AMIT BORASE

CSE202 Homework 1 PID - A53095391

Question 1

Algorithm

Let G = (V, E) be the undirected graph. Then run a DFS on G and obtain a DFS tree G' = (V, E'). A vertex can be cut-vertex only if it's removal leaves graph disconnected.

i.e. any vertex in the DFS tree will be cut-vertex only if it's removal from DFS splits the graph into 2 or more components. In other words, u in V is cut-vertex if any of it's sub-tree in G' does not have a vertex that is connected to G' via non-tree edges.

Using this idea, check if u in V is cut-vertex or not based on following conditions:

1. If u is root-node of DFS tree, then it is cut-vertex only if it has more than or equal 2 children.
2. If u is leaf vertex then u is not a cut-vertex.
3. If u is internal vertex of DFS tree, then u is cut-vertex only if it has at least one child s such that none of the descendant of u via s have any back-edges to any ancestors of u.

Proof of correctness

1. Let's assume u is root-node of DFS tree, and has 2 children x y. Assume u is not a cut-vertex. Then both the children of u, must be connected through an edge between one of their descendants or them. That edge can not be tree edge as it would insert a cycle in DFS tree, and DFS only gives spanning tree. The edge can only be non-tree edge. Suppose the edge is between x' y' where x' y' are one of the nodes from each of the sub-trees of u. Without loss of generality, let us assume DFS started exploring sub-tree with x' first. Then DFS should have explored y' from x', which is not the case as y' has been explored from other sub-tree of u. Hence, assumption is wrong. I.e. the root u is a cut-vertex.
2. If u leaf node then u is not cut-vertex, because since it is a leaf node, upon its removal we'll left with only one component of graph, and not two disconnected ones.
3. Let us first prove that DFS of an undirected graph has only tree edges or back edges. (We'll use this prove claim c). Consider an edge (u, v) in G. Without loss of generality, we can assume that discovery time of u (d(u)) is less than that of v (d(v)). That means v must be finished exploring before u, since v is on u's adj-list. If the edge (u,v) is first explored from u, then v must be undiscovered until that time otherwise we'd have already explored edge from v to u. Thus, (u, v) becomes a tree edge. If (u, v) is explored first from v to u, then (u, v) becomes a back-edge, since u is still being discovered while v is being discovered.

   Coming to the main problem, Let us assume an internal node v of DFS-tree of G is not a cut-vertex. Since it is not a cut-vertex, the sub-tree must be connected via an edge from one of v's descendants to v's ancestors. Based on the DFS property proved shortly, there are only 2 possibilities for such edge, it is either a tree edge or a back edge. Now it cannot be a tree edge otherwise the DFS tree

will contain a cycle which is not possible (by the same logic as in a) above). Hence the edge has to be a back-edge only. let us first prove that DFS of an undirected graph can have only tree-edges or back-edges.

Pseudo-code

To keep track of any back-edges, We'll maintain an array that stores the minimum of the discovery times of ancestors for which the back-edge is found, from either the vertex or any of its descendants.

Listing 1: Question 1

```
1  INPUT -  Undirected graph G(V,E)
2  OUTPUT - prints the details of the vertices which are cut-vertex
3  DFS(G(V, E))
4      for each u in V     // Loop to setup intial values
5          dis[u] = -1         // tracks discovery times
6          par[u] = NIL        // parent of u
7          vis[u] = false      // has u been visited
8          back[u] = 0         // discovery_time of the any ancestors
9          child[u] = 0        // # of child of u
10     dfs_cnt = 0
11     for each u in V
12         if vis[u] == -1
13             DFS_DISCOVER(u)
14
15 DFS_DISCOVER(u)
16     vis[u] = 1
17     dis[u] = ++dfs_cnt;
18     for each v in Adj[u]    // for each edge (u,v)
19         if vis[v] != 1
20             par[v] = u
21             child[u]++
22             DFS_DISCOVER(v)
23             back[u] = min(back(u), back(v))     // adjust back_edge
24             if par[u] == NIL && child[u] > 1    // if root
25                 printf "u is an articulation point"
26             else if par[u] != NIL && back[v] >= dis[u]  // if non-root
27                 printf "u is an articulation point"
28         else if vis[v] == 1
29             back[u] = min(back[u], dis[v])      // adjust back edge
```

Time complexity analysis

- loop through 4 to 9 and 11 to 13 take O(V) time.
- And the loop through 18 to 29 is executed Adj[u] times. Summing it for all the u's in V we get

$$\sum_{v}^{V} Adj[v] = O(E)$$

That Implies total time for executing lines through 10 to 13 is O (E)

Hence total running time of above algorithm is **O(V+E)**, where V is number of vertices in G and E is number of edges in G.

Question 2

Assume G(V,E) is the undirected graph.

Proof by contradiction

Let us assume that there does not exist a pair of vertices u v, that are bicritical w.r.t. s t. => There exists at-least three distinct paths from vertices s t, so that even if we remove single vertex from 2 of the paths, there will still exist a path from s to t.

Then minimum number of nodes in such graph will be nodes along these 3 paths including s and t.
=> 3 * [(shortest path between s t) -1] + 2(for s t) <= total number of nodes
=> 3 * [ceiling-of(n/3) -1] + 2 <= n
=> 3 * ceiling-of(n/3) - 3 + 2 <= n
=> 3 * ceiling-of(n/3) - 1 <= n
Now 3 * ceiling-of(n/3) can be either n or n+1 or n+2. for n+2 case, the equation becomes
(n+2) - 1 <= n
=> which is contradicting.

hence, if the shortest path between the two vertices s and t is strictly greater than ceiling-of(n/3) then there exists a pair of vertices that are bi-critical w.r.t. s and t.

Algorithm

Below algorithm can be used to find all the pair of bi-critical vertices w.r.t. s and t.

- Run a Breadth first search starting at s to find various layers of the BFS spanning tree.
- If the subtrees of s do not have any cross-edges between them then consider only the subtree of s (of the BFS spanning tree) containing vertex t. Otherwise consider subtree of s contaning t along with all other subtrees that share cross-edges with subtree contaig t. Now since there is minimal shortest path between s and t, we can be sure that there will be a subtree of s containing t.
- Remove all the vertices from this subtree that do not lie on any of the path from s to t.
- Now consider the BFS levels of such reduced subtree of s containing t.
  - If there is any level with only 2 nodes in it AND the none of the descedants of these nodes have any cross edges to ancestors then, the given pair of vertices are bi-critical w.r.t s and t
  - If there is any level with only one node u in it, then all the pair of vertices (u, v) where v != u, s, t, is a bi-crtitcal pair w.r.t. s and t.

Pseudo-code

Listing 2: Question 3

```
1 INPUT: undirected graph G(V,E) and s and t
2 OUTPUT: returns all the pairs of bi-critical points of s and t..
```

```
Perform a BFS on G starting at s.
S = subtree of s containing t + any subtrees sharing cross-edges with subtree containg t.
PATH_VERTICES=[]
Perform DFS on S starting at s
   Whenever t is visted by DFS, add all the vertex to PATH_VERTICES
   In such way, get all the vertices visted while traversing from s to t.
Remove all vertices from subtree S except for the ones in PATH_VERTICES and s and t
for all i in levels of S
   if level i has only 2 nodes u and v
       output u and v as a bi-critical pair of s and t.
   elif level i has only 1 node u
       out u and v, where v != u, s, t as a bi-critical pair of s and t
```

Time complexity

BFS of a G will take O(V+E) time. In next step, we'll traverse the subtree contating t, and remove any vertices that do not lie on the containing t, that will take O(V+E) time. Hence the total run-time for such algorithm will again be O(V+E).

Question 3

A directed graph can be one-way-connected if all the strongly connected component of it have a path among them either directly or indirectly (through other strongly connected components).

Algorithm

- Use Kosaraju's algorithm as below to find all the strongly connected components of the graph G.
    1. Perform a DFS on G = (V,E), such that when a node is done exploring, push it onto a stack.
    2. Now build a transpose of G called G', such that there is edge (v->u) in G' for every edge (u->v) in G.
    3. Pop a node u from the stack and perform a DFS on it, all the vertices discovered by this DFS will constitute a strongly connected component of u.
    4. repeat above step till stack is not empty.
- Now build a graph G' such that v in V' corresponds to a strongly connected component of G. And every edge (u,v) in E' means corresponding component of u has an edge to component of v in G.
- Then sort G' in a topological order
- Check if for such topological ordering, there is an edge between each adjacently placed nodes or not, if yes then the given graph G is one-way connected.

Proof of correctness

Let G' = (V', E') be the directed graph formed from G after replacing each strongly connected components of G with a vertex in G' such that if the two strongly connected components are connected via a directed

edge, then there is a edge between their corresponding vertices in G'. Now the graph G' has to be directed acyclic graph, otherwise if there is cycle in it, then it can't have individual strongly connected components corresponding to nodes in the cycle, otherwise they all will form a one-big strongly connected component.

Let us assume that the algorithm yielded a topological sorting of G' be v1, v2, ..., vk, where k is the number of strongly connected components of G, such that all the adjacent vertices in topological sorting have an edge between them. i.e. (vi, vi+1) is in E' for all i = 1,2,....k-1

=> (vi, vi+1) (vi+1, vi+2) is in E' for all i = 1,2, ...k-2

=> vi+2 can be reached from vi through vi+1 and vi can be reached through vi-2

=> extending the above statement to both the extremes of ordering, we get

=> every vi in V' has a path to all vertices after it in the topological ordering and has a path to it from all the vertices before it in topological sorting.

=> every (vi, vj) in V' either has a path vi->vj or vj->vi.

Now assume the graph G is not one-way-connected, that means there exists a pair of vertices x and y such that they don't have any path x->y or y->x

=> x y are in different strongly connected components of G, Let say X and Y. But since strongly connected component are internally connected, above is possible only if X Y do not have a path from X to Y or vice-versa. Which is not possible as corresponding nodes of X Y (vx vy of V' in G') are shown to have a path either vx–>vy or vy->vx.

Hence the assumption is wrong.

Hence, if topological ordering of G' is found such that there exists a (vi, vi+1) in E' for all i = 1,2,....k-1 , then the goven graph G is a one-way-connected graph.

Pseudo-code

Listing 3: Question 3

```
INPUT: Directed graph G(V,E)
OUTPUT: returns true if G is one-way-connected graph.
DFS_STACK(G(V, E), Stack st)
    for each u in V         // Loop to setup intial values for various variable
        dis[u] = -1     // tracks disovery times
        vis[u] = false      // has u been visited
        dfs_cnt = 0
        for each u in V
        if vis[u] == -1
            DFS_DISCOVER_STACK(u)
            st.push(u)

DFS_DISCOVER_STACK(u)
    vis[u] = 1
    dis[u] = ++dfs_cnt;
    for each v in Adj[u]        // for each edge (u,v)
        if vis[v] != 1
            DFS_DISCOVER_STACK(v)
    st.push(u)

GRAPH_REVERSE(G(V, E), G'(V, E'))           // Reserse a graph
    for each u in V
        add_node u to G'
```

```
        for each v in Adj[u]
            add_edge(v,u) to G'

FIND_COMPONENTS(G'(V,E'), stack st)     // Find all strongly connected components
    for u = stack.pop(st) till !stack.empty(st)
        if (u is not already a part of component)
            Perform a DFS on G' with u as a root
            print the nodes discovered by this DFS (they form strongly connected ↩
               component of u)

GRAPH_OF_COMPONENTS(G"(V",E"))
    for scc1 in a group of SCC's
        if there is edge (u,v) in G, for some u in scc1 such that v is not in scc1↩
            but in scc2
            add_edge(s1->s2, G")

GRAPH_IS_1_WAY_CONN(G)
    stack st
    DFS_STACK(G)
    GRAPH_REVERSE(G, G')
    FIND_COMPONENTS(G', st)
    GRAPH_OF_COMPONENTS(G")
    Perform a topological sort on G"(V", E") and store it on Queue Q
    for ui in Q
        if edge(ui->ui+1) is not in E"
            return false
        else
            continue
    return true
```

Time complexity analysis

- The above algorithm does 1 DFS run to build stack and another DFS run to find strongly connected component of the graph G. And the time complexity for these DFS runs remains O(V+E)
- Algorithms also does a reverse of a graph, which traverses all the Adjacency lists, hence its time complexity is also O(V+E)
- Building a graph of strongly connected components again means traversing the adj-lists, which is again O(V+E)
- Topological sorting and traversing the node in topological order takes O(V+E).

Hence the total run time complexity remains **O(V+E)**, where V is number of vertices in G and E is number of edges in G.

Question 4

Since given graph is directed acyclic graph, it can be sorted topologically. We'll walk such topological order starting from given vertex v and mark the rest of the vertices (that are ahead of v in topological order) as

either at even or odd length. If we revisit a vertex again with even length and its previous marking was odd then we modify it to be at even-odd length from u, that is it is accessible from u using both even length and odd length paths. One more thing to note is that if the node is at even-odd length from u, then all the nodes that are accessible from it are also at even-odd length from u.

Algorithm

This algorithm marks all the nodes that are placed after u in topological ordering, as either -1, 0, 1 or 2, where -1 means it is not accessible from u, 0 means it is at even length, 1 means it is at odd length and 2 means it is at both even and odd lengths from u. Important thing to note is that the vertex v marked at even-odd length from u, means all the vertices accessible from v are also at even-odd length from u.

- Topologically sort the vertices in V in the order u1, u2,.....,uv.
- if at all any vertices are placed before u in the topological order, then they can be ignored.
- Start from vertex u and mark it as even length. Then visit all the nodes adjacent to v and mark them even.
- move to the next node in topological sort, if it has not been marked so far then ignore it. Otherwise proceed marking its adjacent nodes. While adjusting the marks, if a child has odd mark (or even) and the current nodes mark is also odd (or even) then child needs to be marked even-odd. Child has to be unconditionally marked even-odd if the current node marking is even-odd.
- repeat the above step till we reach the end of the topological order.
- Print all the vertices that has even or even-odd marking on them.

Proof of correctness

Let G = (V, E) be the directed acyclic graph. So it can be safely assumed that we'll find a topological ordering of the given G. Let v1, v2,..., vi=u, ...,vk (for some i in 0 to k) be the topological ordering of G.

=> There is no edge (vj, vi) such that i<j

=> If we traverse the vertices in a topological order and visit the neighbourhood of a node using its Adj-list, then we're sure to not revisit any vertices that are placed prior to current vertex in topological order.

=> Path length from u of a node v (calculated using the above algorithm) while traversing topological order, will remain constant (and not change due to additional path discovery from u to v) once we've traversed past v in topological order.

=> After topological order has been traversed, nodes that are marked with even or even-odd path lengths will be the nodes that have even length path from u.

Pseudo-code

Listing 4: Question 3

```
1 INPUT: A directed acyclic graph G = (V,E)
2 OUTPUT: List of vertices which are at even length from the vertex u in V.
3 EVEN_LENGTH_PATH(G(V,E))
4     Perform toplogical sort on G and get ordering V'
5     for i iterate over V'
6         mark[i] = inaccessible
```

```
7      u_found = false;
8      for i iterate over V'
9          if u_found==true and i != u
10             u_found=true
11             for j in adj_list[i]
12                 if mark[i]==odd:
13                     if mark[j]==inaccessible
14                         mark[j]=even
15                     elif mark[j]==odd
16                         mark[j]=even-odd
17                 if mark[i]==even:
18                     if mark[j]==inaccessible
19                         mark[j]=odd
20                     elif mark[j]==even
21                         mark[j]=even-odd
22                 if mark[i]==even-odd
23                     mark[j]=even-odd
24         elif u_found==false and i == u
25             u_found=true
26             mark[i]=even
27             for j in adj_list[i]
28                 if mark[j]==inaccessible
29                     mark[j]=odd
30                 elif mark[j]==even
31                     mark[j]=even-odd
32         elif u_found==false and i != u
33             continue
34
35     for i iterate over v'
36         if mark[i] == even || mark[i] ==even-odd
37             print "i"
```

Time complexity analysis

- First part of the algorithm builds a topological sorting of a given DAG (using modified DFS) hence it takes O(V+E) time.
- for loops through line 5-6 and line 12-33 and line 35-37, al of them take O(V) time.
- Second part traverses all the vertices along the topological sorting (O(V)) and checks the adjacent nodes for each of them (O(E)). Hence this again takes O(V+E) time.

Hence complete run-time taken is O(V+E), where V is number of nodes in G and E is number of edges in G.