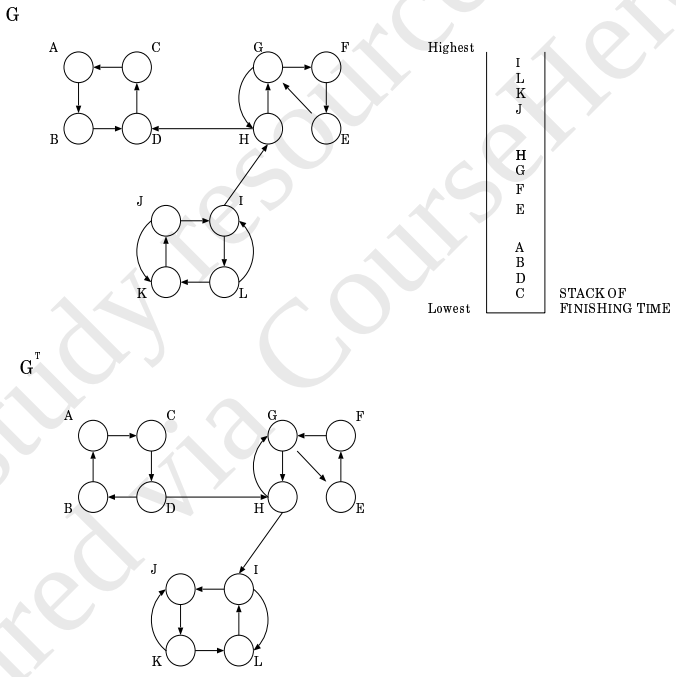
Purdue University
CS 381

S. Hambrusch - C. Guerra
Fall 1999

Solution Sketches to Assignment 7

1) (Graded by Tian Luan) Consider the strongly connected component algorithm presented in class. The DFS traversal on G^T considered the vertices by decreasing finish times. (Recall that the finish time refers to the order of completion of the recursive calls of the DFS on G.) Give an example of a directed graph consisting of 12 vertices and three strongly connected components for which the processing of vertices by decreasing finish times is crucial. In particular, show by example that when the DFS on G^T does not consider vertices by decreasing finish times, the algorithm fails to detect the three strongly connected components.

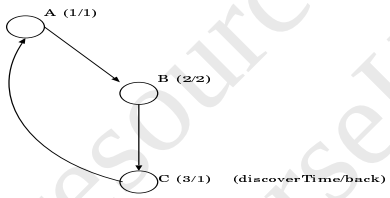


Consider the directed graph shown above. Assume that the first DFS starts a A, then at H and finally I. The finishing times are shown in the corresponding stack. In the second DFS on G^T, vertices are poped from the stack according to finishing times are we generate three strongly connected components in this order: $IJKL, HGEF, ACDB$.

If we do not pop the vertices from the stack by the order of decreasing finish time, these strongly connected components are not generated. For example, if DFS starts at vertex H, we generate a strongly connected component consists of vertices $HQJKLEFG$.

2) (Graded by Tian Luan) Consider the biconnected component algorithm presented in class. Would the algorithm work properly if the test for a biconnected component were changed to $back \geq discoverTime[v]$? If so, explain why; if not, give an example in which it does not work.

The algorithm will not work properly if the test for a biconnected component were changed to $back \geq discoverTime[v]$. With this statement, the algorithm can output a wrong biconnected component or may be unable to detect a biconnected component. Consider the following example.



Assume $B = v$, $C = w$. (B, C) is a tree edge and $wback = 1$. The statement $if\,(back \geq discoverTime[v])$ returns return true since $back[v] = 2$ and $discoverTime[v] = 2$. Thus, vertex B is incorrectly identified as an articulation point and a wrong biconnected component is detected. (The algorithm would output a component consisting of vertices B and C, which is not correct.)

3) The 3-coloring problem is defined as follows: Given are 3 colors (say, R, G, and B) and an undirected graph $G = (V, E)$. You want to determine whether it is possible to assign to every vertex one of the three colors so that every edge of G is incident to vertices having different color.
(i) Give an example of a graph in which every vertex has degree at most 3 which is not 3-colorable.

An example of such a graph is one with 4 vertices where each vertex is connected to every other vertex. If we color any 3 vertices in 3 different colors, the fourth vertex will not have a color, since it is connected to 3 differently colored vertices.

(ii) Develop an $O(2^n n)$ time algorithm to determines whether a graph is 3-colorable. Hint: Once a vertex is colored, there are only two possible colors left for its adjacent vertices.

The algorithm starts off numbering the nodes from 1 to n in the order that the nodes are visited by a DFS. Let Color be an array of size n which will store colors {R, G, B} assigned to nodes of the graph. Initially the color of node 1 is set to R and other colors are undefined. Now, call procedure find-3-colorable (see below). The procedure has a single parameter, x. When we call the procedure, it is the case that nodes 1, ..., n-x are already colored and the partial coloring is valid (no 2 nodes of the same color are connected by an edge). Procedure find-3-colorable(x) sets a boolean variable 3-colorable to "true" if this valid coloring can be extended to a valid coloring of the whole graph (initially 3-colorable is set to false). Since node 1 is initially colored, the procedure is initially called with x=n-1. The entry $parent(i)$ refers to the parent of node i in the DFS tree. Upon completion of find-3-colorable, the value of the variable 3-colorable indicates whether the graph is 3-colorable or now.

find-3-colorable(x)
if x=0 then set 3-colorable to true
else /* extend coloring to node n-x+1 */
{for each color c that is not color(parent(n-x+1)) do
check if there is a node $i < n - x + 1$ of color c with i adjacent to n-x+1
if there is no such conflicting node
then color(x)=c and find-3-colorable(x-1)
}
Running time: The time for the initial DFS is O(n+m). The procedure find-3-colorable(x) makes at most two recursive calls to find-3-colorable(x-1). Other then the recursive calls, the time taken by the rest of the procedure is at most linear time in size of the graph. Thus the recurrence relation for the procedure is T(x)=2T(x-1)+O(n). The solution to this recurrence is $O(2^n n)$.

4) (15 pts.) Let T be a Minimum-cost Spanning Tree of a weighted, undirected graph $G = (V, E)$. Let T' be a Minimum Spanning Tree of the new graph G obtained by one of the following changes in the edge weights:

- all edge weights are increased by a constant number c
  When the cost of every edge in G increases by c, the spanning tree T' of G' is

the same as the spanning tree T of G. The cost of the minimum cost spanning tree increases by $c(n-1)$, but the relative costs with respect to each other remained the same.

- all edge weights are decreased by a constant number c
  T=T', explanation is almost the same as above, except that the cost of every edge now decreased by c.

- the weight of a single edge known **not** to be in T is increased by c
  Since the edge was not picked for the minimum spanning tree in the first place, it will definetely not be picked if its cost increased. Thus, T=T'.

- the weight of a single edge known **not** to be in T is decreased by c
  In this case the edge with the decreased cost might be in the spanning tree of G'. To find the spanning tree T' of G', we add the edge with decreased cost into the tree T. This will produce a cycle. Now we will go through the cycle, pick the edge of the maximum cost in the cycle and remove it from the tree. The spanning tree that is left is the spanning tree T' for G'. Determining this cycle and tree T' takes $O(n)$ time. One possible implementation for finding the cycle is to root tree T and to use parent-links for generating the edges on the cycle.