论题 2-16 作业

姓名：陈劭源　　　　学号：161240004

1 [TC] Problem 22.1-3

For adjacency-list representation:

TRANSPOSE(G)

let $Adj[1\,..\,|V(G)|]$ be a new array of lists
for $i = 1$ **to** $|V(G)|$
　　for each e in $G.Adj[i]$
　　　　$Adj[e].insert(i)$
return graph $(V(G), Adj)$

For adjacency-matrix representation:

TRANSPOSE(G)

let $A[1\,..\,|V(G)|, 1\,..\,|V(G)|]$ be a new array
for $i = 1$ **to** $|V(G)|$
　　for $j = 1$ **to** $|V(G)|$
　　　　$A(i, j) = G.A(j, i)$
return graph $(V(G), A)$

2 [TC] Problem 22.1-8

If we use hash table with collision resolution by chaining, the expected time to determine whether an edge is in the graph is $O(1+\alpha)$, where α is the load factor. (We will not adopt open addressing, because it is no better than adjacency-matrix, i.e. direct-address tables)

The disadvantage of this scheme is that, we still need a great number of memory space, even if the graph is very sparse.

Instead of a hash table, we can use binary search tree as array entry $Adj[u]$, containing the vertices v for which $(u, v) \in E$. This alternative requires $O(\log \mathrm{od}(u))$ time for determining whether an edge is in the graph, where $\mathrm{od}(u)$ is the out-degree of u, usually worse than hash table.

3 [TC] Problem 22.2-3

In BFS, whether a vertex is black or gray does not affect the order in which the statements are executed, so we can remove line 18[1], i.e. leave the vertex gray after all its white adjacent nodes have been inserted to the

[1] According to the errata sheet (`http://www.cs.dartmouth.edu/~thc/clrs-bugs/bugs-3e.php`), the problem should be "... if line 18 was removed". This error has been corrected in the third printing of this book.

queue, without changing the result the procedure produces.

4 [TC] Problem 22.2-4

Instead of going through the adjacency list, we have to go through a row of the adjacency matrix to determine the edges in the graph, which takes a running time of $O(|V|^2)$.

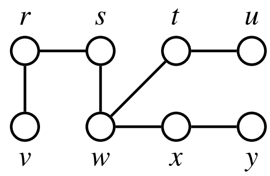
5 [TC] Problem 22.2-5

By Theorem 22.5, after BFS is done, $u.d$ is the distance from the source to u. Changing the order of the vertices in each adjacency list does not change the graph the lists represents, thus does not change $u.d$.

For the graph shown in Figure 22.3, if the adjacency lists are:

vertex	r	s	t	u	v	w	x	y
Adj	v,s	r,w	w,x,u	t,x,y	r	s,t,x	w,t,u,y	x,u

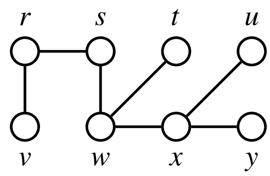
the breadth-first tree computed by BFS (from s) is:



If we change to the order of the vertices in the adjacency list of x:

vertex	r	s	t	u	v	w	x	y
Adj	v,s	r,w	w,x,u	t,x,y	r	s,x,t	w,t,u,y	x,u

the breadth-first tree will be:



6 [TC] Problem 22.3-6

For every edge (u,v) $(u.d < v.d)$, if (u,v) is encountered first, then v must remain unvisited, because (v,u) will be encountered when visiting v. Hence (u,v) is a tree edge. If (v,u) is encountered first, then both u,v are visited, and thus (u,v) is a back edge.

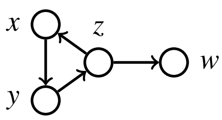
Since, in an undirected graph, every edge is either a tree edge or a black edge, we conclude that (u,v) $(u.d < v.d)$ is encountered first iff (u,v) is a tree edge, and (v,u) $(u.d < v.d)$ is encountered first iff (u,v) is a back edge. Therefore, these two schemes are equivalent.

7 [TC] Problem 22.3-7

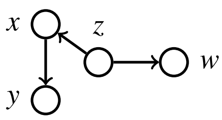
```
DFS(G)
time = 0
let S be a new stack
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
    S.push(u)
while S is not empty
    u = S.pop()
    if u.color == WHITE
        time = time + 1
        u.d = time
        u.color = GRAY
        S.push(u)
        for each vertex v ∈ G.Adj[u]
            if v.color == WHITE
                v.π = u
                S.push(v)
    elseif u.color == GRAY
        u.color = BLACK
        time = time + 1
        u.f = time
```

8 [TC] Problem 22.3-8

Consider the following graph:



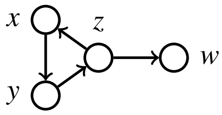
In this graph, there exists a path from y to w. If we perform depth-first search on this graph from z in the order z, x, y, w, the depth-first forest is:



In such order of DFS, $y.d < w.d$, however, w is not a descendant of y in this forest.

9 [TC] Problem 22.3-9

Consider the following graph:



In this graph, there exists a path from y to w. If we perform depth-first search on this graph from z in the order z, x, y, w, w is visited after the adjacency list of y has been examined, i.e. $w.d > y.f$.

10 [TC] Problem 22.3-12

```
DFS(G)
    ......
    c = 0
    for each vertex u ∈ G.V
        if u.color == WHITE
            c = c + 1
            DFS-VISIT(G, u, c)
```

```
DFS-VISIT(G, u, c)
    ......
    u.cc = c
    for each vertex v ∈ G.Adj[u]
        if u.color == WHITE
            v.π = u
            DFS-VISIT(G, v, c)
    ......
```

11 [TC] Problem 22.4-2

```
COUNT-SIMPLE-PATHS(G, s, t)
    for each vertex u ∈ G.V
        count[u] = 0
    count[s] = 1
    T = TOPOLOGICAL-SORT(G)
    x = T.head
    while x ≠ NIL
        for each vertex v ∈ G.Adj[x.key]
            count[v] = count[v] + count[x.key]
        x = x.next
    return count[t]
```

12 [TC] Problem 22.4-3

We can modify DFS to determine whether the undirected graph contains a cycle.

```
DFS(G)
    ......
    for each vertex u ∈ G.V
        if u.color == WHITE
        DFS-VISIT(G, u, NIL)
    print "does not contain a cycle"
```

```
DFS-VISIT(G, u, p)
    ......
    u.cc = c
    for each vertex v ∈ G.Adj[u]
        if u.color == WHITE
            v.π = u
            DFS-VISIT(G, v, u)
        elseif v ≠ p
            print "contains a cycle"
            terminate DFS
    ......
```

In this algorithm, every vertex is visited at most once. If a vertex is to be visited for the second time, it means that the graph contains a cycle, and the procedure will be immediately terminated. Hence the algorithm run in $O(|V|)$ time, independent of $|E|$.

13 [TC] Problem 22.5-5

```
COMPONENT-GRAPH(G)
    compute the strongly connected components of G
    assign each vertex v an integer label v.scc between 1 and k, denoting which strongly
    connected components v belongs to, where k is the number of the components.
    G^scc.V = {1, 2, ···, k}
    let edges[1..k] be a new array of lists // adjacent lists with duplicate edges and loops
    for each edge (u, v) ∈ G.E
        edges[u.scc].insert(v.scc)
    let mark[1..k] be a new array
    for i = 1 to k
        mark[i] = 0
    for i = 1 to k
        for each vertex j ∈ edges[i]
            if i ≠ j and mark[j] ≠ i
                G^scc.Adj[i].insert(j)
                mark[j] = i
    return G^scc
```

In this procedure, line 1, line 2, line 3, line 4-6, line 7-14 cost $O(|V|+|E|)$ time respectively, and thus the algorithm takes a total running time of $O(|V|+|E|)$.

14 [TC] Problem 22.5-6

IS-SEMICONNECTED(G)

1 $G^{scc} =$ COMPONENT-GRAPH(G)
2 $T =$ TOPOLOGICAL-SORT(G^{scc})
3 **for** $i = 1$ **to** $T.length - 1$
4   **if** $(T[i], T[i+1]) \notin G^{scc}.E$
5   **return** FALSE
6 **return** TRUE

In this procedure, line 1, line 2 takes a running time of $O(|V|+|E|)$, respectively. Line 3-5 takes a running time of $O(1+\mathrm{od}(v))$ for each vertex v to verify whether or not $(T[i], T[i+1]) \in G^{scc}.E$, where $\mathrm{od}(v)$ is the out-degree of v, and $O(|V|+|E|)$ in sum. Therefore, the total running time is $O(|V|+|E|)$.

We break the proof of the correctness of this algorithm into the following two steps:

Step 1: A graph G is semiconnected if and only if its component graph G^{scc} is semiconnected.

Proof 'if': for every two distinct nodes u, v in G, if they are in the same strongly connected component, then they are semiconnected; otherwise, let $u \in C_1$, $v \in C_2$, where C_1, C_2 are two distinct strongly connected components. Since the component graph is semiconnected, we have $C_1 \rightsquigarrow C_2$ or $C_2 \rightsquigarrow C_1$. Without loss of generality, assume $C_1 \rightsquigarrow C_2$. For every edge (C_i, C_j) in the path $C_1 \rightsquigarrow C_2$, replace it with two vertices $u \in C_i, v \in C_j$, where $(u, v) \in G$. For every two incident edges $(C_i, C_j), (C_j, C_k)$ and their corresponding vertices $u \in C_i, v, w \in C_j, x \in C_k$, since v and w are in the same component, we can add a path $v \rightsquigarrow w$ between v and w. Finally, we get a path from u to v. Therefore, G is semiconnected.

'only if': for every two distinct nodes C_1, C_2 in G^{scc}, choose two elements $x \in C_1, y \in C_2$, then $x \rightsquigarrow y$ or $y \rightsquigarrow x$. Replace each node in $x \rightsquigarrow y$ (or $y \rightsquigarrow x$) with the component it belongs to, we obtain a path, or more precisely, a walk from C_1 to C_2 (or from C_2 to C_1), therefore G^{scc} is semiconnected.

Step 2: A DAG G is semiconnected if and only if for every two vertices u, v adjacent in the topological sort of G, they are connected by a directed edge.

Proof 'if': for every two distinct vertices u, v, assume, without loss of generality, that u appears before v in the topological sort, i.e. the topological sort is $\cdots, u, w_1, w_2, \cdots, w_n, v, \cdots$, then $(u, w_1), (w_1, w_2), \cdots, (w_n, v)$ are directed edges in G, i.e. $u \rightsquigarrow v$. Therefore, G is semiconnected.

'only if': suppose, to the contrary, that there exists two vertices u, v adjacent in topological sort G, but they are not connected by a directed edge (u, v). It is impossible that $v \rightsquigarrow u$, because u appears before v in topological sort. Therefore $u \rightsquigarrow v$. Since they are not connected by a directed edge, the path from u to v must contain at least 3 vertices, i.e. there must exist another vertex w, such that $u \rightsquigarrow w \rightsquigarrow v$. By the property of topological sort, w appears after u but before v. However, u and v are adjacent in the topological sort, which leads to contradiction.

Note that the component graph is a DAG. Combine the two steps, we obtain a complete proof of the correctness of the algorithm.