COL351 Assignment 1

Garvit Dhawan, Pratik Nimbalkar

TOTAL POINTS

49 / 60

QUESTION 1

Minimum Spanning Tree 20 pts

1.1 Unique MST 5 / 5

✓ **+ 5 pts** *Correct*

1.2 Fault resilient 15 / 15

✓ **+ 15 pts** *Correct*

QUESTION 2

2 Interval covering 5 / 15

✓ **+ 0 pts** *Incorrect*

+ 5 *Point adjustment*

Partial marks.

1. Out of intervals with same starting point, one with the max ending point should be selected.

2. Even when there are no overlapping intervals, interval with min starting point and max ending point should be selected as next interval in the solution Y.

QUESTION 3

Bridge edges 25 pts

3.1 Transitive relation 2 / 3

✓ **+ 3 pts** *Correct*

- 1 *Point adjustment*

a to c might be a walk on merging paths from a to b and b to c, need to show that path can be constructed by removing extra edges.

3.2 Equivalence classes 5 / 5

✓ **+ 5 pts** *Correct*

3.3 Matrix A 12 / 12

✓ **+ 7 pts** *Correct Algorithm*

✓ **+ 5 pts** *Proof of Correctness*

3.4 Set E0 5 / 5

✓ **+ 3 pts** *Algorithm*

✓ **+ 2 pts** *Proof of Correctness*

COL351 Assignment 1

Garvit Dhawan-2020CS50425

Pratik Nimbalkar-2020CS10607

Minimum Spanning Tree

(a)

We prove this using contradiction.

Let T_1 and T_2 both be different MSTs of graph G.

Let e1=(x,y) be smallest edge such that e1 lies in either T_1 or T_2.

without loss of generality, lets assume, e1 ϵ T_1.

Let p be path (x,y) in T_2.

Now, path(x,y) in T_1 is e1.

If all edges of p are in T_1, then there would be two paths from x to y in T_1, which would be a cycle.

Therefore there must exist e2 ϵ p such that e2 does not belong to T_1.

Now, e2 is also an edge such that e2 lies in either T_1 or T_2.

Therefore, weight(e1)<weight(e2) (Since, e1 is smallest weight edge in either T_1 or T_2)

Hence, if we replace e2 by e1 in T_2, we get a new spanning tree different from T_2(let it beT_3). Such that weight(T_3)<weight(T_2)

(T_3 will be a spanning tree. This was proved in class)

Therefore, T_2 is not MST, which is a contradiction.

Now, T_2 U e1 must have a cycle.

(b)

Algorithm:

1. Compute an MST of G(let it be T).
2. For each edge e(x,y) ϵ E, we check whether e lies in T or not.
    a. If it does not lie in T, then we can say that MST of G-e is same as MST of G, then we can check the next edge.
    b. If it lies in T, then we find DFS(x), DFS(y) in T-e (not in G-e) and while calculating them, we remove every edge we encounter from E'(a copy of E-e).
    Then after this, E' would only contain those edges which connect A, B (where A is the component connected to x and B is the component connected to y in T-e.)
    (This is correct since any edge in A would be covered by DFS(x), thus would be removed from E', and we can argue similarly for edges in B. Also, any edge other than that in A or B would not be removed since, it would not exist in T-e)

1.1 Unique MST **5 / 5**

✓ **+ 5 pts** *Correct*

COL351 Assignment 1

Garvit Dhawan-2020CS50425

Pratik Nimbalkar-2020CS10607

Minimum Spanning Tree

(a)

We prove this using contradiction.

Let T_1 and T_2 both be different MSTs of graph G.

Let e1=(x,y) be smallest edge such that e1 lies in either T_1 or T_2.

without loss of generality, lets assume, e1 ϵ T_1.

Let p be path (x,y) in T_2.

Now, path(x,y) in T_1 is e1.

If all edges of p are in T_1, then there would be two paths from x to y in T_1, which would be a cycle.

Therefore there must exist e2 ϵ p such that e2 does not belong to T_1.

Now, e2 is also an edge such that e2 lies in either T_1 or T_2.

Therefore, weight(e1)<weight(e2) (Since, e1 is smallest weight edge in either T_1 or T_2)

Hence, if we replace e2 by e1 in T_2, we get a new spanning tree different from T_2(let it beT_3). Such that weight(T_3)<weight(T_2)

(T_3 will be a spanning tree. This was proved in class)

Therefore, T_2 is not MST, which is a contradiction.

Now, T_2 U e1 must have a cycle.

(b)

Algorithm:

1. Compute an MST of G(let it be T).
2. For each edge e(x,y) ϵ E, we check whether e lies in T or not.
    a. If it does not lie in T, then we can say that MST of G-e is same as MST of G, then we can check the next edge.
    b. If it lies in T, then we find DFS(x), DFS(y) in T-e (not in G-e) and while calculating them, we remove every edge we encounter from E'(a copy of E-e).
    Then after this, E' would only contain those edges which connect A, B (where A is the component connected to x and B is the component connected to y in T-e.)
    (This is correct since any edge in A would be covered by DFS(x), thus would be removed from E', and we can argue similarly for edges in B. Also, any edge other than that in A or B would not be removed since, it would not exist in T-e)

Now, we check if ∃ an edge in E' with weight equal to that of e.
If yes, then MST of G-e would be same as MST of G and we will check for the next edge, otherwise not so and we will end our algorithm saying that G is not edge-fault-resilient.

After checking all edges, if we have not said that G is not edge-fault-resilient, then we can say that G is edge-fault-resilient, since for all edge e we have seen that we can find MST of G-e same as MST of G.

Correctness Proof: $\in E$.

We have to check whether the MST of G-e is also an MST of G for all edges e

Lets consider 2 cases: one in which the MST of G does not include edge e and the other in which it does include.

For the 1st case, when e is not included in MST of G, we can say the MST of G-e will be same as MST of G

Let T be MST of G, and T does not include e. Then, removing e does not disconnect any vertices in T.

Therefore, in G-e, T will span all vertices.

T will have minimum weight because we have not added any edges, only removed and the edge removed was also not a part of T. If we assume that MST of G-e has weight greater than that of T, then e must have been a part of MST of G, but it is not so. Therefore, by contradiction, we can say that T will have minimum weight.

Hence, T is MST of G-e.

Therefore, MST of G-e is also MST of G.

For the 2nd case, when MST of G includes edge e(x,y), then in G-e we would need to find another path connecting x and y. This is because MST of G-e will definitely not have edge e.

Lets suppose MST of G is T. Now, if we want MST of G-e (let it be T') to be MST of G as well, then

Total weight of T' must be same as total weight of T(if not so, then T and T' will both be MST of G, such that weight of T'< weight of T or weight of T < weight of T'. if the first case holds then T would not be MST of G and if second case holds then T' would not be MST of G. Thus both cases are contradictory. Therefore, weight of T must be equal to weight of T' if both are MSTs of G)

Let A and B be the two connected components in T-e. Then, there must exist edge e' such that e' has the minimum weight of all edges connecting A and B (if there is no edge connecting A and B other than e, then G-e could not have a spanning tree, and if there are such edges, then there would be one of them which would have minimum weight).

(i)

Lets assume weight of e' > weight of e

⇨ Then, weight of T' > weight of T

This is because e' is the minimum weighted edge connecting A and B in G-e. So, any edge in T' connecting A and B must have weight same as e' (otherwise T' would not be MST, by exchange argument),
Therefore, (weight of T' = (weight of A + weight of B + weight of e')) > ((weight of A + weight of B + weight of e) = weight of T)

Hence, there cannot be MST of G-e which can be MST of G (Since all MSTs of G-e would have same weight as T')

(ii)

Lets assume weight of e' <weight of e

⇨ Then if we exchange e by e' in T, T-e U e' would have lesser weight, that is, T would not have been MST(contradiction)

Therefore, such a case would not exist.

(iii)

Lets assume weight of e' = weight of e

Then, weight of T'= weight of T.

This is because e' is the minimum weighted edge connecting A and B. So, any edge in T' connecting A and B must have weight same as e' (otherwise T' would not be MST, by exchange argument),

Therefore, (weight of T' = (weight of A + weight of B + weight of e')) = ((weight of A + weight of B + weight of e) = weight of T)

Therefore, T' will also be MST of G.

Time Complexity:

For each edge in T, we compute DFS(x) and DFS(y) in T-e, which takes O(n) time.

(Since, if x has k vertices, it would have k-1 edges(it is a connected component of a tree)

And y would have n-k vertices and n-k-1 edges(for same reason as that for x)

Thus, DFS(x) would take O(k+(k-1)) time and DFS(y) would take O((n-k)+(n-k-1)) time.

Therefore, together they would take O(k) + O(n-k) = O(n) time)

Now, for checking minimum weighted edge in E', we would take O(m-n-1) time

(since we removed O(n) edges from E to get E' and finding minimum takes linear time)

Therefore, for each edge in T, we would require O(m-n) + O(n) = O(m) time.

And since T has n-1 edges, we would have to do this O(n) times.

For other edges, we need not check.

Therefore, total time complexity:

T(m,n) = O(n)*O(m) = O(m*n)

1.2 Fault resilient **15 / 15**

✓ **+ 15 pts** *Correct*

Interval Covering

Let X={ [si,ei] | 1<=i<=n}, where si, ei are the starting point, ending point of the ith interval respectively.

First, we sort the intervals based on starting point si's. Without loss of generality, lets assume the ordering after sorting is s1<s2<s3….<sn.

Algorithm:

First we define cover(m), which calculates the smallest covering starting from the mth interval.

Cover(m):

Let solution set be S

1. Add [sm,em] to S.
2. For(i = m+1 to n)
   2.1. If (si>=em) break;
   2.2. If (ei<em), then
        i++;
        continue;
   2.3. Else
        break;
3. max=i;
4. for(j = i+1 to n)
   4.1. if(ej>emax && sj<em)
        max=j;
5. Compute Cover(max);
6. Add Cover(max) to S;
7. Return S;

Therefore, Smallest covering of X will be smallest covering starting from the first interval, that is,

Smallest covering(X) = cover(1);

Time Complexity:

Cover(m) computes the smallest covering starting from the mth interval, that is, it calculates, the smallest covering of size (n-m).

Cover(m) computes 2 separate loops which are linear in n, and in each iteration of both loops, we have O(1) operations. Then we compute Cover(max).

Therefore, in worst case,

T(n) = T(n-1) +O(n)+O(n) (since in worst case, max cannot be lesser than m+1)

Where, T(n) = time required to compute smallest covering of set having size n

⇨ T(n) = T(n-1) +O(n)

⇨ Therefore, T(n) = O(n^2)

Therefore, we compute the smallest covering in polynomial time.

Correctness Proof:

If we sort the intervals based on starting point, then we greedily take the first interval.

This is correct greedy approach since the interval [s1,s2] (part of the fist interval before the starting of next interval) cannot lie in any other interval except the first one.

If we get an interval with start value greater than or equal to the end value of first interval, then there can be an interval in between, which does not overlap with either of these intervals. Therefore, before checking such interval, we need to check all those intervals which overlap with the first interval.

We are skipping intervals whose end value is less than end value of first interval, since this interval, is completely covered in the first interval.

Now, for intervals having start value less than the end value of first interval, we find the interval with the maximum end value.

This step is correct since we are taking the interval whose, start value is less than the end value of the first interval, that is, it overlaps with the first interval. Therefore, any interval with end value less than our chosen interval is completely covered in either the first interval or our chosen interval.

If there exist interval j in the optimal solution such that ej<emax, then if we swap ej with emax in our solution, then we get a better solution.

This is because both jth and maxth intervals overlap with the first interval, so all values of jth interval are covered in either the first or the maxth interval. And, the values in the interval [ej,emax] (part of maxth interval not in jth interval) are also added to our solution. **Hence, swapping emax with ej expands our cover without any loss.**

Therefore, using **exchange argument** we can say that including emax will improve our solution

Then, we can recurse on the maxth interval, since for the intervals with start value greater than the maxth interval we can treat maxth interval as their 1st interval with respect to our algorithm.

This is correct since for checking these intervals, we need a particular interval in our algorithm which exists in the solution and either overlaps with these intervals or if not overlapping, then there does not exist any intervals in between our chosen interval and the intervals we are checking. This interval is the maxth interval.

Hence, Cover(m) correctly computes the smallest covering starting from the mth interval(when sorted by starting value).

Therefore, Cover(1) correctly computes the smallest covering starting from the 1st interval, that is the smallest covering of X.

Bridge Edges

(a)

Let aRb and bRc.

2 Interval covering 5 / 15

✓ **+ 0 pts** *Incorrect*

+ 5 *Point adjustment*

Partial marks.

1. Out of intervals with same starting point, one with the max ending point should be selected.

2. Even when there are no overlapping intervals, interval with min starting point and max ending point should be selected as next interval in the solution Y.

⇨ Therefore, T(n) = O(n^2)

Therefore, we compute the smallest covering in polynomial time.

Correctness Proof:

If we sort the intervals based on starting point, then we greedily take the first interval.

This is correct greedy approach since the interval [s1,s2] (part of the fist interval before the starting of next interval) cannot lie in any other interval except the first one.

If we get an interval with start value greater than or equal to the end value of first interval, then there can be an interval in between, which does not overlap with either of these intervals. Therefore, before checking such interval, we need to check all those intervals which overlap with the first interval.

We are skipping intervals whose end value is less than end value of first interval, since this interval, is completely covered in the first interval.

Now, for intervals having start value less than the end value of first interval, we find the interval with the maximum end value.

This step is correct since we are taking the interval whose, start value is less than the end value of the first interval, that is, it overlaps with the first interval. Therefore, any interval with end value less than our chosen interval is completely covered in either the first interval or our chosen interval.

If there exist interval j in the optimal solution such that ej<emax, then if we swap ej with emax in our solution, then we get a better solution.

This is because both jth and maxth intervals overlap with the first interval, so all values of jth interval are covered in either the first or the maxth interval. And, the values in the interval [ej,emax] (part of maxth interval not in jth interval) are also added to our solution. **Hence, swapping emax with ej expands our cover without any loss.**

Therefore, using **exchange argument** we can say that including emax will improve our solution

Then, we can recurse on the maxth interval, since for the intervals with start value greater than the maxth interval we can treat maxth interval as their 1st interval with respect to our algorithm.

This is correct since for checking these intervals, we need a particular interval in our algorithm which exists in the solution and either overlaps with these intervals or if not overlapping, then there does not exist any intervals in between our chosen interval and the intervals we are checking. This interval is the maxth interval.

Hence, Cover(m) correctly computes the smallest covering starting from the mth interval(when sorted by starting value).

Therefore, Cover(1) correctly computes the smallest covering starting from the 1st interval, that is the smallest covering of X.

Bridge Edges

(a)

Let aRb and bRc.

then ∃ a - b path (a,u1,u2...,un,b) such that (ui,uj){forall i!=j,1<=i,j<=n} is not a bridge edge

and

∃ b - c path (b,v1,v2,....,vm,c) such that (vi,vj){forall i!=j,1<=i,j<=m} is not a bridge edge

Let ∃ a - c path (a,u1,u2,.....,un,b,v1,v2,...vm,c) passing through b.

This path has no bridge edge.

Hence aRc.

Therefore, R is a transitive relation since aRb and bRc implied aRc`.

(b)

Definition of equivalence classes - If R is an equivalence relation on A and x ∈ A, then the equivalence class of x, is the set of all elements of A that are related to x, i.e. Equivalence class of x = {y ∈ A | xRy }.

Hence, firstly we need to prove that R is an equivalence relation.

For this we need to prove that R is Reflexive, Transitive as well as Symmetric relation.

Symmetric –

Let aRb. Thus, ∃ a path a-b (a, u1, u2, ...un, b) such that (ui,uj){forall i!=j,1<=i<=n,1<=j<=n} is not a bridge edge.

Since given graph is an undirected graph, then every edge from x to y is same as edge from y to x(for all x,y ∈ V). Hence, we can say that there will exist path b-a (b, un,.....n1,a) which will also not have any bridge edge such edges (ui,uj) {for all i!=j,1<=i,j<=n}. Hence we can say that bRa.

Thus, R is a symmetric relation.

Transitive- proved in part a

Reflexive – Any relation on set X is said to be reflexive if it relates every element of X to itself, i.e. aRa. We can always go to any vertex x from the vertex x itself by not moving anywhere and hence the path from x-x would not contain any bridge edges since we are not covering any edge at all. Hence, aRa.

Thus we can see that R is an equivalence relation.

Now to find the equivalence classes produced by R,

1. we will find all bridge edges using DFS (Depth First Search).

2. Let Set BE be the set containing all bridge edges.

For every edge e ∈ BE= (x,y) we compute DFS x, DFS y in graph (V,E- BE)

All vertices in DFS x will be the equivalence class of x

3.1 Transitive relation **2 / 3**

✓ **+ 3 pts** *Correct*

- 1 *Point adjustment*

a to c might be a walk on merging paths from a to b and b to c, need to show that path can be constructed by removing extra edges.

then ∃ a - b path (a,u1,u2...,un,b) such that (ui,uj){forall i!=j,1<=i,j<=n} is not a bridge edge

and

∃ b - c path (b,v1,v2,....,vm,c) such that (vi,vj){forall i!=j,1<=i,j<=m} is not a bridge edge

Let ∃ a - c path (a,u1,u2,.....,un,b,v1,v2,...vm,c) passing through b.

This path has no bridge edge.

Hence aRc.

Therefore, R is a transitive relation since aRb and bRc implied aRc`.

(b)

Definition of equivalence classes - If R is an equivalence relation on A and x ∈ A, then the equivalence class of x, is the set of all elements of A that are related to x, i.e. Equivalence class of x = {y ∈ A | xRy }.

Hence, firstly we need to prove that R is an equivalence relation.

For this we need to prove that R is Reflexive, Transitive as well as Symmetric relation.

Symmetric –

Let aRb. Thus, ∃ a path a-b (a, u1, u2, ...un, b) such that (ui,uj){forall i!=j,1<=i<=n,1<=j<=n} is not a bridge edge.

Since given graph is an undirected graph, then every edge from x to y is same as edge from y to x(for all x,y ∈ V). Hence, we can say that there will exist path b-a (b, un,.....n1,a) which will also not have any bridge edge such edges (ui,uj) {for all i!=j,1<=i,j<=n}. Hence we can say that bRa.

Thus, R is a symmetric relation.

Transitive- proved in part a

Reflexive – Any relation on set X is said to be reflexive if it relates every element of X to itself, i.e. aRa. We can always go to any vertex x from the vertex x itself by not moving anywhere and hence the path from x-x would not contain any bridge edges since we are not covering any edge at all. Hence, aRa.

Thus we can see that R is an equivalence relation.

Now to find the equivalence classes produced by R,

1. we will find all bridge edges using DFS (Depth First Search).

2. Let Set BE be the set containing all bridge edges.

For every edge e ∈ BE= (x,y) we compute DFS x, DFS y in graph (V,E- BE)

All vertices in DFS x will be the equivalence class of x

Correctness Proof:

In graph G'=(V,E- BE), there are no bridge edges. Hence if ∃ path x – y ∈ G' then that path contains no bridge edges. Hence xRy. Now, DFS(x) in G' will cover all vertices v such that ∃ a path x to v. Therefore for all v ∈ DFS(x) in G' xRv. Therefore all vertices of DFS(x) in G' belong to equivalence class of x.

Let there be bridge edge a – b. We have proved that all vertices of DFS(a) in G' and DFS(b) in G' belong to equivalence classes of a and b respectively.

Let equivalence class of A be Ea and that of B be Eb.

Now we prove that all vertices of DFS(a) are the only vertices that belong to equivalence class of A, i.e. Ea ∩ Eb is NULL.

Let b' belong to Eb and let a' belong to Ea. ∃ path a to a' and path b to b' but edge a-b is a bridge edge. Let ∃ path P from a' to b' such that P does not include a -b. Therefore we can find path a – a' and path b - b' (since aRa' and bRb'). Therefore we can find path a -b (a – a' – b' -b) such that this path does not contain edge a – b. Therefore, a – b is not a bridge edge which is a contradiction. Therefore all paths from a' – b' contain bridge edge a – b. Therefore, (a',b') does not belong to R.

This is true for all a', b' belong to DFS(a), DFS(b). Therefore, Ea ∩ Eb is NULL. Hence the only vertices in equivalence class of A are the ones covered by DFS(a) in G'.

Hence proved.

Time Complexity Analysis:

In 1, we can calculate all bridge edges in O(m+n) time.

In 2, In all the steps we are computing the DFS for a subset of vertices. Thus, we would cover all edges and vertices only twice at max.

For example, there is only one bridge edge e=(x,y). Thus, in G-e, we would have 2 connected components. Let first component have k vertices, l edges. Then, the other one will have (n-k) vertices and (m-l-1) edges.

Thus, for computing DFS(x) we would require O(k+l) time, and for DFS(y), we need O(n-k+m-l-1) time.

Therefore, in total we need O(k+l) + O(n-k+m-l-1) = O(n+m) time complexity.

This, argument can be extended to any number of bridge edges.

Therefore, overall we need O(m+n) + O(m+n) = O(m+n) time.

(c)

Algorithm:

Here, A is the witness matrix we are filling, and we are assuming that for any 2 vertices x,y, A(x,y) is -1 if x and y belong to the same equivalence class

1. Compute set BE of all bridge edges using DFS of G (discussed in class).

3.2 Equivalence classes **5 / 5**

✓ **+ 5 pts** *Correct*

Correctness Proof:

In graph G'=(V,E- BE), there are no bridge edges. Hence if ∃ path x – y ∈ G' then that path contains no bridge edges. Hence xRy. Now, DFS(x) in G' will cover all vertices v such that ∃ a path x to v. Therefore for all v ∈ DFS(x) in G' xRv. Therefore all vertices of DFS(x) in G' belong to equivalence class of x.

Let there be bridge edge a – b. We have proved that all vertices of DFS(a) in G' and DFS(b) in G' belong to equivalence classes of a and b respectively.

Let equivalence class of A be Ea and that of B be Eb.

Now we prove that all vertices of DFS(a) are the only vertices that belong to equivalence class of A, i.e. Ea ∩ Eb is NULL.

Let b' belong to Eb and let a' belong to Ea. ∃ path a to a' and path b to b' but edge a-b is a bridge edge. Let ∃ path P from a' to b' such that P does not include a -b. Therefore we can find path a – a' and path b - b' (since aRa' and bRb'). Therefore we can find path a -b (a – a' – b' -b) such that this path does not contain edge a – b. Therefore, a – b is not a bridge edge which is a contradiction. Therefore all paths from a' – b' contain bridge edge a – b. Therefore, (a',b') does not belong to R.

This is true for all a', b' belong to DFS(a), DFS(b). Therefore, Ea ∩ Eb is NULL. Hence the only vertices in equivalence class of A are the ones covered by DFS(a) in G'.

Hence proved.

Time Complexity Analysis:

In 1, we can calculate all bridge edges in O(m+n) time.

In 2, In all the steps we are computing the DFS for a subset of vertices. Thus, we would cover all edges and vertices only twice at max.

For example, there is only one bridge edge e=(x,y). Thus, in G-e, we would have 2 connected components. Let first component have k vertices, l edges. Then, the other one will have (n-k) vertices and (m-l-1) edges.

Thus, for computing DFS(x) we would require O(k+l) time, and for DFS(y), we need O(n-k+m-l-1) time.

Therefore, in total we need O(k+l) + O(n-k+m-l-1) = O(n+m) time complexity.

This, argument can be extended to any number of bridge edges.

Therefore, overall we need O(m+n) + O(m+n) = O(m+n) time.

(c)

Algorithm:

Here, A is the witness matrix we are filling, and we are assuming that for any 2 vertices x,y, A(x,y) is -1 if x and y belong to the same equivalence class

1. Compute set BE of all bridge edges using DFS of G (discussed in class).

2. For each bridge edge (a,b) in BE, we compute DFS(a) and DFS(b) in G-BE(from previous part we know that this will give us equivalence class of a, b respectively). After computing DFS(a), we will remove the equivalence class of a from the set of vertices, so that it does not get computed twice if there is another bridge edge connected to the class.
3. While computing DFS(x) in G-BE, for every vertex covered, we store a pointer to x(making x a representative vertex of its equivalence class).
4. Let G' = {V_{BE},E'}, where V_{BE} is the set of roots of equivalence classes computed(j is a root of equivalence class if that particular equivalence class is calculated by taking DFS(j) in G-BE).

   Each (vi, vj) (vi, vj ∈ V_{BE}) is an edge in E' if ∃ a bridge edge connecting equivalence classes of vi, vj.

   Now, for each vertex v in V_{BE} , we find DFS of v in G' and for each vertex v' in DFS of v, we set A(v,v,') = A(v',v) = (v'',v) , where v'' is the parent of v' in DFS(v)
5. Now, for any 2 vertices x, y in G, we compute find(x) and find(y) (where, find(m) is the vertex n pointed by the pointer stored with m while computing DFS(n), that is find(m) returns the representative vertex of the equivalence class to which m belongs)
6. If find(x) = find(y), we set A(x,y)=A(y,x) = -1
7. Else we set A(x,y)=A(y,x)= A(find(x),find(y))

Correctness Proof:

Our first step correctly gives us the set BE of all bridge edges in G (discussed in class).

Our second step correctly finds the equivalence classes (proved in part b)

Now, for any 2 vertices x,y not related under R, there will exist a bridge edge in all paths form x to y

Let x belong to equivalence class of a , y belong to equivalence class of b, then if we find bridge edge e connecting a,b then we can say that A(x,y) = e (since, if we remove e, there is no path from a to b, since e is a bridge edge, and a is connected to x, b is connected to y. So, there does not exist any path from x to y if we remove e. If there were a path existing from x to y after removing e, then there would exist a path from a to b which would not include e, then e would not be a bridge edge which would be a contradiction). Therefore, bridge edge e will exist in all paths from x to y.

Therefore, while computing equivalence class of lets say v, if we add a pointer to v from each vertex in equivalence class of v, then for finding bridge edge between any 2 vertices of equivalence classes v, v' we can just find bridge edge from v to v' and that would be our required bridge edge.

Now, to find bridge edges between vi such that vi are roots of equivalence class (j is a root of equivalence class if that particular equivalence class is calculated by taking DFS(j) in G-BE), we take equivalence class of vi as a single vertex and the bridge edges connecting two equivalence classes as edges connecting these vertices(which are equivalence classes considered as a single vertex).

Now in this graph ({vi | vi is root of equivalence class},BE), if we compute DFS, then we can find a bridge edge connecting any two vertices vi, vj

This is correct because any 2 equivalence classes are joined by only bridge edges.

Hence, our algorithm is correct.

Time Complexity:

Time to calculate BE = time to calculate DFS of G = O(m+n)

Time to calculate all equivalence classes = O(m+n) (proved in part b)

(adding pointer to representative vertex for every vertex takes only O(1) time, so total time complexity remains same)

Now, to find bridge edge between v and v' in step 4, for each v, calculating v' will take O(n+(n-1)) time (since the edges of G' are bridge edges, and number of bridge edges <=n-1(proved in classs))

In worst case, we do this n times. Therefore, total time required = O(n)*O(n) = O(n*n)

Time required to calculate find(x) and find(y) = O(1) (as discussed in class, the find function takes constant time to calculate the representative vertex, since we directly get the representative vertex through the pointer).

Therefore, to fill each position in the matrix A, we need O(1) time and there are n*n positions. Therefore, total time to fill matrix = O(1)*O(n*n)

Hence total time Complexity = O(m+n) + O(m+n) + O(n*n) + O(1)*O(n*n) = O(n*n). (since O(m) <=O(n*n) in worst case)

(d)

Let (a,b) be bridge edge and Ea, Eb be equivalence class of a, b respectively. Then let there be set S {a', b' such that for all (a' ϵ Ea – a and b' ϵ Eb) or (a' ϵ Ea and b' ϵ Eb -b)}, then S = E0.

Correctness Proof:

a', b' does not belong to E (since they belong to different equivalence classes and They are only connected through a bridge edge which does not include in our set). Therefore S ∩ E is NULL.

For each bridge edge (a,b) in G we can find alternate path P in graph G' = (V, E U S) connecting a, b such that P does not include edge (a,b)(p= a-a'-b'-b, where edge a',b' is present in G but not in G'). Therefore in G', (a,b) will no longer be a bridge edge. Thus, there are no bridge edges in G'. Hence, (V,E U S) contains no bridge edge.

Since S satisfies both conditions for E0 ,we can say that S=E0.

Algorithm and Time Complexity:

For each bridge edge (a,b), we can find calculate DFS(a), DFS(b) in G'(where G' = (V,E-BE), BE = set of all bridge edges in G) and take any two of their vertices(one from each DFS, except the pair(a,b)) and include that pair in our solution.

For simplicity, we can take the first child from each DFS.

From part b, we know that we can compute the set of all bridge edges and all DFS in G' in O(m+n) time.

Therefore, we can compute E0 in O(m+n) time

3.3 Matrix A **12 / 12**

✓ **+ 7 pts** *Correct Algorithm*

✓ **+ 5 pts** *Proof of Correctness*

Time to calculate BE = time to calculate DFS of G = O(m+n)

Time to calculate all equivalence classes = O(m+n) (proved in part b)

(adding pointer to representative vertex for every vertex takes only O(1) time, so total time complexity remains same)

Now, to find bridge edge between v and v' in step 4, for each v, calculating v' will take O(n+(n-1)) time (since the edges of G' are bridge edges, and number of bridge edges <=n-1(proved in classs))

In worst case, we do this n times. Therefore, total time required = O(n)*O(n) = O(n*n)

Time required to calculate find(x) and find(y) = O(1) (as discussed in class, the find function takes constant time to calculate the representative vertex, since we directly get the representative vertex through the pointer).

Therefore, to fill each position in the matrix A, we need O(1) time and there are n*n positions. Therefore, total time to fill matrix = O(1)*O(n*n)

Hence total time Complexity = O(m+n) + O(m+n) + O(n*n) + O(1)*O(n*n) = O(n*n). (since O(m) <=O(n*n) in worst case)

(d)

Let (a,b) be bridge edge and Ea, Eb be equivalence class of a, b respectively. Then let there be set S {a', b' such that for all (a' ϵ Ea – a and b' ϵ Eb) or (a' ϵ Ea and b' ϵ Eb -b)}, then S = E0.

Correctness Proof:

a', b' does not belong to E (since they belong to different equivalence classes and They are only connected through a bridge edge which does not include in our set). Therefore S ∩ E is NULL.

For each bridge edge (a,b) in G we can find alternate path P in graph G' = (V, E U S) connecting a, b such that P does not include edge (a,b)(p= a-a'-b'-b, where edge a',b' is present in G but not in G'). Therefore in G', (a,b) will no longer be a bridge edge. Thus, there are no bridge edges in G'. Hence, (V,E U S) contains no bridge edge.

Since S satisfies both conditions for E0 ,we can say that S=E0.

Algorithm and Time Complexity:

For each bridge edge (a,b), we can find calculate DFS(a), DFS(b) in G'(where G' = (V,E-BE), BE = set of all bridge edges in G) and take any two of their vertices(one from each DFS, except the pair(a,b)) and include that pair in our solution.

For simplicity, we can take the first child from each DFS.

From part b, we know that we can compute the set of all bridge edges and all DFS in G' in O(m+n) time.

Therefore, we can compute E0 in O(m+n) time

3.4 Set E0 **5 / 5**

✓ **+ 3 pts** *Algorithm*

✓ **+ 2 pts** *Proof of Correctness*