Solutions for Chapter 22

Zhixiang Zhu
zzxiang21cn@hotmail.com

December 8, 2012

Solution to Exercise 21.1-1

Computing the out-degree of every vertex takes $\Theta(V+E)$. Computing the in-degrees also takes $\Theta(V+E)$.

Solution to Exercise 22.1-3

Just traverse all the edges and do the reversion. It takes $\Theta(V+E)$ for the adjacency-list representation, and $\Theta(V^2)$ for the adjacency-matrix representation.

Solution to Exercise 22.1-4

It is necessary to traverse the adjacency-list of G to complete the task. The basic idea is:

1. Create an empty adjacency-list L for G'
2. Traverse the adjacency-list of G. When visiting each edge of G, check that whether its undirected counterpart is already contained in L, and whether it is a self-loop; if both answers are no, then insert the undirected counterpart into L.

While traversing G itself already takes $O(V+E)$ running time, we have to think of a way to make the checking of each edge in step 2 take constant time. This can be accomplished by maintaining a boolean array of size $|V|$, denoted by $A[1..|V|]$. Of course, this means that we need to number all the vertices from 1 to $|V|$. The detailed procedure is as follows:

```
SHRINK-MULTIGRAPH(G)
 1  create a graph G' consisting of V[G] but no edges
 2  create a boolean array A[1..|V|] and initialize all its elements to TRUE
 3  for each vertex u ∈ V[G]
 4        do for each vertex v ∈ Adj[u]
 5              do if A[v] and u ≠ v
 6                    then insert edge (u, v) into G'
 7                         A[v] ← FALSE
 8           for each vertex v ∈ Adj[u]
 9              do A[v] ← TRUE
10  return G'
```

Solution to Exercise 22.1-5

This solution is wrong. See Philip Bille's for the correct one.

```
SQUARE(G)
1  create a graph G^2 consisting of V[G] but no edges
2  for each vertex u ∈ V[G]
3        do for each vertex v adjacent to u in G
4              do for each vertex w adjacent to v in G
5                    do insert (u, w) into G^2
6  return G^2
```

For adjacency list representation, SQUARE takes $O(V + E^2)$ running time.

For adjacency matrix representation, line 1 takes $\Theta(V^2)$. The **for** loop in line 2 iterates for $|V| + 1$ times. The **for** loops in line 3 and line 4 also iterate from 1 to $|V|$ both, because we have to traverse all the columns in the matrix to find out the adjacent vertices of a specific vertex. However, the body of the **for** loop in line 3 is only executed for edges, so it is executed for $|E|$ times in total during the whole running procedure. And for each edge, the **for** loop in line 4 iterates from 1 to $|V|$, so the **for** loop of line 4–5 takes $\Theta(VE)$ time during the whole running procedure. Therefore, SQUARE takes $\Theta(V^2 + VE)$ running time for adjacency matrix representation.

Solution to Exercise 22.1-6

This problem can be equivalently described as: given a $|V| \times |V|$ adjacency matrix $A = (a_{ij})$, determine that whether there's a integer j where $1 \le j \le |V|$ such that $a_{ij} = 1$ for $1 \le i \le |V|$ except $i = j$, and $a_{jk} = 0$ for $1 \le k \le |V|$. The algorithm is given as follows:

```
HAS-UNIVERSAL-SINK(A)
 1  i ← j ← 1
 2  while j ≤ |V|
 3        do if a_ij = 0
 4              then j ← j + 1
 5              else  if i = j
 6                          then i ← i + 1
 7                                  j ← j + 1
 8                          else  i ← j
 9  if i > |V|
10     then return FALSE
11     else  for j ← 1 to i − 1
12                 do if a_ij = 1
13                          then return FALSE
14           for j ← 1 to |V|
15                 do if i ≠ j and a_ji = 0
16                          then return FALSE
17           return TRUE
```

To show why HAS-UNIVERSAL-SINK works correctly, we use the following loop invariants:

At the start of each iteration of the **while** loop of lines 2–8:

1. $i \le j$;
2. all the elements in $A[i, i..j-1]$ are 0;
3. in every column of the sub-matrix $A[1..i-1, 1..i-1]$, either the diagonal element being 1 or some element above the diagonal being 0.

Here's the argument of the loop invariants:

Initialization: $i = j = 1$, all the three invariants trivially hold true.

Maintenance: At the start of each iteration of the **while** loop, if $a_{ij} = 0$, i will maintain its value and j will be increased by 1, so we still have $i \leq j$; the increment of j will cause $a_{ij} = 0$ be appended to the vector $A[i, i..j-1]$, so the second variant will remain true; the sub-matrix $A[1..i-1, 1..i-1]$ will remain so the third invariant trivially holds true because of the start of the current iteration. If $a_{ij} = 1$ and $i = j$, i and j will both be increased by 1, so we will still have $i = j$; the second invariants will trivially remain true since $i > j - 1$; the sub-matrix $A[1..i-1, 1..i-1]$ will be expanded by one row at the bottom and one column on the right, with the bottom element (i.e. the diagonal element) in the newly added column being 1, so the third invariant will remain true. If $a_{ij} = 1$ and $i < j$, we will set i to the value of j, then similarly the first and the second invariants will remain true; the sub-matrix $A[1..i-1, 1..i-1]$ will be expanded by $j - i$ rows at the bottom and $j - i$ columns on the right, with the elements in the ith row of the newly added columns being 0 because of the second invariant of the start of the current iteration, so the third invariant will remain true because those elements are all above the diagonal.

Termination: When the **while** loop terminates, we will have $j = |V| + 1$. If $i = j$, then according to the third loop invariant, in every column of the adjacency matrix A, either the diagonal element being 1 or some element above the diagonal being 0, so there's no such integer j where $1 \leq j \leq |V|$ such that $a_{ij} = 1$ for $1 \leq i \leq |V|$ except $i = j$, and $a_{jk} = 0$ for $1 \leq k \leq |V|$. If $i < j$, then $1..i-1$ don't meet the requirements similarly because of the third loop invariant; $i + 1..|V|$ definitely don't meet the requirements either, because of the second loop invariant. So i is the only possible qualified integer. The statements after line 8 checks the qualification of i. If $a_{ki} = 1$ for $1 \leq k \leq |V|$ but $k \neq i$, and $a_{ik} = 0$ for $1 \leq k \leq |V|$, then i is the qualified integer, otherwise there's no such integer.

During each iteration of the **while** loop, we increase either i or j, or both. The loop starts when $i = j = 1$, and terminates when j reaches $|V|$, so it iterates for at most $2|V| - 1$ times. This procedure can be viewed as maintaining a cursor which traverse from the upper left corner of A to the lower right corner. During each iteration we move the cursor closer to the lower-right corner. The two **for** loops after line 8 both iterate for at most $|V| + 1$ times. Therefore, the total running time of HAS-UNIVERSAL-SINK is $\Theta(V)$, which is also $O(V)$.

Solution to Exercise 22.1-7

Suppose that $C = BB^T$, then we have $c_{ij} = \sum_{k=1}^{|E|} b_{ik}b_{jk}$. In the case of $i = j$, we have b_{ik}^2 being either 0 or 1, so that $c_{ii} = \sum_{k=1}^{|E|} b_{ik}^2 =$ the number of edges leaving or entering vertex i, i.e. the sum of in-degree and out-degree of vertex i. In the case of $i \neq j$, observe that b_{ik} and b_{jk} cannot be both -1 or both 1, because it is impossible for an edge to leave or enter two vertices, so $b_{ik}b_{jk}$ is either 0 or -1. The value being -1 means there's an edge leaving vertex i and entering vertex j, or an edge leaving vertex j and entering vertex i. Therefore, the absolute value of c_{ij} is the number of such edges between vertex i and vertex j.

Solution to Exercise 22.2-3

The overhead of initialization is still $O(V)$. Each vertex is still enqueued at most once and dequeued at most once. However, scanning the adjacency vertices takes $O(V)$ time for each vertex now. Thus, BFS takes $O(V + V^2) = O(V^2)$ time in this situation.

Solution to Exercise 22.3-1

Directed graph:

	WHITE	GRAY	BLACK
WHITE	tree/back/forward/cross	back/cross	cross
GRAY	tree/forward	tree/back/forward	tree/forward/cross
BLACK	impossible	back	tree/back/forward/cross

Undirected graph:

	WHITE	GRAY	BLACK
WHITE	tree/back	tree/back	impossible
GRAY	tree/back	tree/back	tree/back
BLACK	impossible	tree/back	tree/back

Solution to Exercise 22.3-4

These arguments can be proved with theorem 22.7 (parenthesis theorem). Notice that the case $d[u] < f[u] < d[v] < f[v]$ is impossible to occur.

Solution to Exercise 22.3-5

See the proof of theorem 22.10.

Solution to Exercise 22.3-6

In order to implement a non-recursive DFS, we need to maintain in each vertex u an additional field *next-adj*$[u]$, which points to the first vertex in $Adj[u]$ or NIL if $Adj[u] = \emptyset$ when initialized.

```
DFS(G)
 1  for each vertex u ∈ V[G]
 2        do color[u] ← WHITE
 3           π[u] ← NIL
 4           next-adj[u] ← the first vertex in Adj[u]
 5  time ← 0
 6  create an empty stack S
 7  for each vertex u ∈ V[G]
 8        do if color[u] = WHITE
 9              then color[u] ← GRAY
10                   d[u] ← time ← time +1
11                   PUSH(S, u)
12                   repeat v ← POP(S)
13                          if next-adj[v] = NIL
14                             then color[v] ← BLACK
15                                  f[v] ← time ← time +1
16                             else PUSH(S, v)
17                                  if color[next-adj[v]] = WHITE
18                                     then π[next-adj[v]] ← v
19                                          color[next-adj[v]] ← GRAY
20                                          d[next-adj[v]] ← time ← time +1
21                                          PUSH(S, next-adj[v])
22                                  next-adj[v] ← next vertex in Adj[v]
23                     until STACK-EMPTY(S)
```

Solution to Exercise 22.3-9

To print out every edge together with its type in the directed graph G, modify the **for** loop of lines 4-7 of DFS-Visit as follows:

```
 4  for each v ∈ Adj[u]
 5        do if color[v] = WHITE
 6              then π[v] ← u
 7                   print (u, v), "tree edge"
 8                   DFS-Visit(v)
 9           elseif color[v] = GRAY
10              then print (u, v), "back edge"
11           elseif d[u] < d[v]
12              then print (u, v), "forward edge"
13           else print (u, v), "cross edge"
```

If G is undirected, the modification is as follows:

```
4  for each v ∈ Adj[u]
5        do if color[v] = WHITE
6              then π[v] ← u
7                   print (u, v), "tree edge"
8                   DFS-Visit(v)
9              else  print (u, v), "back edge"
```

Solution to Exercise 22.4-2

The idea is to perform DFS-Visit on s. We maintain in each vertex a boolean member b which indicates whether there's a path from this vertex to t, and which is initialized as FALSE.

```
Count-Number-Of-Paths(G, s, t)
1  for each vertex u ∈ V[G]
2        do color[u] ← WHITE
3           b[u] ← FALSE
4  n ← 0
5  DFS-Visit'(s)
6  return n
```

```
DFS-Visit'(u)
 1  color[u] ← GRAY
 2  if u = t
 3     then b[u] ← TRUE
 4          n ← n + 1
 5     else  for each v ∈ Adj[u]
 6                do if color[v] = WHITE
 7                      then DFS-Visit'(v)
 8                           if b[v] = TRUE
 9                              then b[u] ← TRUE
10                   elseif b[v] = TRUE
11                      then b[u] ← TRUE
12                           n ← n + 1
13  color[u] ← BLACK
```

The running time of Count-Number-Of-Paths is $O(V + E)$.

Solution to Exercise 22.4-4

Using TOPOLOGICAL-SORT, only back edges in G are "bad" (from lemma 22.11). Suppose that there is an algorithm that can get fewer "bad" edges than TOPOLOGICAL-SORT does, then there must be at least one cycle in which all the edges are "good" in the better result. If we consider all the cycles which contain just one back edge each (i.e. all other edges in the cycle are tree edges), there must be at least one such cycle whose edges are all "good" in the better result. Let's say (v, u) is a back edge in such a cycle, then v must appear before u in the better ordering. Suppose the path from u to v is $(u, s_1), (s_1, s_2), \ldots, (s_{n-1}, s_n), (s_n, v)$, where $n \geq 1$. If all these edges are "good", we would have u appearing before s_1, s_1 appearing before s_2, and so on, then at last s_n appearing before v, so that u appears before v, which is a contradiction. Therefore, such better algorithm does not exist, TOPOLOGICAL-SORT produces the minimum number of "bad" edges.

Solution to Exercise 22.4-5

We can use a modified version of BFS:

```
TOPOLOGICAL-SORT'(G)
 1  Q ← ∅
 2  for each vertex u ∈ V[G]
 3        do if in-degree[u] = 0
 4              then ENQUEUE(Q, u)
 5  while Q ≠ ∅
 6        do u ← DEQUEUE(Q)
 7           insert u into the end of a linked list
 8           for each v ∈ Adj[u]
 9                 do remove (u, v) from G
10                    if in-degree[v] = 0
11                       then ENQUEUE(Q, v)
12  return the linked list of vertices
```

If G has cycles, the vertices in the cycles won't be returned in the linked list, i.e. they won't appear in the ordering.

Solution to Exercise 22.5-1

Adding a reverse of an edge in the component graph G^{SCC} can decrease the number of strongly connected components of G by 1.

Solution to Problem 22-2

Because G is a connected undirected graph, we have $|V| \leq |E| + 1$. Therefore, $O(V + E) = O(E)$.

a. This is trivial.

b. This is trivial.

c. The idea is to use a modified version of DFS-VISIT. Notice that we won't know the final value of $low[v]$ until the whole subtree of v has been visited.

COMPUTE-LOW(u)

```
 1  color[u] ← GRAY
 2  time ← time +1        ▷ time is set to 0 before the whole recursive procedure
 3  d[u] ← time
 4  low[u] ← d[u]
 5  for each v ∈ Adj[u]
 6        do if color[v] = WHITE
 7              then π[v] ← u
 8                   COMPUTE-LOW(v)
 9                   if low[v] < low[u]
10                      then low[u] ← low[v]
11             elseif π[u] ≠ NIL and d[v] < low[π[u]]
12                then low[π[u]] ← d[v]
13  color ← BLACK
14  f[u] ← time ← time +1
```

d. Find v where $low[v] = d[v]$.

e. An edge (u, v) of G is a bridge **only if** it does not lie on any simple cycle of G: If there's a simple cycle $< u, v, \ldots, u >$ (denoted by $u \to v \overset{P}{\rightsquigarrow} u$ in G and you remove (u, v), any path $u' \overset{P'}{\rightsquigarrow} v'$ that contains (u, v) before the removal can replace (u, v) with $u \overset{P^r}{\rightsquigarrow} v$ where P^r is the reverse of path P, so that the connection is not affected.

An edge (u, v) of G is a bridge **if** it does not lie on any simple cycle of G: If (u, v) does not lie on any simple cycle, its removal will break any path that contains (u, v) (because if there's another substitution path $u \overset{P}{\rightsquigarrow} v$, (u, v) is on a simple cycle, which leads to a contradiction), so that the connection is broken.

f. If $low[v] < d[v]$ and there's no back edge from v, $(\pi[v], v)$ is in a simple cycle and is not a bridge; otherwise $(\pi[v], v)$ is a bridge.

g. The definition of 'partition' here meas a nonbridge edge must lie on exactly one biconnected component of G. From ***e*** and the definition of biconnected components (*bcc*), a nonbridge edge must lie on at least one *bcc*. Now suppose that we have three nonbridge edges $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$ and $e_3 = (u_3, v_3)$. Also suppose that e_1 and e_2 lie on the same *bcc*, e_1 and e_3 lie on the same *bcc*, but e_2 and e_3 lie on different *bcc*s, so that e_1 lies in two *bcc*s. From the definition of *bcc*, we can suppose that there're two simple cycles $u_1 \to v_1 \overset{P}{\rightsquigarrow} u_2 \to v_2 \overset{Q}{\rightsquigarrow} u_1$ and $u_1 \to v_1 \overset{R}{\rightsquigarrow} u_3 \to v_3 \overset{S}{\rightsquigarrow} u_1$, but then we have $u_2 \to v_2 \overset{Q}{\rightsquigarrow} u_1 \overset{S}{\rightsquigarrow} v_3 \to u_3 \overset{R}{\rightsquigarrow} v_1 \overset{P}{\rightsquigarrow} u_2$, so that $e_2 = (u_2, v_2)$ and $e_3 = (u_3, v_3)$ are also in the same *bcc*, and e_1 lies on just one *bcc*.

h. Remove all the bridges, and then run DFS.

Solution to Problem 22-3

a. See instructor's manual, but there's an error in the nonconstructive proof: $x \in V'' \cup V_C$ should be changed to $x \in V'' \cap V_C$.