ADA Tutorial 1 - Tuesday

August 5, 2025

Question 1

A cut-vertex is a vertex whose removal disconnects the graph. A graph $G = (V, E)$ is 2-vertex connected (or biconnected) if it has no cut-vertex. Give a procedure that checks in linear time if a graph is biconnected.

Proof. Let $T_{DFS}(x)$ be the DFS tree of $x \in V$. Since $T_{DFS}(x)$ is an undirected graph, it will have only tree edges or back edges. For $v \in V$, we define $low(v)$ to be earliest discovery time (of a vertex) reachable from v via a back edge.
If x has more than one child we report x as a cut vertex. During DFS backtracking, we check for each internal node v if $min\{low(w) : \text{w is a child of v in} T_{DFS}(x)\} < disc(v)$. If this is not the case we report v as a cut vertex.

We make the following claims:

1 a leaf $T_{DFS}(x)$ of is never a cut vertex.

2 x is a cut vertex if and only if it has at least two children.

3 a non-leaf, non-root vertex u is a cut vertex if and only if $\exists$ child w of u here is no descendant of w with a back edge reaching above u

The first two claims are trivial. The proof of the third claim is as follows.
Suppose u is a cut vertex. Removing u from the graph separates the graph into at least two components. The root x of the DFS tree is in one of these components, say, S.There is at least one other component, say, S'. Let w be the first vertex in S' encountered by the DFS. Since all paths from x to w go through u, w has to be a descendant of u in the DFS tree. Since w is the first vertex in S' encountered by the DFS, w has to be the immediate child of $u$in the DFS tree. Since all paths from x to w go through u, there cannot be a descendant of w that has a back edge going above u.
For the reverse implication, If u has a child w, with no descendant of w having a back edge going to a vertex above u, then removing u disconnects the child from the parent of u, therefore u is a cut vertex.
Since our algorithm is just a a run of DFS we conclude that it is linear time.

□

Question 2

Consider the following procedure to check if a directed graph $G = (V, E)$ is strongly connected and prove its correctness.

i **Pick a vertex $v \in V$ and run a DFS from v. If some vertex is not reached in the DFS then G is not strongly connected.**

ii **Reverse (replace edge (u, v) with the edge (v, u)) every edge of $G$and let the resulting graph be G^R.**

iii **Run a DFS on G^R from v. If some vertex is not reached in the DFS then G is not strongly connected.**

iv **Declare G is strongly connected.**

Proof. To prove that this process (which we will refer to as P) is correct we need to show:

1 A graph that is accepted by the procedure above is strongly connected.

2 A strongly connected graph is accepted by the procedure above

Proof of [1]: Suppose P accepts graph G then there is a vertex v that can reach all vertices as $[i]$ holds true. Moreover, v can also reach every vertex in G^R which implies that every vertex can reach v. This means for any pair of vertices (u, w),

$$u \longrightarrow v \longrightarrow w \text{ and } w \longrightarrow v \longrightarrow u$$

Every pair of vertices has paths to and from one another, which means G is strongly connected.
Proof of [2]: Let G be strongly connected. Check [i] is true since DFS from every vertex reaches every other vertex. Check[iii] also holds for all vertices of G (if G is strongly connected so is G^R). Therefore P accepts G. □

Question 3

A strongly connected component (SCC) is a maximal set of vertices that are strongly connected. Prove that any two strongly connected components of a graph are disjoint.

Proof. The proof of this was discusssed in class using contradiction. Can you also try to prove this by showing that the relation $R \subseteq V \times V$ st $(u, v) \in R$ iff there is a path from u to v and v to u is an equivalence relation. □

Question 4

Modify the procedure in Q2 to identify all strongly connected components in a graph.

Proof. The graph G' obtained by "shrinking" strongly connected components of G is a *directed acyclic graph*. Notice that if you start DFS from a node of G that is in a sink strongly connected component of G' then only that strongly connected component would be returned. We can recurse by deleting that componenet and starting DFS from another sink strongly connected component. So how do we recognize such such a sink strongly connected component? Turns out, that the sink strongly connected components are sources in G^R, and there is an easy way to find sources using finish time of the vertices during DFS. The complete method (Kosaraju's Algorithm) is described in Section 3.4 of Algorithms by Dasgupta, Papadimitrou and Vazirani. □

Question 5

A directed graph $G = (V, E)$ is weakly connected if for every pair of vertices u, v, there is a path from u to v or from v to u. Give a linear time algorithm to check if G is weakly connected.

Proof. The Algorithm to check for a weakly connected graph is as follows:

1 Compute the Strongly connected components (SCC's) of G as described above in linear time.

2 Since the graph of the SCC's is a DAG, compute topological sort in linear time.

3 Check if the graph of the SCC's Have a Hamiltonian path (A path that visits each node exactly once). If yes, G is weakly connected.

We need to prove that the Algorithm given above is correct and that step [3] takes linear time to compute.
Correctness: Suppose the topological ordering of the SCC's are $S_1, S_2, \ldots S_k$. A DAG has a Hamiltonian path if and only if there is a topological order $S_1, S_2, \ldots S_k$ such that there is an edge from S_i to S_{i+1} for all $i = 1$ to $k - 1$ (why is this true?).
So, for any two vertices $u \in S_i$, $v \in S_j$: if $i < j$, path from $u \longrightarrow v$ exists. If the two vertices are in the same SCC, then connectivity goes both ways. *Time Complexity*: We need to check whether two consecutive SCCs in the topological ordering is connected. This takes $O(k)$ time, which makes the algorithm linear.

□

Question 5

The distance between a pair of vertices u, v is the length of the shortest path between u and v. The diameter of a graph is the maximum distance between any pair of vertices. Give a linear-time algorithm to compute the diameter of a tree.

Proof. The Algorithm works as follows:

- From arbitrary $x \in V$, do DFS/BFS to find the farthest node v from x.
- Do BFS/DFS from v and let u be the farthest from V
- Return $d(u, v)$ as the diameter of the tree.

The algorithm works on the basis of the following lemma:
Lemma: The farthest vertex from any vertex of a tree is one of the endpoints of the diameter of the tree. □