ESO207A: Data Structures and Algorithms

Practice Set *4a: Basic Graph Algorithms* –

Problem 1. You are given a tree $T = (V, E)$ in adjacency list format along with a designated root vertex $r \in V$. Recall that u is said to be an *ancestor* of v in T if u lies on the unique simple path from the root to v.

We would like to answer queries of the form "is u an ancestor of v" in *constant* time. To do this, you are allowed to pre-process the tree in linear time. Can you think of an algorithm for pre-processing and for answering queries.

Solution. . We are given a rooted tree with root r. Say we run DFS from the root r and store the discovery and finish times for all vertices. Such a DFS will visit the children of the node recursively (and in the order given in the adjacency list) and then return to the node after coloring both children black. So if u is an ancestor of v, then, u will be on stack while v is being explored. Hence, the interval $[v.d, v.f]$ is completely contained in $[u.d, u.f]$. On the other hand, if u and v are not related to each via ancestor descendant relation, then one will finish before the other, and so the intervals will not overlap. Hence, the pre-processing algorithm is just DFS together with storing the $[u.d, u.f]$ intervals. The test is u an ancestor of v is: the interval of u encloses the interval of v.

Problem 2. You are given a tree $T = (V, E)$ along with a designated root vertex $r \in V$. This then uniquely defines a rooted, oriented tree. The parent of any node $v \neq r$ is denoted as $p(v)$ and is the node adjacent to v in the unique path from r to v. By convention, set $p(r) = r$. Define $p^1(v) = p(v)$. The grandparent of v is defined as $p(p(v))$ and denoted as $p^2(v)$. Similarly, $p^k(v)$ is defined as $p(p^{k-1}(v))$ and is the kth ancestor of v.

Associated with each vertex in the tree is a non-negative integer label $l(v)$. You have to update the labels of all the vertices in T according to the following rule: $l_{new}(v) = l(p^{l(v)}(v))$.

Solution. . At any time in DFS, the sequence of gray vertices, sorted in decreasing order of $u.d$ is the DFS stack of vertices, top to bottom. We need to peek into this stack. Extend the $\textsc{Top}(S)$ operation on a stack S as $\textsc{FromTop}(S, k)$. It returns the kth entry from the top of the stack. So $\textsc{FromTop}(S, 0)$ is $\textsc{Top}(S)$. No updates are done. As implementation is done as arrays, this takes $O(1)$ time. If $k \geq$ the number of elements in the stack -1, the bottom most value is returned.

The algorithm is the following. (1) Maintain the DFS stack of vertices explicitly. (2) As soon as DFS_Visit is started on a new vertex u, push it on this stack, and set

$$v = \textsc{FromTop}(S, u.l),\ u.l_{new} = v.l$$

Problem 3. This problem concerns strongly connected directed graphs and is the counterpart to bipartiteness of undirected graphs discussed in class.

1. Prove that a strongly connected directed graph is bipartite only if it has no odd cycles.
2. Suppose that BFS of G starting from any vertex does not yield any edge whose end points have the same parity (odd/even) of the index of level sets. (i.e., there is no edge (u, v) such that u is in L_i, v is in L_j and i and j are either both odd or both even. Show that in this case, G is bipartite.

3. Conversely, suppose that BFS on G yields an edge (u, v) with both u and v having the same level set index parity. Now construct an odd cycle in G involving at least one of u or v (or both). Hence we have shown that a directed graph is bipartite iff it has no odd cycles (argue).

4. Give a linear time algorithm to test if a directed graph has an odd cycle. Find one such cycle if there is one.

Solution. 1. Suppose G is directed and has an odd cycle $v_0 \to v_1 \to v_2 \ldots \to v_{2k} \to v_0$. Color v_0 red, then v_1 must be colored blue, then v_2 must be red, alternating and so on. Then v_{2k} must be colored red, but then it is adjacent to v_0 which is red. Hence no bi-coloring is possible and G is not bipartite.

2. This part is obvious. Run BFS, and consider level sets, $L_0, L_1, \ldots$. Color all vertices in odd levels as red and even levels as blue. Now, if there are no edges from odd levels to odd levels or even levels to even levels, we have found a valid bi-partition of the graph.

3. Suppose we run BFS and find two vertices $u \in L_i$ and $v \in L_j$ such that i and j are either both odd or both even. Say for example, they are both odd. Now do a BFS (or DFS) from v. Being a strongly connected component, there will be a path Q from v to s. If this path Q has odd length, then we get an odd cycle, namely, $s \rightsquigarrow u$ the original BFS path from s to u, followed by the edge (u, v), followed by the second BFS path v to s. The parity of the length of this path is odd + 1 + odd = odd. If the path Q has even length, then too we get an odd cycle, consider the BFS path from $s \rightsquigarrow v$ followed by the second BFS path from v to s. The parity is odd + even = odd. An analogous argument follows when the original parity is even. Thus we have found an odd length cycle in G in this case.

4. Separate the graph into its strong components. Run BFS on each strong component from any vertex s say and test as in (2) whether the component is bipartite. If a component is determined to be not bipartite, then, there is an edge (u, v) such that $u \in L_i, v \in L_j$ and i and j have the same parity. Now, run a second BFS from v and depending on whether the distance of s is odd or even, we get an odd cycle. If all components are bipartite then G is bipartite, otherwise not.

Problem 4. Given a DAG G, give a linear time algorithm that tests whether G contains a directed path that touches every vertex exactly once.

Solution. Consider the topological ordering $u_1, u_2, \ldots, u_n$. G contains a directed graph that touches every vertex exactly once iff the path is $u_1 \to u_2 \to u_3 \ldots \to u_n$. So for each vertex u_i, test if u_{i+1} is adjacent to it. This can be done in $deg(u_i)$ time. So the total cost is $\sum_u deg(u) = |E|$.

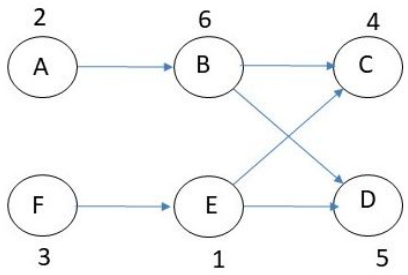
Why is the property true? Suppose there is a path touching all vertices. Order the vertices along this path. This ordering must match the topological ordering, since every path in G must be consistent with the topological ordering.

Problem 5. Given a directed graph $G = (V, E)$ where each node $u \in V$ has a *price* attribute $u.price$. Define the array *cost* as follows: for each $u \in V$

$$cost[u] = \text{ price of the cheapest node reachable from } u \text{ (including } u \text{ itself)}$$

For example, in the graph below, prices are shown for each vertex. The *cost* values of the nodes A, B, C, D, E, F are 2, 4, 4, 5, 1, 1 respectively.

Solution.



1. Obtain the strongly connected component graph G^{SCC} corresponding to the graph. In every strongly connected component C, the cost of all the vertices is the same, since their reachability set is the same.

2. For every strong component $C \in V^{\text{SCC}}$, initialize $C.price = \min_{u \in C} u.price$.

3. Topologically sort the vertices of V^{SCC} of G^{SCC}.

4. for every vertex u in reverse topological order
   for every vertex v adjacent to u
   $u.price = \min(u.price, v.price)$.

   The whole algorithm is linear time.