Homework 9 Solution

1. *The input is a connected undirected graph G, a spanning tree T of G, and a vertex $v \in V(G)$. Design an $O(|E|)$ time algorithm to determine whether T is a valid DFS tree of G rooted at v, i.e., determine whether T can be an output of DFS under some order of edges starting with v.*

   Apply DFS in the spanning tree, meanwhile traverse the graph using the order of doing DFS in the tree. If any back edge is founded between a visited vertex in the tree and an unvisited vertex in the tree. Then the tree is not a DFS tree.

2. *Characterize conditions of undirected graphs that contain a vertex v such that there exists a DFS tree rooted at v that is identical to a BFS tree rooted at v. Note that two spanning trees are identical if they contain the same set of edges.*

   The DFS and BFS are identical if and only if the graph do not have circle.

3. *Given an undirected graph G, the length (no weight) of the smallest cycle is called the girth of the graph. Design an $O(|E|)$ time algorithm to compute the girth of a given graph G.*
   LCA O(n) preprocess, O(1) query.
   Use BFS to traverse the graph, then use LCA to compute the cycle (BFS recording all the ancestors in order and depth, so the number of edges in the cycle can be obtained in O(1) for the next step).
   Then compute the number of edges in the cycle in O(1). Compare it to the current girth. If it is smaller then set it as the current girth. If it is larger than or equal to the current girth. Don't do anything. The current girth is initialized to be infinity.

4. *Given a connected undirected graph G and three edges $(u_1, v_1), (u_2, v_2), (u_3, v_3)$ of $E(G)$, design an $O(|E|)$ time algorithm to determine whether there exists a cycle in G that contains both (u_1, v_1) and (u_2, v_2), but does not contain (u_3, v_3).*

   Step1: remove (u_3, v_3) from G and we get G'
   Step2: remove (u_1, v_1) from G' and we get G''
   Step3: start from v_1 use BFS to find all path to u_1
   Step4: check if one of the path contain (u_2, v_2) if yes then exists such circle, otherwise, no such circle exists.
   Time complexity analysis:
   Step1 and step2 takes $O(1)$, step3 takes $O(|E|)$, step4 at most takes $O(|E|)$. Thus the time complexity is $O(|E|)$.

5. *Depth-first numbers are given to the vertices in the graph G shown in Figure 1.*

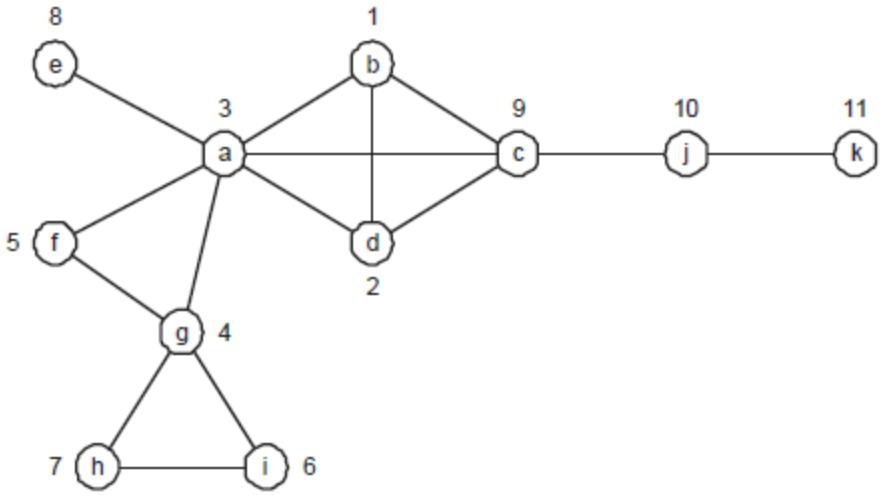
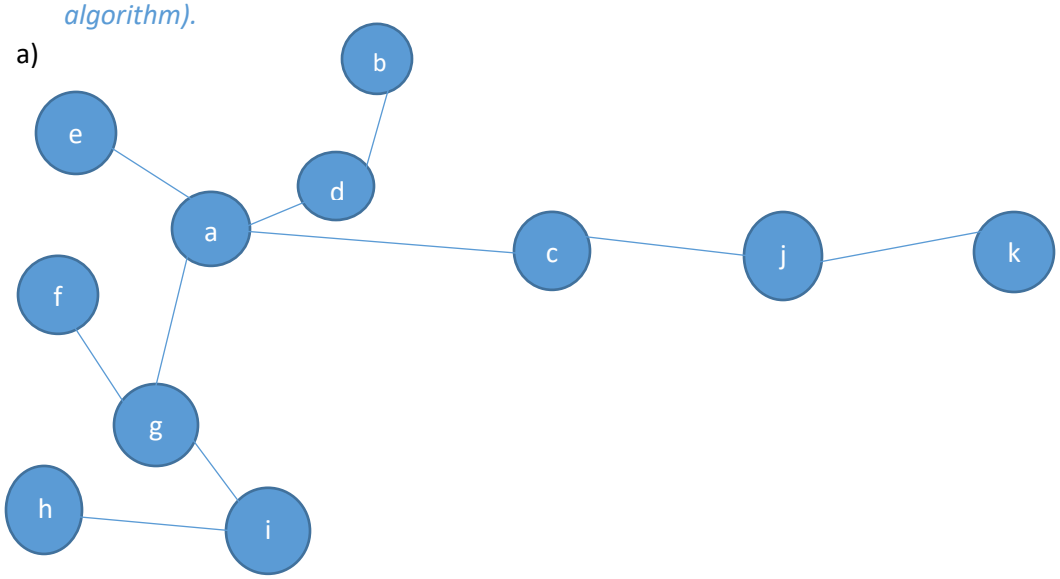


Figure 1:

(a) Show the depth-first-search tree of G corresponding to the depth-first number.
(b) Compute the low point of each node in G using the definition of the low point (not by an algorithm).

a)



b) Low point: g, c, j

6. *Consider the following problem: Is there a vertex v in G such that every other vertex in G can be reached by a path from v? If G is an undirected graph, the question can be easily answered by a simple depth-first (or breadth-first) search and a check to see if every vertex were visited. Describe an algorithm to solve for a directed graph. What is the complexity of your algorithm?*

Use Floyd Warshall algorithm for all way shortest path. Setting the edge weight as 1. Check row v and see if there is infinity in the row. If no, then all other vertices are reachable from v. The time complexity will be $O(|V^3|)$.

7. *Consider a graph G.*
   (a) Explain how one can check a graph's acyclicity by using breadth-first-search.
   (b) For either of the two search algorithms, DFS and BFS, always find a cycle faster than the other? If your answer is yes, indicate which of them is better and explain why it is the case; if your answer is no, give two examples supporting your answer.

   (a) Start BFS from an arbitrary vertex. And build the BFS tree.
   If there is an edge connecting to the visited vertex, then the graph is not acyclic.
   If the BFS tree is built and no cross edge and back edge is found. Then the graph is acyclic.

   (b) No.
   Any tree takes equal time for DFS and BFS to decide acyclicity.

8. *A graph is called bipartite if all its vertices can be partitioned in two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y.*
   (a) Design a DFS-based algorithm for checking whether a graph is bipartite.
   (b) Design a BFS-based algorithm for checking whether a graph is bipartite.

   a) Assign the first vertex you visit in odd set.
   While applying DFS, assign the vertex you visit to odd set when its parent is in even set and assign the vertex you visit to even set when its parent is in odd set.
   If you are trying to assign a vertex in odd set into even set then break, and the graph is not bipartite and vice versa.
   After all vertex is visited, then the graph is bipartite.

   b) Build a BFS tree using BFS.
   Check other edges other than the tree edges. If there exists an edge connecting between even depth vertices (both are even depth vertices) or odd depth vertices (both are odd depth vertices), then break, and the graph is not bipartite.
   After all edge is checked. The graph is bipartite.