Homework 6 - solutions

Problem 1 (22.1-3)
Solution:
Adjacency-list representation. We need to go through the adjacency list of each vertex and “move” the edge into the vertex it points to, that is: if $v \in Adj[u]$ then there is an edge from $u$ to $v$, which in the transpose graph has to point in the other direction, that is: we should have $u \in Adj[v]$. For simplicity we construct a new graph $G' = transpose(G)$. We denote by $Adj(G)[u]$ the adjacency list of $u$ in the graph $G$, and correspondingly for $G'$. Here’s the pseudocode:

1 for each vertex $u$ in $V(G)$
2 do add $u$ to $V(G')$
3 for each vertex $u$ in $V(G)$
4 do for each $v$ in $Adj(G)[u]$
5 do add $u$ to $Adj(G')[v]$

The work is $O(V + E)$.

Problem 2 (22.2-4)
Solution:
By Theorem 22.5 on page 537 it follows that $d[v] = \delta(s, v)$, that is: the distance between $s$ and $v$. This last number is independent of any representation of the graph, hence $d[v]$ is independent of the order in which the vertices come up in the adjacency-list.

Problem 3 (22.2-6)
Solution:
This problem can be restated as deciding in $O(V + E)$ time whether an
undirected graph is bipartite. The goal is construct 2 teams A and B, such
that every edge joins a vertex from team A with one from team B. Ultimately
in order to verify that a graph is bipartite one has to verify all the edges,
hence one can hope that the problem can be solved by looking at each edge
at most once. This can be achieved the following way:

1. add an extra field $t[u]$ to each vertex $u$ s.t.

$$t[u] = \begin{cases} 
0, & \text{if } u \text{ hasn’t been assigned to any team yet} \\
1, & u \in A \\
-1, & u \in B 
\end{cases}$$

2. “Scroll” through the edges using one of the graph search algorithms we
studied and

3. assuming one end, $u$, of the current edge has been assigned to a team
already, say $A$, force the other end, $v$ in the opposite team, here $B$; if $v \in
A$ already then the graph is not bipartite, else, the algorithm finishes
when everybody has been assigned to a team in a noncontradictory
manner.

Our pseudo-code, the function $IS\_BIPARTITE(G, s)$, is a slight modification
of the Breadth-first-search algorithm. Gray vertices have already been
assigned to a team. In addition to the BFS, when we encounter WHITE
vertices in $\text{Adj}[u]$ we assign them to the opposite team (line 18), and for each
NON-WHITE vertex in $v \in \text{Adj}[u]$ we need to check that $u$ and $v$ are on
different teams (line 21), because both of them have already been assigned.
If they are not in different teams then the algorithm stops, because the graph
is not bipartite. If it never reaches line 23 (doesn’t find any mismatch) then
it is bipartite.

Of course, computing $d[u]$ and $p[u]$ is not necessary, we just kept them
for stressing that $IS\_BIPARTITE(G, s)$ lies on top of BFS. Like in the
case of BFS, the work for $IS\_BIPARTITE(G, s)$ is $O(V + E)$.

$IS\_BIPARTITE(G, s)$
1 for each vertex $u$ in $V[G]$
2 do color[$u$] <- WHITE
3 d[$u$] <- INFINITY

2
4 pi[u] <- NIL
5 t[u] <- 0
6 color[s] <- GRAY
7 d[s] <- 0
8 pi[s] <- NIL
9 t[s] <- 1
10 Q <- {s}
11 while Q != VOID
12 do u <- head[Q]
13 for each v in Adj[u]
14 do if color[v] = WHITE
15 then color[v] = GRAY
16 d[v] <- d[u]+1
17 |> v goes in the opposite team of u
18 t[v] <- (-t[u])
19 pi[v] <- u
20 ENQUEUE(Q,v)
21 else if t[v] = t[u]
22 |> v had already been assigned to the same team as u
23 then return FALSE
24 DEQUEUE(Q)
25 color[v] <- BLACK
26 return TRUE

Problem 4 (22.3-2)
Solution:
The input of the algorithm is the graph in Figure 22.6 on page 548 in the textbook, and the output is encapsulated in 3 values for each vertex, namely $d, f, \pi$ from which we can easily reconstruct the resulting Forest and classify the edges. Hence we present our result here in form of a table, which we fill in the following way:

1. The root is $q$ hence $d[q] = 1$ (enter time is 1) and $r$ has no parents, hence $\pi[q] = \text{nil}$.

2. We examine $q$'s next: $d[s] = 2, \pi[s] = q$ and since $f[q]$ is undefined yet
it means $q$ is grey, and since $s$ is white, $qs$ is a tree edge.

3. Next is $qv$, we get $d[v] = 3, \pi[v] = s$, and for the same reasons $sv$ is a tree edge.

4. We follow $v\tilde{w}$ to get $d[w] = 4, \pi[w] = v, v\tilde{w}$ is a tree edge.

5. $\tilde{w}$’s is examined next, but $s$ is grey (it has a $d$ value but not an $f$ value), hence $w$ turns black and $f[w] = 5$, making $\tilde{w}$’s a back edge (grey to grey).

6. At the next times 6 and 7 we just finish $f[v] = 6$ and $f[s] = 7$, and $s, u$ turn black.

7. The next edge is $q\tilde{t}$, a tree edge and $d[t] = 8, \pi[t] = q$.

8. Next we finish off the subtree of $t$ (no more details) and arrive at $q$ again.

9. $q$ has one more edge to check, namely $q\tilde{w}$, which is a grey-to-black edge. In order to classify it we need to compare the times $d[q]$ and $d[w]$. Since $d[q] < d[w]$ it means that $w$ is in the same tree as $q$, hence it is a descendant, therefore $q\tilde{w}$ is a forward edge (see next problem line 10), and $f[q] = 16$.

10. We start a new tree which has just $r \to u$.

11. Back edges: $\tilde{w}s, y\tilde{q}, z\tilde{x}$; Forward edges: $q\tilde{w}$; Cross edges: $r\tilde{y}, u\tilde{y}$

<table>
<thead>
<tr>
<th>vert</th>
<th>$r$</th>
<th>$q$</th>
<th>$s$</th>
<th>$t$</th>
<th>$u$</th>
<th>$v$</th>
<th>$w$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi$</td>
<td>nil</td>
<td>nil</td>
<td>$q$</td>
<td>$q$</td>
<td>$r$</td>
<td>$s$</td>
<td>$v$</td>
<td>$t$</td>
<td>$t$</td>
<td>$x$</td>
</tr>
<tr>
<td>$d$</td>
<td>17</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>$f$</td>
<td>20</td>
<td>16</td>
<td>7</td>
<td>15</td>
<td>19</td>
<td>6</td>
<td>5</td>
<td>12</td>
<td>14</td>
<td>11</td>
</tr>
</tbody>
</table>

**Problem 5 (22.3-7)**

**Solution:**

Choose the following graph described in adjacency-list format with $V = \{a, b, c\}$:
1. \(a : b, c\)

2. \(b : a\)

3. \(c:\)

A DFS search starting at \(a\) will discover \(b\) at time 2 and \(c\) at time 5, so \(d[b] < d[c]\), \(c\) is not an ancestor of \(b\) in the DFS tree and there is a path connecting \(b\) with \(c\), namely \(b \rightarrow a \rightarrow c\).

**Problem 6 (22.3-9)**

**Solution:**

We know how to first classify edges by looking at the colors of the endpoints of each edge. The colors however don’t distinguish between forward and cross edges, case in which we look at the time functions. Namely, if \(u\) is grey and \(v\) is black and \(d[u] < d[v]\) then \(v\) was visited after \(u\), hence \(v\) belongs to the same tree as \(u\), and the edge \(uv\) is a backward edge. Here’s the modified code:

```plaintext
DFS-VISIT(u)
1 color[u] <- GRAY
2 d[u] <- time <- time + 1
3 for each v in Adj[u]
4    do if color[v] = WHITE
5        then pi[v] <- u
6        PRINT 'u v is a tree edge'
7        DFS-VISIT(v)
8    else if color[v] = GREY
9        then PRINT 'u v is a back edge'
10       else if d[u] < d[v]
11          then PRINT 'u v is a forward edge'
12          else PRINT 'u v is a cross edge'
13 color[u] <- BLACK
14 f[u] <- time <- time + 1
```