3-7 traveling

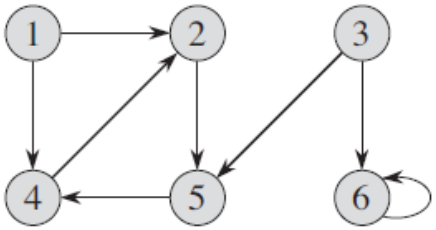
Jun Ma

majun@nju.edu.cn

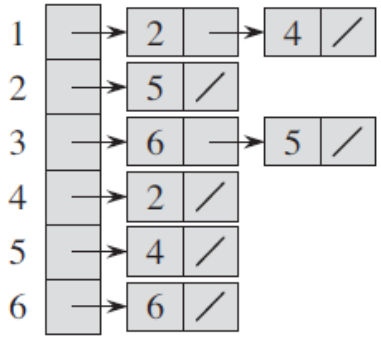
November 4, 2020

TC 22.1-3

The ***transpose*** of a directed graph $G = (V, E)$ is the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^{T} is G with all its edges reversed. Describe efficient algorithms for computing G^{T} from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.



(a)

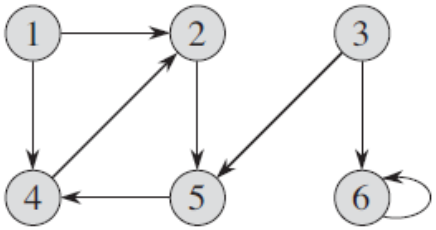


(b)

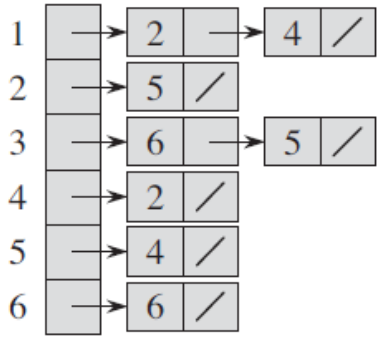
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



(a)



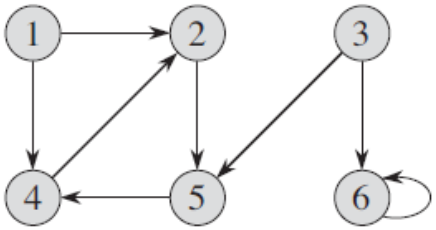
(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

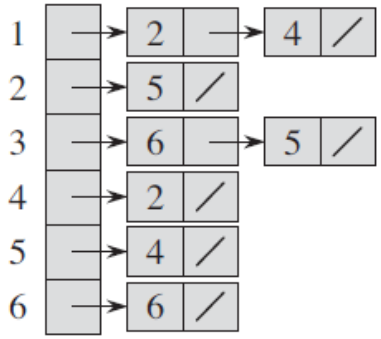
(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

- Linked list: $O(V + E)$



(a)



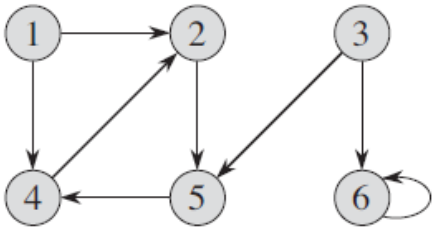
(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

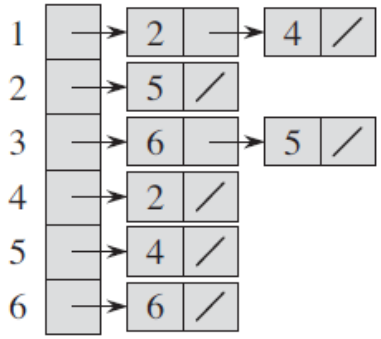
(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

- Linked list: $O(V + E)$
- Adjacency-matrix:



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

- Linked list: $O(V + E)$
- Adjacency-matrix:
  - $O(V^2)$

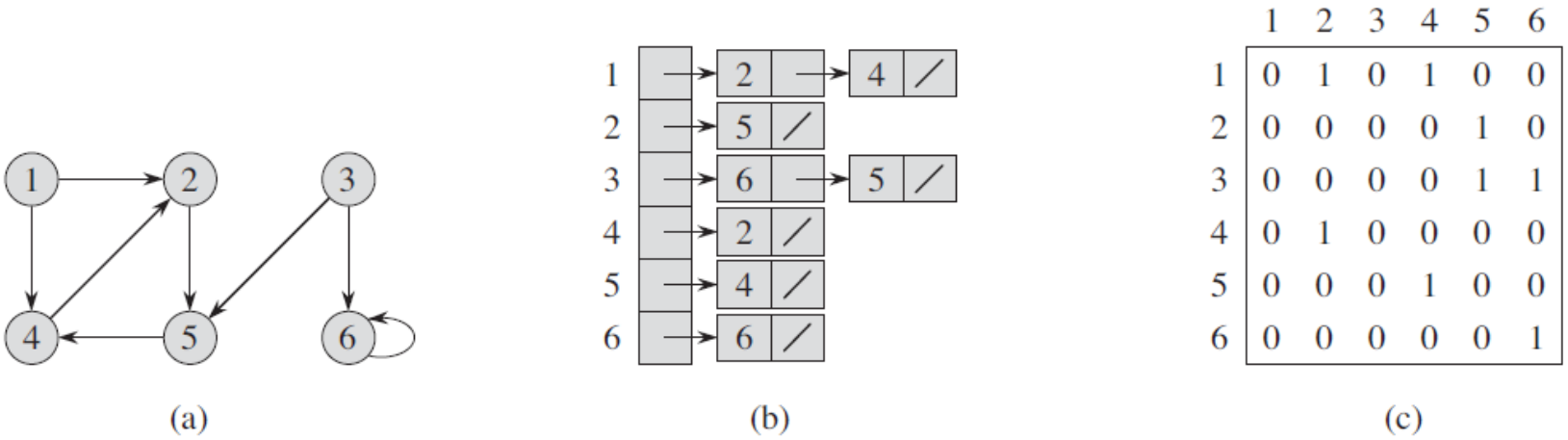


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

- Linked list: $O(V + E)$
- Adjacency-matrix:
  - $O(V^2)$
  - $O(1)$?

TC 22.1-8

Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$. If all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph? What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared to the hash table?

- Expected time: depends on the implementation of hashtable. Can be assumed as $O(1)$.
- Disadvantages
  - Extra space
- Alternate data structure: AVL, RB-Tree, etc.
  - Space: $O(n)$
  - Time: $O(\lg n)$

TC 22.2-3

Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if line 18 was removed.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

TC 22.2-3

Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if line 18 was removed.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

TC 22.2-3

Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if line 18 was removed.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

TC 22.2-4

What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

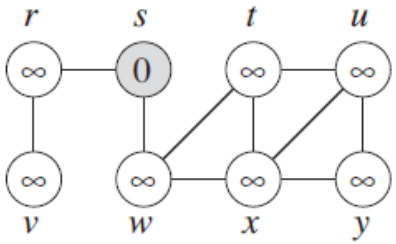
$$O(V + V^2) = O(V^2)$$

TC 22.2-5

Argue that in a breadth-first search, the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Using Figure 22.3 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

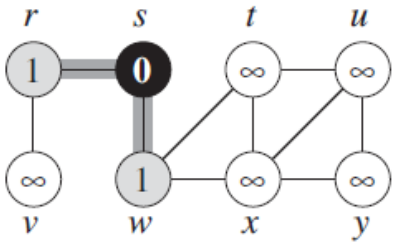
```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

(a)



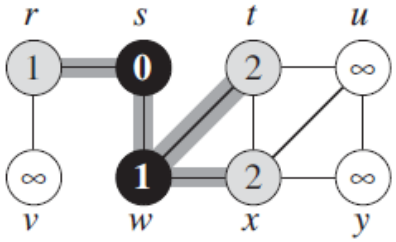
Q | s | | | |
0

(b)



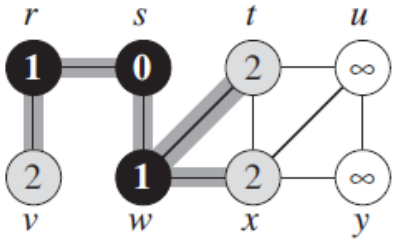
Q | w | r | | |
1 1

(c)



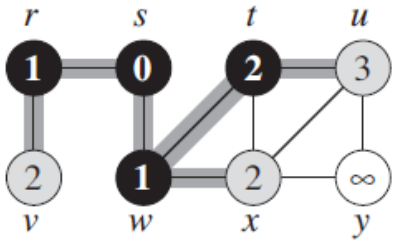
Q | r | t | x | |
1 2 2

(d)



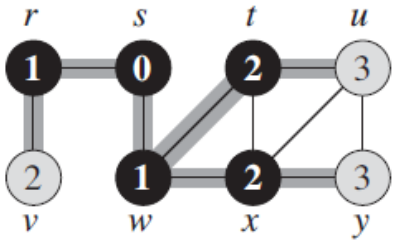
Q | t | x | v | |
2 2 2

(e)



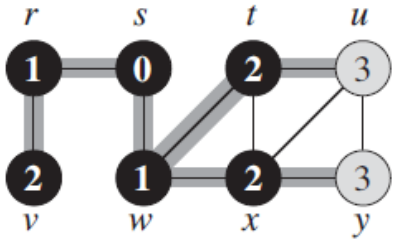
Q | x | v | u | |
2 2 3

(f)



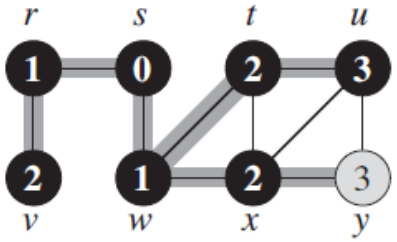
Q | v | u | y | |
2 3 3

(g)



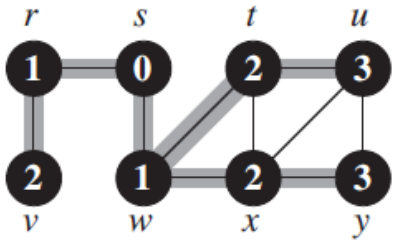
Q | u | y | | |
3 3

(h)



Q | y | | | |
3

(i)



Q $\emptyset$

TC 22.3-6

Show that in an undirected graph, classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search is equivalent to classifying it according to the ordering of the four types in the classification scheme.

TC 22.3-9

Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$.

```
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
```

TC 22.3-9

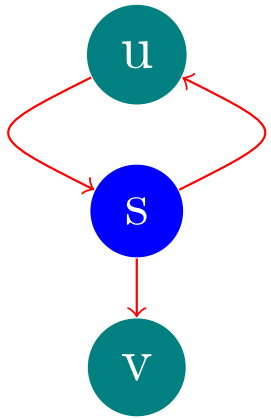
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$.

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
```



TC 22.3-12

Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

TC 22.3-12

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

```
DFS_VISIT(G, u, cc)
   u.cc = cc
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, u, cc)
8  u.color = BLACK
9  time = time + 1
0  u.f = time
```

TC 22.4-2

Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex p to vertex v: pov, $poryv$, $posryv$, and $psryv$. (Your algorithm needs only to count the simple paths, not list them.)

TC 22.4-2

```
1: procedure PathCounter(G, s, t)
2:     if s = t then
3:         return 1;
4:     if s.color! = white then
5:         return s.count;
6:     else
7:         s.color ← gray
8:         for each v ∈ s.Ajc do
9:             c ←PathCounter(G, v, t)
10:            s.count ← s.count + c
11:    s.color ← black
12:    return s.count
```

TC 22.4-3

Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

Theorem 22.10

In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

- Simply run DFS. If find a back edge, there is a cycle.
- The complexity is $O(V)$ instead of $O(E + V)$.

TC 22.5-5

Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

TC 22.5-5

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

- For each tree T in the DFS forest, construct a vertex V_T in the component graph G_{SCC}
- For each edge $(u, v) \in G.E$, add an edge $(u.cc, v.cc)$ to G_{SCC} if it doesn't exists.

TC 22.5-7

A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

TC 22.5-7

A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

- Obtain the component graph G_{SCC} of G
- Do topological sort on G_{SCC}. Assume vertexes in G_{SCC} are sorted as $V_1, V_2, ..., V_m$
- Check if for every i, there is edge (V_i, V_{i+1}).

Lemma (1)

A graph is semi-connected ***iff*** *its corresponding component graph is semi-connected.*

Lemma (2)

A DAG is semi-connected in a topological sort ***iff*** *for each i, there there is an edge (v_i, v_{i+1})*

Proof of Lemma (2)

$\Rightarrow$.

- Assume $\exists i$, s.t. $(v_i, v_{i+1}) \notin G.E$.
- There is no such path $(v_{i+1} \rightsquigarrow v_i)$
- So, G cannot be semi-connected, the assumption is wrong.

□

$\Leftarrow$.

Obviously! □


Thank
You!