CS412 Algorithms: Design & Analysis

Spring 2024

Dhanani School of Science and Engineering

Habib University

Practice Problems

Week 8

1. Make a 3-by-3 chart with row and column labels $WHITE$, $GRAY$, and $BLACK$. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

Solution:

For directed graphs:

$from \backslash to$	$BLACK$	$GRAY$	$WHITE$
$BLACK$	$Allkinds$	$Back, Cross$	$Back, Cross$
$GRAY$	$Tree, Forward, Cross$	$Tree, Forward, Back$	$Back, Cross$
$WHITE$	$Cross, Tree, Forward$	$Cross, Back$	$allkinds$

For undirected graphs, note that the lower diagonal is defined by the upper diagonal:

$from \backslash to$	$BLACK$	$GRAY$	$WHITE$
$BLACK$	$Allkinds$	$Allkinds$	$Allkinds$
$GRAY$	–	$Tree, Forward, Back$	$Allkinds$
$WHITE$	–	–	$Allkinds$

2. Show that edge (u, v) is

   (a) a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,

   (b) a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and

   (c) a cross edge if and only if $v.d < v.f < u.d < u : f$.

Solution: a) Since we have that $u.d < v.d$, we know that we have first explored u before v. This rules out back edges and rules out the possibility that v is on a tree that has been explored before exploring u's tree. Also, since we return from v before returning from u, we know that it can't be on a tree that was explored after exploring u. So, This rules out it being a cross edge. Leaving us with the only possibilities of being a tree edge or forward edge. To show the other direction, suppose that (u, v) is a tree or forward edge. In that case, since v occurs further down the tree from u, we know that we have to explored u before v, this means that $u.d < v.d$. Also, since we have to of finished v before coming back up the tree, we have that $v.f < u.f$. The last inequality to show is that $v.d < v.f$ which is trivial.

b) By similar reasoning to part a, we have that we must have v being an ancestor

of u on the DFS tree. This means that the only type of edge that could go from u to v is a back edge. To show the other direction, suppose that (u, v) is a back edge. This means that we have that v is above u on the DFS tree. This is the same as the second direction of part a where the roles of u and v are reversed. This means that the inequalities follow for the same reasons.

c) Since we have that $v.f < u.d$, we know that either v is a descendant of u or it comes on some branch that is explored before u. Similarly, since $v.d < u.d$, we either have that u is a descendant of v or it comes on some branch that gets explored before u. Putting these together, we see that it isn't possible for both to be descendants of each other. So, we must have that v comes on a branch before u, So, we have that u is a cross edge. To See the other direction, suppose that (u, v) is a cross edge. This means that we have explored v at some point before exploring u, otherwise, we would have taken the edge from u to v when exploring u, which would make the edge either a forward edge or a tree edge. Since we explored v first, and the edge is not a back edge, we must of finished exploring v before starting u, so we have the desired inequalities.

3. Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

Solution: We can't just use a depth first search, since that takes time that could be worst case linear in $|E|$. However we will take great inspiration from DFS, and just terminate early if we end up seeing an edge that goes back to a visited vertex. Then, we should only have to spend a constant amount of time processing each vertex. Suppose we have an acyclic graph, then this algorithm is the usual DFS, however, since it is a forest, we have $|E| \leq |V| - 1$ with equality in the case that it is connected. So, in this case, the runtime of $O(|E| + |V|)O(|V|)$. Now, suppose that the procedure stopped early, this is because it found some edge coming from the currently considered vertex that goes to a vertex that has already been considered. Since all of the edges considered up to this point didn't do that, we know that they formed a forest. So, the number of edges considered is at most the number of vertices considered, which is $O(|V|)$. So, the total runtime is $O(|V|)$.

4. How can the number of strongly connected components of a graph change if a new edge is added?

Solution: It can either stay the same or decrease. To see that it is possible to stay the same, just suppose you add some edge to a cycle. To see that it is possible to decrease, suppose that your original graph is on three vertices, and is just a path passing through all of them, and the edge added completes this path to a cycle. To see that it cannot increase, notice that adding an edge cannot remove any path that existed before. So, if u and v are in the same connected component in the original graph, then there are a path from one to the other, in both directions. Adding an edge wont disturb these two paths, so we know that u and v will still be in the same SCC in the graph after adding the edge. Since no components can be split apart, this means that the number of them cannot increase since they form a partition of the set of vertices.

5. Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

Solution: Given the procedure given in the section, we can compute the set of vertices in each of the strongly connected components. For each vertex, we will give it an entry SCC, so that $v.SCC$ denotes the strongly connected component (vertex in the component graph) that v belongs to. Then, for each edge (u, v) in the original graph, we add an edge from $u.SCC$ to $v.SCC$ if one does not already exist. This whole process only takes a time of $O(|V| + |E|)$. This is because the procedure from this section only takes that much time. Then, from that point, we just need a constant amount of work checking the existence of an edge in the component graph, and adding one if need be.