Vatsal Jingar (2020CS50449)
Stitiprajna Sahoo (2020CS10394)

Assignment 1
COL351 (Analysis and Design of Algorithms), Semester-1 (2022-23)

Note : For proving the correctness of algorithms, we have provided many subclaims and claims in every section (For eg. Algorithm, Idea, or Seperate) which would help in building up the proof of correctness.

1 MINIMUM SPANNING TREE

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

1. **Sub-claim 1.1**: *A tree can be considered as 2 connected components C_1 and C_2, with an edge e connecting these 2 components.*
   If C_1 and C_2 have more than one edge (e and e') connecting them, it will form a cycle. This can be proved by the fact that, vertex $u \in C_1$ and $v \in C_2$ can be traversed by 2 paths P_1 and P_2 such $e \in P_1$ and $e' \in P_2$. So traversing from u to v via P_1 and back from v to u via P_2 will hence create a cycle.
2. **Sub-claim 1.2** *A tree can be thus considered to be 2 connected components C_1 and C_2, joined by an edge e around any 2 vertices $v_1 \in C_1$ and $v_2 \in C_2$, where the edge e connecting C_1 and C_2 may not necessarily be the edge between v_1 and v_2.*
3. **Sub-claim 1.3** *We can consider a graph G to be a tree, if it has 2 connected components around any 2 vertices, connected by only 1 edge.*
   Sub-claim 1.3 is a result of *sub-claim 1.1* and *subclaim 1.2*

Note:

1.1 Q: Prove that G has a unique MST if all edge weights in G are distinct

Solution:

- Let us assume, we have 2 different Minimum Spanning Trees T and T' for a graph G with distinct edge weights. $\implies$ T$\backslash T'$ is a non-zero set whose least weighted edge is e.
- By *sub-claim 1.3*, adding edge e to T' will create a cycle. Correspondingly, there exists an edge $e' \in T'$ that is not a part of T, otherwise it would create a cycle. Replacing e' with e would still be a spanning tree for the graph G. *(By using cut property)*
- Let this new spanning tree be $T'' = T' - e' + e$. Without loss of generality, if we consider $weight(e) < weight(e')$ (as all edge weights are distinct), then sum of weights(T'') $<$ sum of weights(T').
- But as per our previous assumption, T' is a Minimum Spanning Tree, so for any other spanning tree, sum of weights(T'') $>=$ sum of weights(T'). So there is a contradiction to our assumption that there can be 2 different Minimum Spanning Trees T and T' for a graph G with distinct edge weights.
- Hence, this proves the uniqueness of Minimum Spanning Tree for a distinct weighted edges graph by contradiction.

1.2 Q: A graph G = (V, E) is said to be edge-fault-resilient if the following condition holds: "For each edge e ϵ 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:

- **Claim 1** : If edge e is not one of the heaviest for every cycle then the edge appears in every MST (And uniquely weighted).
  - **Proof**:
  - Consider a MST of G without e. If we add edge e in that MST, it will create cycle C. As edge e is not a heaviest edge for any of the cycles formed by it. It is always possible to swap this edge with any other edge. So, here, we can replace edge e in place of another e'.
  - **Subclaim** : Replacing an edge with any other edge in cycle also creates a spanning tree. Consider the MST as two acyclic connected components after removing edge e'. Now we can join them using edge e. Now both the connected components are joined, and due to single edge there is no cycle formation also. So the replacement of edge creates a spanning tree.
  - Let the tree T = MST(G) $- e'$ + e
  - e' is the edge with weight greater than e. As edge e is not one of the heaviest edge for all the cycles it is always possible to choose such an edge e'.
  - If wt(e) $<$ wt(e'), wt(T) $<$ wt(MST(G)). So this is a contradiction. Hence proved.
  - Critical Case, if the cycle have same weighted edge and not heaviest then all such edges should be taken in account. (Can be proved by similar contradiction argument)
- **Claim 2** : If an edge occurs in every MST, then it is not one of the heaviest edge for any of the cycles that pass through it.
  - **Proof**:
  - Consider an edge e, that occurs in all MSTs of graph G.
  - Consider any arbitrary MST, T of graph G.
  - Suppose we swap this edge with another edge present in any cycle in which this edge occur. We can swap the edges always as the edge e occurs in all MSTs.
  - If this edge would be an edge with heaviest weight, we can consider the new tree formed after the swapping be T'.
  - The swapped edge has weight less than the weight of e.
  - So, the weight of T' will be lesser than the MST. As we have assumed that the tree T is MST of graph G. This will make a contradiction.
  - **Critical Case** : If every edge available for swap has weight equal to the weight of the edge e, then it will produce a different MST.
- **Sub-Claim** : Every neighbour of vertex x, will be descendant for the root node x in DFS(x).
  - **Proof**: Proven in Class. (Lecture Slide 7)

  In this way, it is proven that the above claim 1 and 2 creates a necessary and sufficient condition for edges to present in every MST.
  The contra-positive statement of this statement, that
  If an edge is one of the heaviest edge for any one cycle, it will be not necessary that it will be present in every MST.
- **Idea** :
  - Step 1 : Create a MST, using *Kruskal's algorithm*.
  - Step 2 : For all edges not present in MST, take every edge one by one and add it to the MST.
  - Step 3 : For every edge, it can have weight heaviest or equal to some of the edge in cycle.
  - Step 4 : As if it is not heaviest, then we can prove by contradiction that we can replace it with any edge creating a less weighted tree and yet the considered tree is MST, we can not have weight less than MST.

- Step 5 : If the edge have weight equal to some other edge or heaviest, consider these both edges as not necessary.
- Step 6 : This is because for same weighted they are interchangeable thus creating a different MST not involving the initial edge in it.
- Step 7 : For the edge with heaviest weight for that cycle, it would not be considered in that tree, similar proof by contradiction.
- Step 8 : As we know that the MST will have edges n-1.
- Step 9 : Now, we have checked edges greater than equal to m-n-1.
- Step 10 : Now the remaining edges, select every edge, and for every edge take one of its end point, remove that edge and run a DFS on the graph with considering only those having weight lesser or equal to weight of edge removed.
- Step 11 : If the DFS can reach another point, this means for some cycle we have edge e as heaviest edge and hence it is not necessary for all MSTs.
- Step 12 : Else if it can not reach there then there does not exist any cycle where e is one of the heaviest edge and we have also allowed to traverse over same weighted edge, we will handle trivial cases also.
- Step 13 : Do this for all remained edges.

- **Algorithm** :
  - T = Kruskal(G)
  - for edge = (u,v) in T
  - DFS(u), with edges having weight lighter or same, by removing (u,v)
  - if v ϵ DFS(u)
    continue;
    else
    return false;
  - return true;

- **Time Complexity** :
  - Generating Kruskal MST, will be bounded by O(ElogE) which is O(EV) i.e O(mn)
  - Performing DFS for every edge in T will take, O((n-1)*(m+n)) which is O(mn + n2) as G is connected graph, n is bounded by O(m) because for being a connected graph we know m >= n-1.
  - So, this step take O(mn)
  - Total = O(mn)

- **Proof of Correctness** :
  - Proof of Kruskal was provided in class, hence we are assuming it will provide a correct MST.
  - All the edges not included in Kruskal MST are not necessary as they are not in this tree.
  - For remaining edges that are edges of the tree, we can perform modified DFS, where we can get a path if there exist a cycle for which this is one of the heaviest or have same weighted edges in the cycle, which means they are also not necessary.
  - If such a path exists, it can mean either of the following:
    * The edge is heaviest for some of the cycles it is present in, which proves it can not be present in all MST. *This is the contrapositive of Claim 1, so hence proved*
    * All the edges of some cycle have same weight, which means any edge can be replaced by any other edge of the cycle in the spanning tree and still would result in a MST, due to same weight of the tree.*(trivial)*
  - If no such path exists, it is not heaviest for any cycle nor all edges of any cycle have same weight, *proven by claim 2*.
  - Same procedure is done for all the edges, hence whole graph is covered.
  - By the above claims and statements, it is sufficient for proving that the algorithm will provide correctly provide the necessity of every edge to be present in every MST.
  - Hence Proved.

2 INTERNAL COVERING

Q: Let X be a set of n intervals on the real line. A subset of intervals Y $\subseteq$ X is called a covering if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y . Present a polynomial-time greedy algorithm to compute the smallest covering of X. Prove the correctness using exchange argument.

Solution:

Algorithm (Idea):

- Let us assume that there exists a solution for this problem, where #total intervals=n.
- By greedy approach, we will sort the intervals according to their starting points.
- Initially, pick the interval with the least start point and farthest end point let this interval be I_m, remove it from the set and add it to the internal covering set, let's say V.
- Remove the intervals from S, whose end points are less than equal to that of the end point of I_m. (Removal step)
- Now, consider all the intervals whose start time is less than or equal to that of finish time of I_m (if exists). Take the interval with maximum finish time as the new I_m and again remove all the intervals having finish time less than equal to that of chosen interval.
- If no such interval exists, choose an interval with minimum start time from S having maximum length as new I_m and do the removal step.
- Repeat this step recursively until all intervals have been removed from the set.
- The final obtained set covering V will be the optimal(minimal) set covering.

Algorithm:

- Sort the intervals to $S = I_1, I_2, I_3, ...I_n$ where $start(I_1) \leq start(I_2) \leq start(I_3)... \leq start(I_n)$.
- Pick interval I_m with start(I_m)=start(I_1) and end(I_m) is maximum.
- $V = V \cup I_m$
- Until S is empty, run the following loop:
  - $S = S \setminus I : I = \{I_1, I_2, ..I_r\}$ where $\forall k \in \{1, r\}$, end(I_k)$\leq$end(I_m).
  - $N = I \in S$ such that, start(I) $\leq$ finish(I_m), where I_m is the previous chosen interval S, such that end-point of I is maximum in N.
  - If $N = \phi$, I = interval with minimum start point in S. If multiple such intervals exist, pick the one with maximum end point.
  - $V = V \cup I$

Time Complexity Analysis:

- The sorting step of the algorithm will be of the order of $O(nlogn)$ (as given it is real line we can not use bucket sorting).
- Then for finding the interval with required start time and largest finish time will bound to $O(n)$, for every interval added to V, (which can be maximum order of n). This step accounts to a time complexity bound of ($O(n) \times O(n) = O(n^2)$).
- For every interval I_m added to V (order of $O(n)$), we need to remove the intervals with end-points <= end-point(I_m), which will be of order of $O(n)$. This step again accounts to ($O(n) \times O(n) = O(n^2)$).
- Net bound of time complexity =

$$O(nlogn) + O(n^2) + O(n^2) = O(n^2)$$

Correctness Proof

- *Proof by exchange argument*:

- Consider an optimal solution of this problem as O = [v1,v2,v3,..vm]
- Let the solution set returned by our algorithm is A = [i1,i2,i3,...ik]
- As we have to cover all the intervals, v1 or i1 both would have start time equal to the interval of minimum start time.
- We know, that our algorithm will take the interval having max length. So, $finish(v_1) \leq finish(i_1)$.
- We know that $|O| \leq |A|$.
- Let us assume that for all t, $finish(v_t) \leq finish(i_t)$, Now we will prove that $finish(v_{t+1}) \leq finish(i_{t+1})$.
- Let the set after removal of the intervals if we have considered i_t for covering be S. After removal of intervals fully overlapping with i_t i.e having finish time less than or equal to of i_t, S will have intervals having finish time greater than i_t.
- If S has such elements, V = that have start time less than equal to that of i_t, then i_{t+1} must take a interval from V otherwise it will not be able to cover some of the intervals.
- In this case, our algorithm will also take interval i_{t+1} from V, and by the approach of our greedy algorithm, it will always take max length interval from V, so we can say that $finish(v_{t+1}) \leq finish(i_{t+1})$
- If V is a null set than also our algorithm will take value with max length so for second case also, $finish(v_{t+1}) \leq finish(i_{t+1})$.
- Now, we can say that till m, we have finish time of i_m greater than equal to that of v_m.
- Now for our solution to be optimal we need to show $k = m$.
- As O is an optimal solution, after including v_m, S becomes empty.
- As finish(i_m)≥finish(v_m), i_m will also cover the intervals that can be covered by v_m, and as by selecting v_m, the set S is empty same goes with i_m.
- Hence no further elements in A are needed for covering. Hence $k = m$.
- Hence proved, A is also an optimal solution.

3 BRIDGE EDGES

Let G = (V, E) be an undirected connected graph with $n > 3$ vertices and m edges. Define a relation R on V as follows: xRy iff there exists an xy path in G not containing bridge edges.

- **Sub-claim 3.1:** *Suppose the path from x to y has edge set E and vertex set V such that removing any edge from E, there will still exist a path from x to y, implies that E contains no bridge edges.*
  - **Proof:** This can be proved as: Pick any edge $e \in E$,(without loss of generality) suppose its end points are u, v.
  - As removing e still has a path from x to y, we can still trace a path from u to v as from u to x to y to v, here x to y via the other path.
  - $\implies$ $\forall$e $\in E$, removing e will still have a path from u to v, which means all edges in E are non-bridge edges.
- **Sub-claim 3.2:** *Suppose the path from x to y has edge set E and vertex set V such that path between them contains no bridge edges, removing any edge from x to y will result in still a path remaining from x to y.*
  - **Proof:** Remove any edge from E, its end points u, v always have another path p' between them because of the non-bridge property of the edge e.
  - Therefore x, y remain connected and a path exists between them (via p').

3.1 Q: Prove that R is a transitive relation.

Solution:

- **Claim:** *If xRy and yRz, then xRz.*
  - xRy: There is path P_1 containing edgeset E_1 along x to y contains no bridge edges. Similarly, yRz: there is path P_2 containing edgeset E_1 from y to z contains no bridge edges.
  - Now, $P = P_1 + P_2$ is a path from x to z. Now, removing any edge from E_1 will still have a path from x to y, by *sub-claim 3.2*. Similarly, removing any edge from y to z still has a path from y to z.
  - Therefore, if we remove any edge along the path P, there will always remain a path from x to y as well as y to z, hence there exists always a path from x to z too, via y.
  - Therefore, from *sub-claim 3.1*, path P contains no bridge edges. Hence proved transitive relation is satisfied by R, i.e. xRy and yRz $\implies$ xRz.

3.2 Q: Show that the equivalence classes induced by R are computable in O(m+n) time.

Solution:

- **Sub-claim 3.2.1:** *A bridge edge present in a connected component of a graph divides it into 2 different connected components.*
  - Proof:
  - We can assume any connected component as two connected components m_1, m_2 which are connected by a bridge edge e. If we remove this bridge edge, there is no path connecting m_1 and m_2.
  - There can be only 1 edge connecting m_1 and m_2 i.e. the bridge edge e. this can be proved as: If another edge e' would exist between m_1 and m_2, then for any vertices connected by e, removing e would still have a path between those vertices via e', and so e can't be a bridge edge. hence contradiction occurs.
  - Hence Sub-Claim 3.2.1 proved.
- **Sub-claim 3.2.2:** *A bridge edge divide k connected components into k+1 connected components.*
  - Proof:
  - Let the bridge edge be present in m^{th} component. And if remove this bridge edge then this component is divided in two components, as this was the only edge connecting the 2 components. So total there will be k+1 connected components.
  - Hence Sub-Claim 3.2.2 proved.
- **Idea**
  - Run a DFS on graph G.
  - On DFS tree, recurse till you reach leaf nodes, update the highest point of child nodes that is the back edges to ancestors, but do not consider the parent to child edges.
  - If highest point do not exist let it be null. (we will assume null to be less than any level).
  - Let's consider the edge between leaf's and its parent as e_1.
  - If highest point of any leaf node is less than level of parent, then e will not make bridge edge else if leaf node has highest point greater than equal to that of parent level, then e is bridge edge. (checking process)
  - Now recurse up. While recursing up, for all nodes, update the highest point of its parents, which would be minimum(value of all highest points of children,level of ancestor to whom if there is back edge from parent node).
  - HP(node) = min(HP(child1), HP(child2), HP(child3)... HP(childn) , level of ancestor to which if there exist a back edge from the node.)
  - Repeat the checking process for this node too.
  - If there exist any bridge edge, add both of its vertices in a set S.
  - After the recursion is over, remove all the bridge edges from graph G.
  - Run a DFS on every un-visited vertex from set S, and all the edges covered by it, will contain a set of edges containing no bridge edges between them, consider them together as one connected component C.
  - Some vertices can occur in C also so remove them set S, i.e. mark it as visited.
  - Repeat this step till S is null.
  - All the vertices in one component are in one equivalence class.
  - Return these connected components.
- **Algorithm**
  - E = ϕ
  - T = DFS(G)
  - HighestPoint(x) =

- HighestPoint(T1) + HighestPoint(T2) + ... + HighestPoint(TN)
- If(min(All HP of chilren), Backedgelevel(x))) >= Level(x)
- S := S + x + parent(x)
- Remove (x,parent(x)) from G

- For x in S , i =0:
- $T_{partial}$ = DFS(x)
- Set x := visited
- $C_i := T_{partial}$
- E := E + C_i, i := i + 1

- return E

- **Time Complexity**
  - DFS of G = O(m+n)
  - Recursive approach for calculation of bridge = O(m)
  - For all the DFS run for the x in S :
  - Let for one DFS(x_i) be O(m_i + n_i)
  - For all such x it is sum of all such O(m_i + n_i).
  - Sum of m_i = m and Sum of all n_i = n.
  - Total is bounded by O(m+n).
- **Proof of Correctness**
  - Proof of correctness will include
    - Proving correctness of whether the highest point calculated is correct or not.
    - The equivalence classes are connected components.
  - The relation R is equivalent, as if x to y does not have bridge edge, then (as graph is undirected) y to x will also not have path between it. Also no bridge edges are there for x to x (Reflexive relation).
  - Transitive property is proved above in 3.1 for a connected graph.
  - Claim 1 : *Equivalence classes are connected components.*
    - Proof : According to the definition of equivalence classes every element in that class should be related to any other element with the relation R
    - For any element in connected component C, x will have path to all other vertices, this is because they are present in one connected component.
    - As we have removed all the bridge edges present in the graph there is no bridge edge in the path of x to any other vertices in connected component.
    - Also, if two vertices are in different component, they will always need a bridge edge to make a path between them. Hence if two vertices are present in different connected components they do not satisfy the relation.
    - Hence Equivalent classes are connected components.
  - According to the definition of Highest point of graph, it is defined recursively that is the minimum level at which its back edge or its children back edge can reach. So the recursive algorithm will always provide the correct highest point.
  - This is can be proved by induction on nodes. Base Case : For leaf nodes they have no children so their back edge decide highest point. Let the algorithm gives correct highest point for level n, we will prove it will also provide the correct hp for level n-1.
  - As the definition of Hp, it will minimum of all the children hp at level n, and own back-edge level. And algorithm does same. Hence proved.
  - If level(x) is lesser than or equal to hp of its children, then if x and parent(x) edge is removed there is no way to reach to parent(X) from x, as all the children can max reach up to level(x).
  - Removing all the bridge edges will make connected components (Proven by Sub-claim 3.2.1 and Sub-Claim 3.2.2).
  - Run a DFS on every bridge end, this will return a connected component. Re-cursing same algorithm will return connected component for remaining vertices. DFS will not able to reach other connected components as the bridge edge is removed and we know DFS visits all the connected vertices. (Proven in class).

3.3 Q: A matrix A of $n \times n$ size is called a witness matrix for G if for each $x, y \epsilon V$ unrelated under relation R, A(x, y) stores a bridge edge e satisfying that all the $x - y$ paths in G contain e. Design an O(n^2) time algorithm to compute a witness matrix for G.

Solution:

- Idea
  - This idea extends the idea of the previous question for querying any two vertex bridge edges.
  - In this algorithm, we will return only one one bridge edge but we can also return all bridge edges for given two vertices.
  - In previous question we have found out the connected components. And the proof for the same is given in that. Here we will proof the claims needed for further extension and the concise idea for querying.
  - We have found out the set S containing all the bridge edges of the graph. Now start a DFS of graph G, and whenever you reach the bridge edge, do not consider it for traversal, when the DFS will stop, store this tree as one connected component. Let the set C = $[C_1, C_2, ...C_k]$ denotes the all the connected components. For all the components store the bridge edge to whom they are connected. For all the vertices store the connected component in which they are present.
  - Now consider all the connected components as nodes and bridges to be edges connecting them.
  - Claim 1 : Graph formed by considering this connected components as nodes and bridge edges as edges will be tree.
  - Claim 2 : The bridge edges can atmost be n-1.
  - So the cardinality of edge set for this new tree T, will be atmost n-1.
  - Run the DFS on this component tree T. Now for every vertex, Do a DFS on this tree while storing the bridge edges needed for each node. (Start the DFS from the component in which the vertex is present.)
  - Now if the DFS is run from vertex v, which is present in C1, and let e1 is the bridge needed for reaching to C2. Then for all vertices of C2, store the bridge edge e1 for reaching from v. In this way we will create a matrix where for every row, which will denote the vertex we will run DFS on component in which this ith vertex is present. If this jth vertex is present in C1, then for all nodes of C1, M[i][j] will be 0 else if jth vertex is present in different component then for all vertices of C2, M[i][j] will take the value needed for reaching C2 from C1.
  - Fill the matrix in the above way.
- Algorithm
  - M = Witness Matrix
  - C = $[C_1, C_2..., C_k]$ C_r : (Connected Components) Stores all the vertices of it
  - V = $[v_1, v_2, ..., v_n]$, v_i : (vertices) Stores the component in which it is present
  - S = $[b_!, b_2....b_m]$: bridge edges present in original graph
  - G = (C,S)
  - T = DFS(G) : (Time : O(n+n-1))
  - 1. For v in V: (Total Time for loop : O(n)*O(n))
    - $Comp <= v.component$
    - B := $SearchDFS(T, Comp)$ while storing edges to reach the C_i component : (Time : (O(n-1+n)) = O(n))
    - *Bridge Set is B* = $[(b_1), (b_2), ..., (b_n)]$, where b_i edge is edge to reach from comp to C_i.
    - 2. For c in C-{$comp$}: (Total Time : c*O(k) = O(ck) = O(n))
      - 3. For vertex in c: (Time : O(k), k vertices in c)
        M[v][vertex] = B[c], B[c] = will return the corresponding bridge edge
  - Return M
- Time Complexity

- Time = O(n^2) + O($n+m$) = O(n^2)
- Finding the DFS of new graph will take O(n+n-1), as the new edge set will be of cardinality n-1 by Claim 2.
- The first loop takes the time = number of vertices x(Time of DFS of T + Time of inner loop)
- The time of DFS will be O(n).
- The time of inner loop (loop 2) = number of components x Time of second inner loop (loop 3)
- Time of loop 3 : number of vertices in that component x O(1) as Bridge Set is available = O(k), k are the vertices in that component
- So Time of Loop 2 : number of components x O(k), and as number of components x k = n, O(n), n : number of vertices in original graph.
- Time of loop 1 : n x (O(n)(Time of DFS(new graph)) + O(n)) = O(nxn) = O(n^2)
- Time of Algorithm = O(n^2) + O(n+m) = O(n^2)

- Proof of Correctness
  - Claim 1 : The new graph will be tree this can be proved by contradiction.
    Suppose there is a cycle in this new graph, then if one of the bridge edge (that is an edge for this new graph) is removed then there will exist a path connecting some connected components yet. This is a contradiction. Hence proved.
  - Claim 2 : The bridge edges are atmost n-1. This can also be proved by contradiction. Suppose there be more than n-1 bridge edges. So if we form a minimum spanning tree of this graph we will see we need only n-1 edges to connect n vertices. Hence all other edges must not be bridge edge. So bridge edges can be atmost n-1.
  - From the solution of 3.2, we are able to prove that the our algorithm for 3.2 was correct in returning the set of connected components.
  - By the two claims we are able to structurize the new Tree.
  - Further, we have to prove that the three nested loop provide correct witness matrix. that is i) all the M[i][j]'s entries are filled and ii) they are correct.
    * We know that, to reach at any vertex we are storing the bridge edge needed to reach there, by DFS on the tree of connected component, hence we are correctly storing the M[i][j] entry. So, the entry at M[i][j] will be correct.
    * Now, to prove that it fills the entry for every i,j. As the first loop works for all the vertices in V, it will definitely cover all the i indices in matrix.
      For every i, we need to to prove that it will fill entry for every j.
      As we are storing the bridge edge to reach at every vertex for every component, we are covering all the nodes to be reached. In this way all the j indices are covered.
  - Hence, all the M[i][j]'s are filled and all are correct. Hence, proved.

3.4 Q: Describe an O(m + n) time algorithm to compute a set E_0 of vertex-pairs of size at most n−1 such that (i) E ∩ E_0 is empty, and (ii) graph G_0 := (V, E ∪E_0) contains no bridge edges.

- **Claim 3.4.1**: *A tree edge (x, y), with x being parent of y in DFS tree, is a bridge edge iff High-point(y)≥Level(y)*
  - **Proof**: Proven in Class. (Lecture 7, Slide 9)
  - **Claim 3.4.2**:*Consider Tree edges $e_1 = (x, y)$ and $e_2 = (x, z)$, with x being parent of y and z in DFS tree, if we connect y and z by an edge, e_1 and e_2 can not be bridge edge for sure.*
    Proof: Suppose we remove edge e_1 from the graph, x and y can still be connected by the path via edges $x - z$ and $z - y$. Without loss of generality same applies for edge e_2. Hence neither of the edges e_1 or e_2 is a bridge edge.
  - Claim 3.4.3: *Any edge of a graph G posses ancestor descendant relationship in its DFS tree*.
    Proven in Lecture slide 7.

Solution:

- **Idea of the Algorithm:**
  - We need to add an edge set E_0 such that for any $e \in E_0$, $e \notin E$ and the resulting graph $G' = G \cup E_0$ contains no bridge edges.
  - If graph G' has no bridge edge, then removing any of the edge would not result in any 2 vertices having no path between them, which means it would not disconnect the graph.
  - $\implies$ G' is a 2-edge connected graph, i.e. removing any 1 edge out of the graph would not disconnect the graph (i.e. would not disconnect any of its vertices).
  - If T' of the graph G' should satisfy both the below conditions, we can surely claim G' doesn't have any bridge edge:
    - T' should not contain any cut edges, i.e. must satisfy the property that all edges of T' connecting x, y (x being y's parent) are such that High-point(y)Level(y). *By claim 3.4.1*
      Sub-claim 3.4.1.1: *The above condition can be satisfied by the tree if there exists a back edge from every node to its grand-parent (if it exists).*
      - To prove the above claim, we can use the fact that for an edge from $x - y$ (x-parent, y-child) of T' to not be a bridge edge, it needs to have some node in the subtree rooted in y to have a back-edge to ancestors of x, other than the edge $x - y$ itself (as this will make High-point(y)Level(y)).
      - As our subclaim makes this condition satisfied for every edge of the tree, hence no edge of the tree obtained would be a bridge edge.
    - *Sub-claim 3.4.3* Suppose the root's immediate children in T' form a set $S = \{c_1, c_2, ...c_k\}$. We form a chain of edges from $e_1 = c_1 - c_2$, $e_2 = c_2 - c_3$...$e_{k-1} = c_{k-1} - ck$, so that any of the edges connecting the root to its child is not a bridge edge. This can be proven from *sub-claim 3.4.2*, as it removing any of the root's edge to its child would render another path from the root to it via the siblings edge.
- **Algorithm:**
  - T=DFS(G) computed. During its computation check for following:
  - For nodes of level=1, store them in a set S, and form chain of vertex pairs just as in *sub-claim 3.4.3*, and add it to the set.
  - Once the level of node ≥ 2, for every node, check if $\exists$ a back edge to its parent's immediate ancestor.(maintain a visited array to ease this step)
    - If so, then proceed to the next node.
    - Else, add this back edge's end points as vertex pair to the edge set E_0.
  - Keep doing upto the leaves.
- **Time-complexity Analysis:**
  - Time taken to compute DFS tree = $O(n + m)$.
  - Nodes of level 1 can be added while making DFS tree.

- Back-edge can be checked and added while creation of DFS tree in $O(1)$ for each node (by maintaining a visited array).
- So, final time complexity is of the order of $O(m+n)$.

- **Proof of Correctness:**
  - *Claim: The above algorithm computes the set of edges E_0 such that*
    1. $E \cap E_0 = 0$
    2. $G_0 = \{V, E \cup E_0\}$ *has no bridge edges*
    3. $|E_0| \leq n-1$
  - Assume the algorithm computes a set E_0 of edges, we need to prove all the conditions satisfy for E_0.
    1. $E \cap E_0 = 0$
       - Assume $E \cap E_0 \neq 0$. $\implies$ $\exists$ atleast an edge e both $\in E$ and $\in E_0$.
       - But the algorithm only adds a back edge to the set E_0, if it is not originally present in $T = DFS(G)$, or sibling to sibling edge of root's immediate children.
       - If a back edge e is not present in T, then it can not be present in the graph G, i.e. $e \notin E$. (As DFS Tree traverses all the edges of a graph, and an edge $\{x, y\}$ will posses ancestor-descendant relationship as proven in Lecture slide - 7.)
       - It contradicts the fact that $e \in E$. So our assumption is wrong, hence proved $E \cap E_0 = 0$.
       - Sibling to sibling edge of root's immediate children cannot be an edge in the actual graph, as any edge in a graph posses ancestor descendant relationship from Claim 3.4.3.
    2. $G_0 = \{V, E \cup E_0\}$ has no bridge edges
       - Let's assume there is a bridge edge e with end points x, y, x being y's parent in G_0
       - By *Claim 3.4.1*, High-point(y)$\geq$Level(y). $\implies$ $\exists$ no edge from subtree $T'(y)$ to ancestors of x, other than e. (proven in Lecture slide 7).
       - But, our algorithm ensures every node has a back edge to its parent's immediate ancestor (if the ancestor exists, and the back edge doesn't exist).
       - If parent's immediate ancestor doesn't exist, it means the edge is of the root to its child, and these edges are also no more bridge edge as there are chain of vertex pairs of the root's children, which is proven in *Sub-claim 3.4.3*
       - $\implies$ This contradicts the assumption.
       - Hence proved, $G_0 = \{V, E \cup E_0\}$ has no bridge edges.
    3. $|E_0| \leq n-1$
       - Suppose Root has k children. The algorithm adds only edges which are
         - Chain of vertex pairs between root's children. These can be only $k-1$ such pairs, as we are just connecting a node to its next sibling and so on upto the last but one. This step adds $\leq k-1$
         - Back edge of node whose parent's immediate ancestor exists. There can be maximum n-k-1 such nodes. So edges added in this step $\leq n-k-1$
       - $|E_0| \leq n-k-1+k-1 = n-2 \implies |E_0| \leq n-2 < n-1$. Hence proved.
    4. Hence proved the correctness of the algorithm, that produces the edge set E_0, which satisfies all of the given conditions.