Topic 14B Depth First Search Solutions
(corrected 11/14)

---

1. *(0.5)* Using one color to find discovery and finish times in a directed graph. The DFS-VISIT procedure as written uses three colors; WHITE, GRAY, and BLACK. Show that we can find discovery and finish times with two colors, WHITE and one other, and hence only one bit of storage per vertex. Do this by **modifying the relevant code, and arguing that the algorithm still assigns discovery and finish times correctly.** (Note: three colors are needed for some other applications.)

Instruction added in class: Please mark **changes with boldface** and ~~deletions with Format/Text/Strikethorough~~.

Modified Code (edit this):

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

Why it works:
The only tests of color are in line 6 of DFS and line 5 of DFS-Visit. These tests only care whether or not the vertex is white. In the existing algorithm, Gray and Black will lead to the same outcome in these tests. Therefore we need only one color.

The non-white color indicates that a node has just been discovered, so should not be processed again in lines 6-7 of DFS-Visit. Therefore we should mark the node non-white as soon as it is visited, in line 3. (If line 3 were not there, one might get back

to u via a cycle and try to process it again.) It does not matter whether we call it “gray” or “black, as long as the color becomes non-white in line 3. Having done this, line 8 serves no purpose and can be deleted.

---

2. *(1.5)* Finding connected components of an undirected graph.
Modify the relevant code to find the connected components of an **<u>undirected</u>** graph G (these are NOT SCC). Specifically, **assign to each vertex v an integer label v.cc taking on values from 1..k where k is the number of connected components.** Thus, v.cc == u.cc only if v and u are in the same connected component.

Post-class Comment: The above instructions will be modified to clarify that v.cc is the component ID. We’ll probably call it v.CID. We want to use consecutive IDs 1, 2, … k.

Modified Code (edit this):

```
DFS-Find-Components(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  component = 0
6  for each vertex u ∈ G.V
7      if u.color == WHITE
8          component++
9          DFS-VISIT-FC(G, u)
```

```
DFS-VISIT-FC(G, u)
1  time = time + 1
2  u.cc = component
3  u.d = time
4  u.color = GRAY
5  for each v ∈ G.Adj[u]
6      if v.color == WHITE
7          v.π = u
8          DFS-VISIT-FC(G, v)
9  u.color = BLACK
10 time = time + 1
11 u.f = time
```

Why it works:
Each call to DFS-VISIT-FC finds all vertices reachable from u. Since one can go over edges in either direction in an undirected graph, the vertices reached are by definition in the same connected component, so we should label them all with the same component ID. The changes to DFS-Find-Components ensure that each call to DFS-VISIT-FC is under a unique ID.

This solution leaves in the bookkeeping for parent π, discovery time and finishing time, but they could be deleted if we were not interested in those results. (See next solution.)

3. *(3)* Detecting cycles in directed and undirected graphs.

Some graph algorithms (e.g., Topological Sort) won't work on graphs with cycles. Therefore it is useful to determine whether a graph has a cycle. We don't have to find all cycles: just return TRUE if the graph has one and FALSE if not. Modify the code below to do this: the same code should work on both directed and undirected graphs. Then justify why your solution leads to the correct result and analyze time complexity for the two types of graphs.

Modified Code: (2pts)

```
  DFS-Has-Cycle(G)
  1  for each vertex u ∈ G.V
  2       u.color = WHITE
  3       u.π = NIL
  4  time = 0
  5  for each vertex u ∈ G.V
  6       if u.color == WHITE
  7            if DFS-VISIT-HC(G,
u)
  8                 return TRUE
  9  return FALSE
```

```
  DFS-VISIT-HC(G, u)
  1  time = time + 1
  2  u.d = time
  3  u.color = GRAY
  4  for each v ∈ G.Adj[u]
  5       if v.color == GRAY &&
             v != u.π
  6            return TRUE
  7       if v.color == WHITE
  8            v.π = u
  9            DFS-VISIT-HC(G, v)
 10  u.color = BLACK
 11  time = time + 1
 12  u.f = time
 13  return FALSE
```

Why it works:

We need to find the "back edges": edges that go back to a vertex that is still being actively explored. If a graph has a cycle, then the DFS search that starts from one vertex of the cycle will eventually follow a back-edge to that vertex. Conversely, if a DFS search finds a back-edge, there must be a cycle involving the vertex reached by that back-edge. This is where we need more than two colors: these will be the vertices that have been colored GRAY in line 3 of DFS-Visit, but are encountered again in a recursive call (line 10) before they are finished by lines 8-10 and colored BLACK.

With undirected graphs, there is an ambiguity (discussed in CLRS in the section on Classification of Edges): If we go over the edge {u, v} in the direction (u, v) and then in the recursive call go over it in the direction (v, u), is this a back edge? No: each edge must receive one classification, and it is already classified as a tree edge. The problem

is that our code annotates vertices, not edges, so we need to check whether we are going back to the immediate parent of u to exclude this situation. Thus the lines for π are restored.

So, as soon as we see GRAY for a vertex v that is not the parent of u we report TRUE. The rest of the code changes are just to get this report back out the top level (treating DFS-VISIT-HC as a logical predicate), and to return FALSE if it is never reported.

Note: a previous edition of these solutions had an error in that it did not check for the parent situation just discussed. This prior edition stated that the parent π, discovery, and finishing time bookkeeping is not needed and should be deleted because the early exit caused by finding a cycle will leave those values incomplete for some vertices. We now know that we need the parent π. It would be best to implement this annotation in a manner not affecting the client's view of the graph.

Asymptotic time complexity on directed graphs (0.5pt):
$O(V + E)$. This is the same as for the original DFS on directed graphs, because it is possible we will have to explore all vertices and edges. (See below for an example.)

Asymptotic time complexity on undirected graphs (0.5pt):
One might think that this is also O(V + E), but in an undirected graph with |V| vertices you cannot have more than |V| − 1 edges without having a cycle, since a tree has |E| = |V| − 1 edges and a cycle is formed if you add one more edge. So, it is impossible to see |V| edges without having detected a cycle along the way: it is O(V).

The reason that this argument does not apply to directed graphs is it is possible to have a DAG with more than |V| − 1 edges, including cases where the edge set grows faster than |V|. For example, in a DAG of n vertices where E contains (v_i, v_j) for all j > i, the edge set size |E| = n(n-1)/2 = $O(V^2)$. A small example for n=4: G = ({a, b, c, d}, {(a, b), (a, c), (a, d), (b, c), (b, d), (c, d)}}.