

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)

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

For adjacency-matrix representation:

TRANSPOSE(G)

```
1  let  $A[1..|V(G)|, 1..|V(G)|]$  be a new array
2  for  $i = 1$  to  $|V(G)|$ 
3      for  $j = 1$  to  $|V(G)|$ 
4           $A(i, j) = G.A(j, i)$ 
5  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 $O(\log n)$ time for determining whether an edge is in the graph, where n 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 18¹, i.e. leave the vertex gray after all its white adjacent nodes have been inserted to the

¹According 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 $O(|V|^2)$.

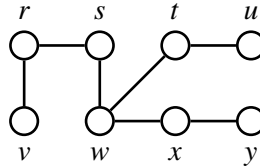
5 [TC] Problem 22.2-5

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

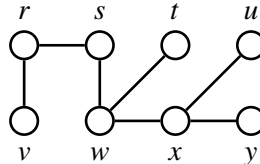
the breadth-first tree computed by BFS (from s) is:



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)

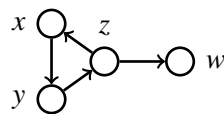
```

1  time = 0
2  let  $S$  be a new stack
3  for each vertex  $u \in G.V$ 
4       $u.color = \text{WHITE}$ 
5       $u.\pi = \text{NIL}$ 
6       $S.push(u)$ 
7  while  $S$  is not empty
8       $u = S.pop()$ 
9      if  $u.color == \text{WHITE}$ 
10          $time = time + 1$ 
11          $u.d = time$ 
12          $u.color = \text{GRAY}$ 
13          $S.push(u)$ 
14         for each vertex  $v \in G.Adj[u]$ 
15             if  $v.color == \text{WHITE}$ 
16                  $v.\pi = u$ 
17                  $S.push(v)$ 
18         elseif  $u.color == \text{GRAY}$ 
19              $u.color = \text{BLACK}$ 
20              $time = time + 1$ 
21              $u.f = time$ 

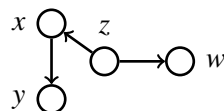
```

8 [TC] Problem 22.3-8

Consider the following graph:



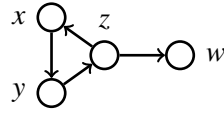
In this graph, there exists a path from y to w . If we perform depth-first search on this graph from z in the order z, x, y, w , the depth-first forest is:



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)

.....

5 $c = 0$

6 **for** each vertex $u \in G.V$

7 **if** $u.color == \text{WHITE}$

8 $c = c + 1$

9 DFS-VISIT(G, u, c)

DFS-VISIT(G, u, c)

.....

4 $u.cc = c$

5 **for** each vertex $v \in G.Adj[u]$

6 **if** $u.color == \text{WHITE}$

7 $v.\pi = u$

8 DFS-VISIT(G, v, c)

.....

11 [TC] Problem 22.4-2

COUNT-SIMPLE-PATHS(G, s, t)

1 **for** each vertex $u \in G.V$

2 $count[u] = 0$

3 $count[s] = 1$

4 $T = \text{TOPOLOGICAL-SORT}(G)$

5 $x = T.head$

6 **while** $x \neq \text{NIL}$

7 **for** each vertex $v \in G.Adj[x.key]$

8 $count[v] = count[v] + count[x.key]$

9 $x = x.next$

10 **return** $count[t]$

12 [TC] Problem 22.4-3

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

DFS(G)

.....

```
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u, \text{NIL}$ )
8  print “does not contain a cycle”
```

DFS-VISIT(G, u, p)

.....

```
4   $u.cc = c$ 
5  for each vertex  $v \in G.Adj[u]$ 
6      if  $u.color == \text{WHITE}$ 
7           $v.\pi = u$ 
8          DFS-VISIT( $G, v, u$ )
9      elseif  $v \neq p$ 
10         print “contains a cycle”
11         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)

```
1  compute the strongly connected components of  $G$ 
2  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.
3   $G^{scc}.V = \{1, 2, \dots, k\}$ 
4  for each edge  $(u, v) \in G.E$ 
5      if  $u.scc \neq v.scc$ 
6           $G^{scc}.E.insert((u.scc, v.scc))$ 
   // use radix sort to sort the edges
7  perform counting sort on  $G^{scc}.E$ , using the second endpoint of each edge as key
8  perform counting sort on  $G^{scc}.E$ , using the first endpoint of each edge as key
9  delete the consecutive duplicate edges in  $(G^{scc}.E)$  // after sorting, duplicate edges must be consecutive
   // construct adjacency lists
10 for each edge  $(u, v) \in G^{scc}.E$ 
11      $G^{scc}.Adj[u].insert(v)$ 
12 return  $G^{scc}$ 
```

In this procedure, line 1, line 2, line 3-6, line 7, line 8, line 9, line 10-11 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   $G^{scc} = \text{COMPONENT-GRAPH}(G)$ 
2   $T = \text{TOPOLOGICAL-SORT}(G^{scc})$ 
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 $O(|V| + |E|)$, line 3-4 takes a running time of $O(|V|)$, therefore the total running time is $O(|V| + |E|)$.

We break the proof of the correctness of this algorithm into the 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 G , then u and v are connected by a directed edge in G .

Proof ‘if’: for every distinct vertices u, v , assume, without loss of generality, that u appears before v in the topological sort, i.e. the topological sort is $\dots, u, w_1, w_2, \dots, w_n, v, \dots$, then $(u, w_1), (w_1, w_2), \dots, (w_n, v)$ 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 the topological sort G , but u and v are not connected by a directed edge in G . It is impossible that $v \rightsquigarrow u$, because u appears before v . 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.