COL351 Assignment 1

Avval Amil - 2020CS10330
Ujjwal Mehta - 2020CS10401

September 1, 2022

1 Minimum Spanning Tree

Let G be an edge-weighted connected graph with n vertices and m edges.

Part (a) Prove that G has a unique MST if all edge weights in G are distinct.

Solution:
Here in order to prove that graph G has a unique MST we will first prove a few claims after which we will apply induction on the number of edges to prove the above statement. Note that since we are saying that at least one MST exists for the graph G so it should be connected and the relation between m and n should be -

$$m \geq n - 1 \tag{1}$$

(since the number of edges in MST are equal to n-1 hence G should have atleast that many edges)

Claim 1 : There exists a unique edge e with maximum weight such that the graph G' = (V,E\{e}) is connected (here assume that m > n-1 and the original graph is G = (V,E) is connected with distinct edge weights).

Proof : Since m > n-1 and since graph G is connected so there will exist at least 1 cycle in graph G and since the weights of all the edges are distinct, we can say that there would exist one such unique edge e with maximum weight(which will be a part of a cycle) upon removing which the new graph G' = (V,E/{e}) will remain connected. Hence the claim is proved.

Claim 2 : If e is the unique edge with maximum weight in graph G = (V,E) upon removing which the graph G' = (V,E\{e}) will remain connected then edge e will never be a part of MST of graph G.

Proof : We will prove this claim using contradiction that is let's say that there does exist an MST of G in which we have this edge e present and let's say in this MST, edge e connects 2 subtrees (of the MST) A and B (connected via edge e). Now since in our original graph (G), edge e was a part of a cycle (since upon removing it too the new graph G' was remaining connected) so we can say that there will exist another edge e' in graph G, which will connect A and B subtrees and weight(e') < weight(e) (by definition of e as it is unique maximum weight edge) so we can generate a new spanning tree by exchanging e and e' and the weight of this new spanning tree will be less then our original MST, hence our assumption that the initial spanning tree with edge e was a MST was wrong so by contradiction we can say that there will not be any MST which contains such edge e. Hence proved.

Now we will prove the original statement using induction on the number of edges (m) in G = (V,E) and using the results of Claim 1 and Claim 2 (we know using

equation (1) that m >= n-1).

Base Case : When we have m = n-1 then our orignal graph is a tree in itself so we can say that it will have a unique MST.

Induction Hypothesis : For the total number of edges K in the graph G' = (V,E) ,|V| = n, |E| = K, there exists a unique MST for all K < m.

Induction : Consider the graph G = (V,E) with the number of edges m, then by Claim 1 we can say that since there exist an edge e with maximum weight upon removing which the graph remains connected and also by Claim 2 we can say that this edge e will not be a part of any MST, hence the MST of G would be same as the MST of G' = (V,E\{e}) which is unique(by induction hypothesis) as the number of edges in G' are m-1. Hence by induction we can say that graph G with unique edge weights has a unique MST. Hence proved.

Part (b) A graph G = (V, E) is said to be edge-fault-resilient if the following condition holds: "For each edge e belongs to E, an MST of graph G - e is also an MST of graph G." Design an O(mn) time algorithm to check if a given connected graph G is edge-fault-resilient.

Solution :
First of all we will provide the algorithm to check if a graph G is edge-fault-resilient or not and then we will prove the correctness of our algorithm using certain claims followed by the time complexity analysis.

Algorithm :
The algorithmic steps are as follows : -

Step 1 : We will generate an MST M of graph G using Kruskal's MST algorithm (as done in the class)

Step 2 : For all the edges {a,b} belong to MST M do the following :-

Step 2a :- Do DFS traversal from vertex a and vertex b in the subtree (M - {{a,b}}) (i.e. 2 trees obtained after removing edge {a,b} in the MST M) in order to get 2 sets A and B (such that a $\in$ A and b $\in$ B) of vertices (i.e. divide the vertices of G in 2 different components) which represent the 2 connected components.

Step 2b :- Traverse over all the edges of the original graph G and if we are not able to find an edge {c,d} such that the end points lie in different components A and B and the weight({c,d}) = weight({a,b}) (also edge {c,d} and {a,b} are different) then stop the algorithm and **return False (i.e. the graph G is not edge-fault-resilient)**

Step 3: Once we have performed Step 2 over all the edges of MST M then we **return True (i.e. the graph G is edge-fault-resilient)**.

Proof of Correctness :
In order to give the proof of correctness of the above algorithm, we will first prove certain claims.

Claim 1 : If M be a given MST of graph G and {a,b} is an edge in that MST M then all the edges in graph G which connects two components of graph (M-{{a,b}}) will have the edge weight $\geq$ edge weight of {a,b} edge.

Proof :
We can prove this claim by contradiction, i.e. if there exists an edge {c,d} in graph G(and not equal to orignal edge {a,b}) connecting the 2 different components of (M - {a,b}) whose weight is $<$ weight({a,b}) then we can replace edge {a,b} with edge {c,d} and the weight of resultant tree will be less then original MST M which will be a contradiction since M is already a MST. Hence proved.

Claim 2 : If {a,b} is an edge of MST M of graph G and there exists no edge {c,d} (not equal to the original edge) connecting the two different components of M - {a,b} whose weight is equal to weight of edge {a,b} then any MST M' of graph G - {a,b} will not be the MST of graph G.
Proof :
Here by Claim 1 we know that if there is any edge {c,d} (not equal to original edge {a,b}) in graph G which connects the different components of (M - {a,b}) then weight({c,d}) $\geq$ weight({a,b}) and since we don't have any such edge with equal weight, so any edge connecting these two components will have an edge weight $>$ weight({a,b}) so we can say that if we consider any MST M' of graph G - {a,b} then in the path from vertex a to vertex b in MST M', there will be atleast 1 edge that has a weight $>$ weight({a,b}) (because the path starts with a vertex in A and ends in a vertex in B and we know that any edge joining a vertex in A to a vertex in B has more weight) so we can just exchange such an edge e' with edge {a,b} in M' to produce a new spanning tree of graph G and this tree will have weight less then M' so clearly M' will not be a MST of graph G. Hence proved.

Proof of Correctness :
Here in the proof of correctness we can say that if we consider any edge e of graph G that is not a part of MST M of graph G then if consider the graph G-e then M will also be its MST since e is not present in MST and the weight of MST of G-e cannot be less then weight of M else M will not be an MST of G, so we only need to check for the edges that are present in MST M of graph G if we want to check whether G is edge-fault-resilient or not.
Now in Step 2 of the algorithm we check for the edge-resilient property for each edge {a,b} of MST M, now here we say that we return False if we cannot find any

other edge e connecting the different components of M - {a,b} and has weight = weight({a,b}) which is actually a direct result of **Claim 2** and hence no MST M' of G - {a,b} will be MST of M and hence G will not be edge-fault-resilient. Now if we find an edge e with weight equal to {a,b} and connecting different components of M - {a,b} then we can generate a new MST of G by exchanging these two edges and this in turn will be an MST of G - {a,b} by similar argument given at the start. So after all such edges have passed this check we can say that our original graph G is edge-fault-resilient. Hence we have proved the correctness of our algorithm.

Time Complexity Analysis :
The time complexity analysis of this algorithm is simple since there are only 3 steps involved.

Step 1 takes O(mlogm + nlogn) time to generate the MST using Kruskal's algorithm (done in class).

In Step 2 we are traversing all the edges of MST M and at each end point of edge {a,b} we are doing DFS from vertex a and vertex b inside the tree and then we are checking for each edge of graph G is we can find some edge {c,d} :-
The time to process 1 edge of MST M is = O(m + n) (i.e. do DFS on MST M and then traverse over all edges m)
So the total time of processing all the edges of MST M = O(n*(m+n))

Step 3 takes O(1) Time.

So the total time complexity of our algorithm is O(n*(m + n) + mlogm + nlogn).
=> O(mn) Time algorithm.

2 Interval Covering

We will first start by giving an informal idea about our algorithm and then move on to the formal description of the algorithm, proof of a few claims and then finally proof of the algorithm's correctness using exchange argument.
The main greedy idea behind our solution is motivated by the fact that it is necessary to take intervals that start the earliest (as otherwise they will not be covered by intervals that start later than they do) and the fact that if multiple intervals start at the same time and we have to take one of them then it is better to take the one with the greatest ending time as it already covers all the intervals covered by the other intervals and it also has a chance to cover a few other intervals. We now formalize this notion below, starting with a few claims and then the algorithm.

Greedy Algorithm

Let there be n intervals in the set X. We can assume without loss of generality that the starting and ending times are positive (if not then just add to all of them a large enough positive number). Also for any interval I let start(I) and end(I) denote the start and end times of the interval.

Step 1
First we sort the intervals according to their starting times, if there are any ties then we break the ties according to the ending times (i.e. if there are two intervals with the same starting time, then the one with the smaller ending time will be first in the sorted order; if both start and end times are the same then there is no difference in which interval comes first).
This step takes $\mathcal{O}(n \cdot log(n))$ time. Let intervals after sorting (in order) be $I_1, I_2 \dots I_n$.

Step 2
We introduce a few variables to keep track of which stage we are in the algorithm - the variable *curr* will contain the index of the interval we are currently considering whether or not to include in our covering and the variable *last* will contain the index of the latest interval that was added to our covering. For the sake of notation we will also create a dummy interval I_0 with start and end times equal to -1. Initialize $curr = 1$ and $last = 0$ (note that we are not actually taking I_0 in our covering even thought *last* is initialized to 0). Also a covering C is initialized to the empty set. Pseudocode for the algorithm is given on the following page.

While ($curr \leq n$) do:

If (I_{curr} and I_{last} do not overlap) then do :
Initialize a variable $M = curr$
Initialize a variable $S = \text{start}(I_{curr})$
$curr{+}{+}$
While ($\text{start}(I_{curr}) = S$) do :
$M = curr$ and $curr{+}{+}$
Add I_M to covering C and $last = M$
Else do :
Initialize variables $m = \text{end}(I_{last})$ and $M = 0$
While ($\text{start}(I_{curr}) \leq \text{end}(I_{last})$) do :
If ($\text{end}(I_{curr}) > m$) then do :
$m = \text{end}(I_{curr})$ and $M = curr$
$curr{+}{+}$
If ($M \neq 0$) then do:
Add I_M to covering C and $last = M$

Return C

Now we prove a few claims about this algorithm.

Claim 1

If the current interval being considered does not overlap with the last interval (IF part of the first if-else statement) in the covering, then the next interval added to the covering must have start time same as that of the current interval and the largest end time out of all such intervals with the same start time.

Proof
Assume that the next interval added to the covering by the algorithm has starting time greater than that of the current interval (it cannot be lesser than that of the current interval as the intervals are considered in sorted order of starting time). But since the variable M is always assigned an interval whose start time is equal to $S = \text{start}(I_{curr})$ in the beginning, this is not possible. So, by contradiction we have proved this statement as in the end the interval I_M is added next. As the highest such value is assigned to M, and if the start times are the same the end times are in increasing order according to the sorting logic we applied, the last such interval (which is added finally) will have the largest end time.

Claim 2

If the current interval overlaps with the last interval (ELSE part of the first if-else statement) then an interval is only added if it partially overlaps (is not fully contained in) with the last interval or does not overlap with the last interval.

Proof
We can see that M is only assigned to intervals which have end time greater than m and the lowest value of m is the end time of the last interval, so if an interval satisfies this condition it cannot have end time less than that of the last interval (which is necessary for it being fully contained in the last interval). Also an interval is added only if M is given a non-zero value. So, the next interval added must either have partial overlap or no overlap with the last interval.

Claim 3

If the current interval overlaps partially with the last interval, then the start time of the next interval added must be less than or equal to the end time of the last interval and its start time must be at least that of the current interval. Also, its end time is the maximum among all intervals which partially overlap with the last interval added.

Proof
If the current interval partially overlaps with the last interval, then it satisfies the condition $I_{curr} > m$ initially and hence will have its value assigned to M. So some interval must be added in this iteration (it cannot be that no interval is added). Now, the WHILE loop under the ELSE statement only executes when the start time of the interval is less than or equal to the end time of the last interval, and execution of the WHILE loop is necessary for assignment of a value to M, so it follows that the next interval added must have start time less than or equal to the end time of the last interval. Also, since intervals are considered in sorted order of start times, this will have starting time at least as much as that of the current interval.
We can say that the end time of the next interval added must be the maximum out of the intervals partially overlapping with the last interval added as a new value is only assigned to M when the end time of that interval is more than the maximum encountered so far (as seen in the inner IF statement).

Claim 4

The first interval added to C has the lowest start time out of all the intervals in the set X.

Proof We can note that the first element added is according to the outermost IF statement (as no interval overlaps with the hypothetical I_0). Also the IF statement assigns the value of M to the index of an interval with start time equal to that of I_1 (specified in the WHILE loop inside the IF statement). This proves the claim.

Claim 5

Once all of set X has been covered, the algorithm does not add any more intervals to C.

Proof
Note that once all the set has been covered, the intervals left will definitely be completely contained in the last interval added to C and by Claim 2 these are never added to C and so no more intervals will be added to C.

Now we can see the formal proof of correctness of the above algorithm.

Correctness of Algorithm

Let $C^* = \{O_1, O_2 \dots O_k\}$ be an optimal covering and let $C = \{G_1, G_2 \dots G_l\}$ be the set returned by our greedy algorithm (both in increasing order of start times). First we need to prove that C is a covering, then we will prove that it is optimal by exchanging elements with that of C^*.

Proof that Algorithm Returns a Covering

We will prove this by induction on the number of elements n in the set X -
Base Case : $n = 1$
Here we can see that trivially as I_0 and I_1 do not overlap, I_1 will be added to the covering, which is a covering.
Induction Hypothesis : For any set of the size $n = k - 1$ the algorithm returns a covering.
Induction Step : Consider the sorting by start times of a set of size $n = k$. Remove the last element I_k in this set. We know due to the induction hypothesis that by the time the algorithm has considered the first $(k-1)$ elements of the set, it has created a covering of the set $(X \setminus \{I_k\})$. Now there are two cases for the last element - the first one that it does not overlap at all with the last interval added, or that it partially or fully overlaps with the last interval added.
If it does not overlap at all with the last interval added, then by Claim 1 proved above it must be added to the covering as it is the only interval with start time equal to its own.
If it is contained in the last interval added then by Claim 2 it will not be added to the covering, but the set returned is still a covering as the I_k is already covered by the last interval added in covering of $(X \setminus \{I_k\})$.
Finally, if it partially overlaps with the last interval added, then by Claim 3 it will be added to the set C as it is the only interval with start time equal to at least its own. This again creates a covering of the whole set X.
Thus, we have proved by induction that the algorithm returns a covering of the set X.

Proof of Optimality of Covering Returned by Algorithm (Exchange Argument)

We first note that since $C^* = \{O_1, O_2 \dots O_k\}$ is an optimal covering and $C = \{G_1, G_2 \dots G_l\}$ is a covering, $l \geq k$ by definition of optimality.
Now consider the set $C_1 = \{G_1, O_2 \dots O_k\}$, we will prove that C_1 is also a covering. First of all, by claim 4 we can say that G_1 has start time equal to that of O_1 (as O_1 also needs to have the minimum start time otherwise it won't cover intervals with the minimum start time). Also, by Claim 1 we can say that G_1 has a larger end time than O_1. This implies that O_1 is contained in G_1 or that G_1 covers all intervals covered by O_1. This shows that C_1 is also a covering and since its size is equal to that of C^*, it is an optimal covering.
Similarly, consider the set $C_2 = \{G_1, G_2, O_3 \dots O_k\}$. We can show that this too is a covering. There are two cases to be considered - one where O_2 overlaps with G_1 and one where it doesn't. If it does not overlap, then we can say that by Claim 1 that it is contained in G_2, and so C_2 is a covering. If it overlaps, then we can say by Claim 3 that end time of G_2 is at least as big as that of O_2. This implies that G_2 and G_1 together cover at least as much as O_2 and G_1 and hence C_2 is a covering. As it has the same size as C^*, it is an optimal covering.
Continuing this process and replacing elements k times, we can say inductively that the set $C_k = \{G_1, G_2 \dots G_k\}$ is an optimal covering of the set X. As all of these are intervals found by the greedy algorithm, we know by Claim 5 that the algorithm does not add any more intervals to this set and hence $C = C_k$.
Now as we have proved that cardinality of C is same as that of C^*, we have successfully proved that our greedy algorithm returns an optimal covering of the set X.

Time Complexity Analysis

As we have seen, step 1 of the algorithm runs in $\mathcal{O}(n \cdot log(n))$ time. We can see that step 2 of the algorithm runs in $\mathcal{O}(n)$ time as the variable *curr* is incremented once for every interval and it never decreases (also note that the other operations like checking for overlaps, comparisons and assignments can all be done in constant time).
This gives an overall time complexity of $\mathcal{O}(n + n \cdot log(n))$ which is $\mathcal{O}(n \cdot log(n))$ and hence is polynomial time.

3 Bridge Edges

(a) Transitivity of $\mathcal{R}$

Let x,y and z be vertices such that $x\mathcal{R}y$ and $y\mathcal{R}z$. We will prove that $x\mathcal{R}z$.

Proof
Let $x, a_1, a_2, \ldots a_m, y$ and $y, b_1, b_2 \ldots b_n, z$ be paths from x to y and y to z without any bridge edges. We can say that these exist from the definition of the relation $\mathcal{R}$.
We can concatenate these paths to create a walk between x and z -
$x, a_1, \ldots b_n, z$ without any bridge edges. Hence, we can say that the set of walks between x and z without any bridge edges is non empty. Using the Well-Ordering Principle on the sizes of all such walks, we can say that there exists a walk between x and z without any bridge edges of minimum size $x, c_1 \ldots c_k, z$. We claim that this is a path.
Assume that it isn't a path and that $c_i = c_j$ for some i and j. Then we can create a shorter walk by removing all elements between c_i and c_j which contradicts the minimality of the walk. Hence, we have shown by contradiction that no two vertices of the minimum length walk containing no bridge edges can be the same. This shows that there is a path containing no bridge edges between x and z or that $x\mathcal{R}z$. Hence, proved.

(b) Equivalence Classes of $\mathcal{R}$

First of all, it is trivial to see that $\mathcal{R}$ is reflexive as well as symmetric, this is because there exists the empty path from any vertex to itself and if there is a path without bridge edges from x to y then you can reverse that path to get a path without bridge edges from y to x. Hence, $\mathcal{R}$ is reflexive, symmetric as well as transitive (equivalence relation).

Claim

If all the bridge edges of G are removed, then the connected components of the remaining subgraph G' of G are the equivalence classes of $\mathcal{R}$.

Proof

Let after removal of all the bridge edges, x and y be in the same connected component of G'. This means that there exists a path in G' from x to y. As G' is a subgraph of G then this implies that there is a path from x to y in G without any bridge edges (as all edges of G' are non-bridge edges). Hence, $x\mathcal{R}y$ in this case.
Now, assume that x and y are not in the same connected component of G'. We will show that they are not related under $\mathcal{R}$ if this is the case. Assume for the sake of contradiction that $x\mathcal{R}y$. Then let there be a path P from x to y without any bridge edges in G. However, since only bridge edges are removed to create

the subgraph G', all edges in the path P are also present in G', which implies that there is a path from x to y in G'. But this contradicts the fact that they were in different connected components.
Hence, we have proved that two vertices are related under $\mathcal{R}$ if and only if they are in the same connected component of G' and so those connected components are precisely the equivalence classes of $\mathcal{R}$.

Algorithm to Compute the Equivalence Classes

Step 1
As done in the lectures, we can first compute all the bridges edges of G in $\mathcal{O}(m+n)$ time.

Step 2 Then remove all the bridge edges of G to get the subgraph G'. This takes $\mathcal{O}(m)$ time.

Step 3 Now, using DFS on the subgraph G', find its connected components. This step again takes $\mathcal{O}(m+n)$ time (running time of DFS on G' will be at most that of running time of DFS on G which was $\mathcal{O}(m+n)$). Return the connected components.

Since we have shown that the connected components of G' are the equivalence classes, these three steps describe the complete algorithm to compute the equivalence classes of $\mathcal{R}$ (proof of correctness is in the claim above).
The total time taken is $\mathcal{O}(m+n)$ as every step is at most $\mathcal{O}(m+n)$.

(c) Witness Matrix

We first prove a few claims and then describe the algorithm for computing the witness matrix. We will denote the witness matrix by W.

Definition

Let the set of equivalence classes induced by $\mathcal{R}$ be $C = \{C_1, C_2 \dots C_k\}$ and let the set of bridge edges of G be $A = \{a_1, a_2 \dots a_l\}$. Define a graph T with vertex set C and edge set $B = \{b_1, b_2 \dots b_l\}$ such that $\forall i \in \{1, 2 \dots l\}$, if $a_i = (x, y)$ such that $x \in C_j$ and $y \in C_k$ then $b_i = (C_j, C_k)$.

Claim 1

There are no self loops in T.

Proof

Suppose there exists an edge $b = (C_i, C_i)$ in T. Then this implies that there existed a bridge edge $a = (x, y)$ in G such that x and y belong to the same equivalence class C_i. Hence, there exists a path from x to y in G without any

bridge edges, which does not contain a as it is a bridge edge. However, if this is the case then considering the path without bridge edges as well as a together creates a cycle containing a. But this implies that a is not a bridge edge as no bridge edges can be part of a cycle (removing an edge from a cycle does not disconnect the graph). So by contradiction, there are no self loops in T.

Claim 2

There is at most one edge between two vertices in T.

Proof

Assume for the sake of contradiction that there exists 2 edges between vertices C_i and C_j in T. Then there must be two pairs of bridge edges joining C_i and C_j in G. However, there cannot be more than one bridge edges joining two equivalence classes as if this is the case, removing one of them does not disconnect the equivalence classes and hence does not disconnect the graph, which means that they can't be bridge edges. Hence, there cannot be more than one edge between vertices in T.

By the above two claims we have proved that T is a simple graph without any self loops or multiple edges.

Claim 3

T is a connected graph.

Proof

Consider for now two vertices of T which do not have a path between them (say C_i and C_j). Then this implies that there are two equivalence classes in G which cannot be reached from each other through bridge edges (as the graph T is essentially made by 'collapsing' the equivalence classes into one big vertex and preserving the bridge edges as they are).
We can formalize this notion as follows - Assume two vertices x and y of G such that $x \in C_i$ and $y \in C_j$. There exists a path P from x to y in G as G is a connected graph. For all the edges of path P, the vertices connected by that edge are in different equivalence classes if and only if the edge is a bridge edge. This follows by the definition of $\mathcal{R}$ and its equivalence classes - if they were in the same equivalence class then there would exist a path between them not containing any bridge edges and hence including the edge directly between them which is a bridge edge would create a cycle which is not possible as bridge edges cannot be part of cycles; also if they were not in the same equivalence class then a non-bridge edge cannot connect them as if this is the case then they are related under $\mathcal{R}$ and hence part of the equivalence class.
From the above argument we can say that the path P contains of subpaths (define a subpath to be a contiguous sequence of vertices which appear in the

same order that they do in the original path) of vertices which are in the same equivalence class, and the subpaths are joined together by bridge edges (define two subpaths to be joined by an edge if the last vertex of the first subpath and the first vertex of the second vertex are connected by that edge and the edge is part of the original path). Let first such subpath contain vertices in equivalence class C_{i_1}, the second in C_{i_2} and so on, which are connected by bridge edges a_{j_1}, a_{j_2} and so on.
But, this implies that there is path in T between C_i and C_j which is of the following form -
$C_i = C_{i_1}, C_{i_2} \dots C_{i_x} = C_j$ such that the edge b_{j_k} joins the vertices C_{i_k} and $C_{i_{k+1}}$ (this is because if a_{j_k} connects two vertices of G which are in classes C_{i_k} and $C_{i_{k+1}}$ then b_{j_k} connects the vertices C_{i_k} and $C_{i_{k+1}}$ in T).
This proves that contrary to our assumption C_i and C_j do indeed have a path between them and as they were any arbitrary vertices of T, it follows that all pairs of vertices of T have a path between them and hence T is connected.

Claim 4

T is acyclic.

Proof

Assume that T contains a cycle. Since the edges of the cycle belong to the set B, this implies that there exists a cycle in the original graph G with edges in the set A (by definition of B which uses A). But since A is the set of bridge edges, no edge in A can be part of a cycle in G, which gives us a contradiction. Hence, T is also acyclic.

From the above two claims, we have shown that T is a tree.

Claim 5

A graph can have at most n connected components, where n is the number of vertices.

Proof

We can see that the graph can have n connected components if it contains no edges (all vertices are isolated). Also if there were more than n connected components, due to the fact that there are only n vertices, at least one connected component must have zero vertices, which is not possible. Hence, the maximum number of connected components in the graph can be n.

Claim 6

If there is a path in T between C_i and C_j containing edges $B' = \{b'_1, b'_2 \dots b'_k\}$ then for any $x \in C_i$ and $y \in C_j$ (x and y are vertices of G) there exists a path

containing all the corresponding bridge edges $A' = \{a'_1, a'_2 \dots a'_k\}$ between them in G.

Proof

Let path between C_i and C_j be $C_i = C_1, C_2 \dots C_{k+1} = C_j$ such that $b'_l = (C_l, C_{l+1})\ \forall l \in \{1, 2 \dots k\}$. Let $a'_1 = (x_1, x_2)$. This implies that $x_1 \in C_1$ and $x_2 \in C_2$. As x and x_1 are in the same equivalence class, there exists a path between them with all vertices in between part of the same equivalence class. To this path append a'_1. Next, let $a'_2 = (x_2, x_3)$. As now x_1 and x_2 are in the same equivalence class there exists a path between them containing vertices all in that class. Append this path to the earlier path and to this append a'_2 to get path between x_1 and x_3. Inductively follow this to get path between $xandx_{k+1}$ and then append path between x_{k+1} and y to get path between x and y containing the desired edges. We can also say that this is a path with no repeated vertices as one vertex cannot come in two equivalence classes (equivalence classes partition the set).
NOTE : An important fact about this proof is that it does not require T to be a tree. This will be useful in the next section as even if we add a few edges to T, we can exactly like above prove that the same result holds.

Algorithm to Compute Witness Matrix

Now we describe the algorithm-

Step 1

Compute the equivalence classes of G induced by $\mathcal{R}$ using part (b) of this problem and store for each vertex which equivalence class it belongs to (this is simply done by the modification of DFS in which we see which connected component a vertex comes in when it is explored for the first time). This step takes $\mathcal{O}(m + n)$ time.

Step 2

Compute all bridge edges of the graph G in $\mathcal{O}(m + n)$ time and store them. Now, for any bridge edge we know which equivalence classes its end points belong to as this information was computed in step 1.

Step 3

Now, as we have both the equivalence classes and the bridge edges as well as the information about which bridge edge connects vertices belonging to which classes, we can create the graph T as specified in the definition above. This step takes $\mathcal{O}(m' + n')$ where m' is the number of bridge edges and n' is the number of connected components. As the number of connected components is at max n

and as $m' = n' - 1$ due to T being a tree (proved above), this step takes $\mathcal{O}(n)$ time.

Step 4

We can initialize an $n' \times n'$ matrix M such that $M[i][j]$ will store an edge that comes in the path from C_i to C_j in T.
Now for each vertex C_i in T, perform DFS rooted at C_i in T. Whenever a new vertex C_j is discovered, put $M[i][j] = b_{ij}$ if b_{ij} was the edge used to discover C_j (b_{ij} thus lies on the path from C_i to C_j).
This step takes $\mathcal{O}(n^2)$ time as we perform DFS for every vertex in T (there are $\mathcal{O}(n)$ vertices in T) and as every DFS takes $\mathcal{O}(m' + n') = \mathcal{O}(n)$ time.

Step 5

Now create the witness matrix W of size $n \times n$. For every entry $W(x, y)$ in it, check which classes x and y lie in (done in constant time as we have precomputed this), if they are the same then do nothing, if they lie in different classes C_i and C_j and $M[i][j] = b_{ij}$ then put $W(x, y) = a_{ij}$.
This step takes $\mathcal{O}(n^2)$ time as we have to check for every pair of vertices and each pair takes constant time due to the precomputation.
Return W.

Proof of Correctness

First, we can see that the witness matrix only contains entries for pairs of vertices which are not in the same equivalence class and hence are unrelated under $\mathcal{R}$.
We know that paths in a tree are unique for every pair of vertices. Using this we can say that if b_{ij} lies on the path between C_i and C_j in T then the corresponding bridge edge a_{ij} of G must lie on every path between a vertex in C_i and a vertex in C_j. This is because if there exists a path which does not go through a_{ij}, then we know that since there is also an edge that goes through a_{ij} (Claim 6), a_{ij} must lie on a cycle which is not possible as it is a bridge edge (this is equivalent to saying that b_{ij} is an edge in a tree and thus cannot be part of a cycle). Thus using this fact and the claims proved above, we have shown that if x and y are not related under $\mathcal{R}$ then $W(x, y)$ gives an edge which must be present in all paths between them.

Time Complexity Analysis

Since G is a connected graph, the maximum value of m can be $n(n-1)/2$ which is $\mathcal{O}(n^2)$. Due to this $\mathcal{O}(m + n)$ is $\mathcal{O}(n^2)$ for G. Now all 5 steps of the algorithm as described above take at most $\mathcal{O}(n^2)$ time.
Hence the running time complexity of this algorithm is $\mathcal{O}(n^2)$.

(d) Vertex Pairs for no Bridge Edges

First we prove a few claims -

Claim 1

If there is an edge between vertices of the same equivalence class, then that edge is part of some cycle.

Proof

If the edge was not part of any cycle, then there exists only one path between its endpoints (through the edge itself). This means that the edge is an bridge edge. However, we know that the endpoints of a bridge edge cannot be in the same equivalence class so there cannot be a non-cycle edge between vertices of the same equivalence class.

Claim 2

An equivalence class of $\mathcal{R}$ cannot have exactly 2 vertices.

Proof

Assume that the equivalence class has exactly 2 vertices. Then they must be connected as there should be a path between them. But since there are only 2 vertices, that edge cannot be part of a cycle. Using Claim 1 we can say that these 2 vertices don't form an equivalence class by themselves then.

Claim 3

If all vertices of a tree have degree 1 then the tree has 2 vertices.

Proof

Let number of vertices by n. Then sum of degrees of vertices is $2n$, and as this is twice the number of edges, for a tree this implies that $2(n-1) = n$ or that $n = 2$.

Claim 4

In a tree in which the root has degree at least 2, there exists a path from every vertex to at least 2 leaf vertices.

Proof

We give a constructive proof for the same by explicitly constructing two such paths.
If the vertex is the root, then first add two edges to paths P_1 and P_2 which

are from the root to any 2 of its children (different) then keep adding edges to both the paths such that each edge is from a vertex to its descendant until the descendants are leaves, these two are guaranteed to be different leaves otherwise there will be a cycle which is not possible in a tree.
If the vertex is not the root, then one path can be the one in which edges towards descendants are added till a leaf is reached. The other path can be one in which first follows the ancestors to the root (subpath from vertex to root), and since the root has at least one other child other than the one which lies on the path to root from the vertex, it can follow the descendants of that child till another leaf is reached (subpath from root to another leaf). Concatenating these two paths we get a path which reaches another leaf.

Notation and Definition

If for a graph G and the tree T of its equivalence classes induced by $\mathcal{R}$ (T is defined in the previous part of this problem), if we say that an edge $f = (C_i, C_j)$ is "attached" to T if it is added to the set of edges of T and we also add a corresponding edge $e = (x_i, x_j)$ in the graph G. This edge e is between any two vertices x_1 and x_2 subject to the condition that $x_i \in C_i$ and $x_j \in C_j$.

Claim 5

If a new edge f is attached to tree T then the corresponding edge e added to graph G is also a new edge and did not belong to the edge set of G before.

Proof

Assume that the edge e was already in the edge set of G. This means that this edge connects two vertices in different equivalence classes and hence is a bridge edge. But if e is a bridge edge then its corresponding edge f already existed in T by definition of T (edges corresponding to bridge edges are already present in T). This is a contradiction as f is a new edge in T.

Algorithm

Step 1

Initialize E_0 to a null set. Construct the tree T from the given graph. This takes $\mathcal{O}(m + n)$ (exactly like steps 1 to 3 in the algorithm of part (c)).

Step 2

If T only has 2 vertices C_1 and C_2, then consider the component with larger size. Take a vertex v_1 of G belonging to that component which is NOT an endpoint of the bridge edge joining the components and the vertex v_2 which is the endpoint of the bridge edge joining the two components of the other component. Add edge (v_1, v_2) to E_0 and return. As the sizes in this step are fixed, it does not add to the asymptotic time complexity of the algorithm.

Step 3

If T has more than 2 vertices, then it must have at least one vertex with degree more than 1 (by Claim 3 and the fact that there are no isolated vertices in trees). Let that vertex be called the root of T. Start a DFS on T rooted at the root and take note of all the leaf nodes. Let the set of leaves in T be $L = \{l_1, l_2 \dots l_t\}$. "Attach" edges in the set $F_0 = \{(l_1, l_2), (l_2, l_3) \dots (l_{t-1}, l_t), (l_t, l_1)\}$. We see that these are new edges for the graph T as no two leaves of a tree are connected by an edge. Let the corresponding edges of the attached edges in graph G be added to E_0. Return E_0.
This step takes $\mathcal{O}(n)$ time for the DFS (as it is on a tree) and $\mathcal{O}(n)$ time for attachment. This is because if the number of vertices in G was n, then we saw that the number of vertices can be at most n in T and as there is at least one non-leaf node (the root), the number of edges attached (which was equal to the number of leaves of T) will be $\mathcal{O}(n)$ too.

Proof of Correctness

Proof is divided into two cases - one where T contains only two nodes and the other where it has more than two.

Case 1 - If T only has two nodes, then there are two possibilities, first that either both the nodes have cardinality 1 and the second that at least one has cardinality 3 or more. This is because the nodes of T are actually equivalence classes of vertices of G, and as we saw in Claim 2 an equivalence class cannot have exactly 2 vertices. The first subcase is not possible as the number of nodes is given to be greater than or equal to 3. In the second subcase, it is guaranteed that the component with larger cardinality will have at least 3 vertices in it, and hence at least one vertex which is not the endpoint of the bridge edge. Connecting such a vertex to the endpoint of the bridge edge in the other component no results in no longer having any bridge edge in the graph G, as the earlier bridge edge is now part of a cycle (there existed a path through the bridge edge between the two vertices before and now they also have a direct edge between them). Also as this edge was not already present in the graph (otherwise they would be part of the same equivalence class), we have proved that for this case that $G_0 = (V, E \cup E_0)$ doesn't have any bridge edges as well as the fact that $E \cap E_0$ is a null set. Also only one edge is added in this case, which is less than n (which is at least 3).

Case 2 - If T has more than 2 nodes, then as the edges in the set F_0 are not in the tree T, it follows by Claim 5 that the corresponding edges in the set E_0 are also not part of the edge set E of G. This proves that $E \cap E_0$ is empty. To show that $G_0 = (V, E \cup E_0)$ does not contain any bridge edges, note that adding edges to a graph cannot convert a non-bridge edge to a bridge edge. So, we only have to check for the earlier bridge edges and the new edges we added.

Note that by Claim 6 of the previous part (check NOTE), since the edges of F_0 by their definition form a cycle (which has the vertices which were earlier leaves of T), there exists a corresponding path in G_0 - this can be seen two different paths between vertices of T_0 (let T_0 be the modified tree after adding edges of F_0) and finding the corresponding paths by Claim 6, as the paths are different this means that there exists a cycle and that the edges of the set E_0 added are parts of cycles and hence not bridge edges.
Again consider T_0. If we delete an original edge b that belonged to T from T_0, we can say that it still remains connected. Assume that the deleted edge had end points C_1 and C_2 such that C_1 was the ancestor of C_2. Then there existed a path from C_1 to a leaf of T through the root node, and there also existed a path from C_2 to a leaf node (Claim 4). Now, since in T_0 all the leaves are connected in a cycle, this implies that there exists a path from C_1 to C_2 in T_0 even after deleting the original edge.
Corresponding to this, there will exist a bridge edge a of G upon whose deletion we can say that the graph G_0 is still connected. This is because there will exist a path from the two end points of the bridge edge to vertices in equivalence classes which were leaf nodes of T, and since those are now connected in a cycle, the two vertices can still reach each other. Thus, the edges which were bridges earlier no longer are after the completion of this algorithm. Hence, we have proved that G_0 doesn't have any bridge edges and that $E \cap E_0$ is empty.
Also note that since T can have at most n nodes, out of which 1 is the root, it can have at most $n-1$ leaves. This means that the cardinality of the set F_0 can be at most $n-1$ and since the cardinality of set E_0 is same as that of F_0, at most $n-1$ edges are part of E_0.

Time Complexity Analysis

By the discussion of the time complexities of the individual steps done above in the description of the algorithm, we can say that the running time of this algorithm is $\mathcal{O}(m+n+n)$ which is $\mathcal{O}(m+n)$.