_1, ..., a_k, y), k \geq 1$ in T such that $\exists a_j \notin C_a \cup C_b$
Since x and y are related, $\exists$ at least 1 more path in graph G between x and y. Let this path be $(x, b_1, b_2, ..., b_l, y)$.
Consider $a_j \notin C_a \cup C_b$. There are two paths from a_j to x : $(a_j, a_(j-1), ..., x)$ and $(a_j, a_(j+1), ..., a_k, y, b_l, ..., b_1, x)$. However, this contradicts the fact $a_j \notin C_a \cup C_b$. Thus, proved the vertices in an equivalence class are connected in the DFS-Tree of G.

This claim, along with the fact that bridge edges separate two equivalence classes, prove the algorithm.

Time Complexity

Generating the DFS-Tree of G takes $O(n + m)$ time. As discussed in lecture 7, we can compute all the bridge edges of a given graph in $O(m + n)$ time as well. Moreover, the function $Compute - EC$ essentially performs a DFS traversal, and thus requires $O(n+m)$ time as well. Finally, since the maximum number of connected components in G is n, the last for loop requires $O(n)$ time. Thus, total time

$$\begin{aligned} T &= O(n+m) + O(n+m) + O(n+m) + O(n) \\ &= O(n+m) \end{aligned}$$

(c) We can compute the equivalence classes $C = \{C_1, C_2, ..., C_k\}$ and the connecting bridge edges $ConnectingBE(C_i, C_j)$ as per (b).
Consider a new graph $G' = (C, E')$ where C represents the set of equivalence classes of V, and $\forall e = (x, y) \in B, e' = (class(x), class(y)) \in E'$ where B is the set of bridge edges of G.

Claim: G' is a tree.
Proof: Since each bridge edge connects two disconnected components in a class,

$$|B| = |C| - 1$$

Moreover, from definition,

$$|E'| = |B|$$

.

$$\Rightarrow |E'| = |C| - 1$$

Thus, G is a tree.

Algorithm

```
T ← Super − Graph(C, Connecting − BE).
function SINGLE-DFS(n, x, e)
    for all a ∈ n do
        for all b ∈ x do
            W(a, b) ← e
        end for
    end for
    for all (x, y) ∈ T, Level(y) > Level(x) do
        SINGLE-DFS(n, y, e)
    end for
end function
for all C_i ∈ C do
    for all e = (C_i, C_j) ∈ T do
        SINGLE-DFS(C_i, C_j, e)
    end for
end for
```

Proof of Correctness The algorithm is trivially correct since we have already proved that the super-graph must be a tree (T). Thus, for any $x \in C_i, y \in C_j, \exists$ a unique path from C_i to C_j, $(C_i, C_{a_1}, C_{a_2}, .., C_{a_k}, C_j)$ in T, which must be a subset of any path from x to y.
If (C_i, C_j) is an edge e, both $Single - DFS(C_i)$ and $Single - DFS(C_j)$ adds this edge to W. Otherwise, $W(C_i, C_j)$ is either (C_i, C_{a_1}) or (C_{a_k}, C_j) depending on whether $Single - DFS(C_i)$ or $Single - DFS(C_j)$ is called last.
Time Complexity

$Single - DFS$ visits every class node once and takes time

$$O(|C_i|) * O(\sum_{j=0, j \neq i}^{c} |C_j|)$$

$$= O(|C_i|) * O(n)$$

Now, the outer loop calling $Single-DFS$ is essentially a traversal as well, and thus runs once for every C_i. Thus, total time

$$T = \sum_{i=0}^{c} T(Single - DFS(C_i))$$

$$= \sum_{i=0}^{c} O(|C_i|) * O(n)$$

$$= O(n) \sum_{i=0}^{c} O(|C_i|)$$

$$= O(n) * O(n)$$

$$= O(n^2)$$

(d) **Algorithm**

We take the graph G' used in the c part of the question for our solution.

$T \leftarrow Super - Graph(C, Connecting - BE)$
$L \leftarrow NULL$
if $|T.root.children| = 1$ **then**
  $L.insert(T.root)$
end if
function GET-LEAF-NODES($node$)
  if $|node.children| = 0$ **then**
    $L.insert(node)$
  else
    for all $n \in node.children$ **do**
      GET-LEAF-NODES(n)
    end for
  end if
end function
GET-LEAF-NODES($T.root$)
$cn \leftarrow L.head$
$E_0 \leftarrow \phi$
while $cn.next \neq NULL$ **do**
  $E_0 \leftarrow E_0 \cup \{(cn, cn.next)\}$
  $cn \leftarrow cn.next$
end while
return E_0

Proof of Correctness For any two nodes $x, y \in G'$, there can be two cases:

Case 1: x and y share ancestor-descendant relationship. w.l.o.g assume x is ancestor of y. Now, $\exists$ a path $(x, a_1, a_2, ..., a_n, y)$ in T. Moreover, let k_1 be any leaf node in subtree of y.
From $x, \exists$ a path to the root. Now, if root in T had degree ≥ 2, go into another subtree from root, to a leaf node k_2. Otherwise, let $k_2 = root$ itself. In both cases, k_1 and k_2 are connected in the new edge set E_0. Consider the path $(x, b_1, b_2, ..., root, c_1, c_2, ..., k_2, d_1, d_2, ..., k_1, e_1, e_2, ..., y)$. Clearly, none of b_i, c_i, d_i, e_i can be equal to a_j. Thus, we have 2 paths having no common edges between x and y.

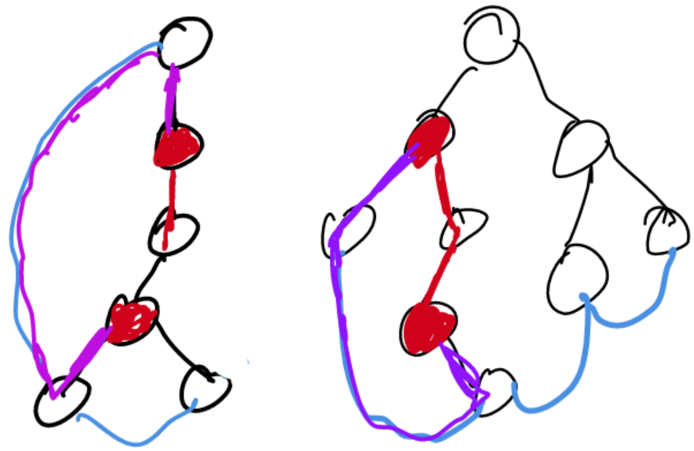


Figure 1: Case 1

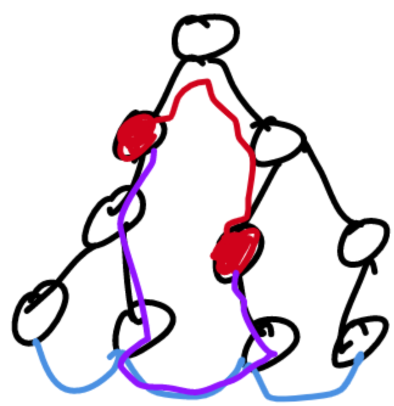


Figure 2: Case 2

Case 2: x and y do not have an ancestor-descendant relationship. In this case, again $\exists$ a path $(x, a_1, a_2, ..., a_n, y)$ in T. Moreover, subtrees of x and y are distinct.
Consider any leaf k_1 in subtree of x and leaf k_2 in subtree of y. Again, k_1 and k_2 are connected in the new edge set E_0.
Consider the path $(x, b_1, b_2, ..., k_1, c_1, c_2, ..., k_2, d_1, d_2, ..., y)$. Again, none of b_i, c_i, d_i can be equal to a_j. Thus, we have 2 path shaving no common edges between x and y.

Hence, for any two equivalence class nodes $x, y \in T$, there exist 2 paths between x and y with no common edges. Hence, for each node $a \in x$ and $biny$, there exist 2 paths between a and b with no common edges. Hence, there exist no bridge edges in this new graph.

We know the maximum number of leaves (including root) in a tree with n nodes can be n. Since $e \in E_0$ connect only leaves, $|E_0| \leq n - 1$.

Time Complexity Computing the equivalence classes and tree T takes $O(n + m)$ time. The function $Get - Leaf - Nodes$ is essentially a DFS traversal and thus also takes $O(n)$ time. Finally, creating E_0 also takes $O(n)$ time, resulting in overall complexity of:

$$T = O(n + m) + O(n) + O(n)$$

$$= O(n)$$