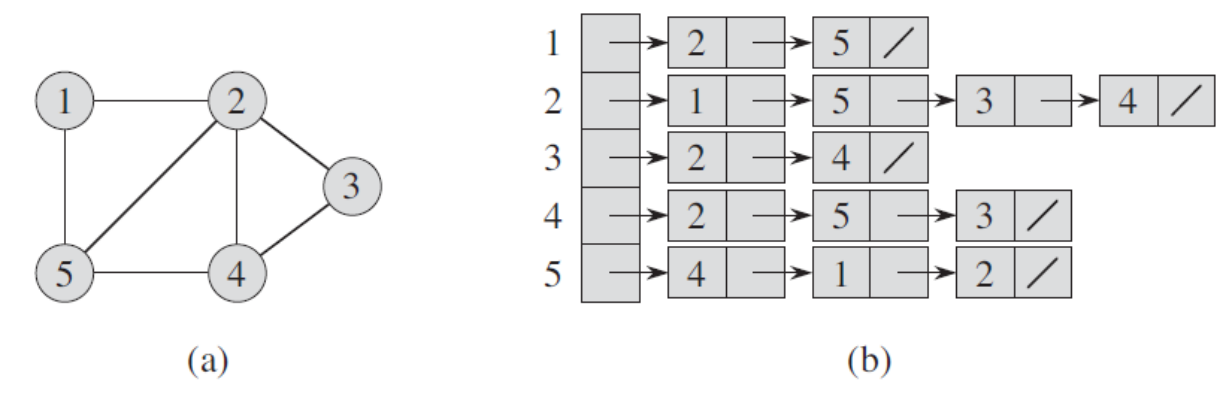
Elementary Graph Algorithms

交通大學 資訊工程系

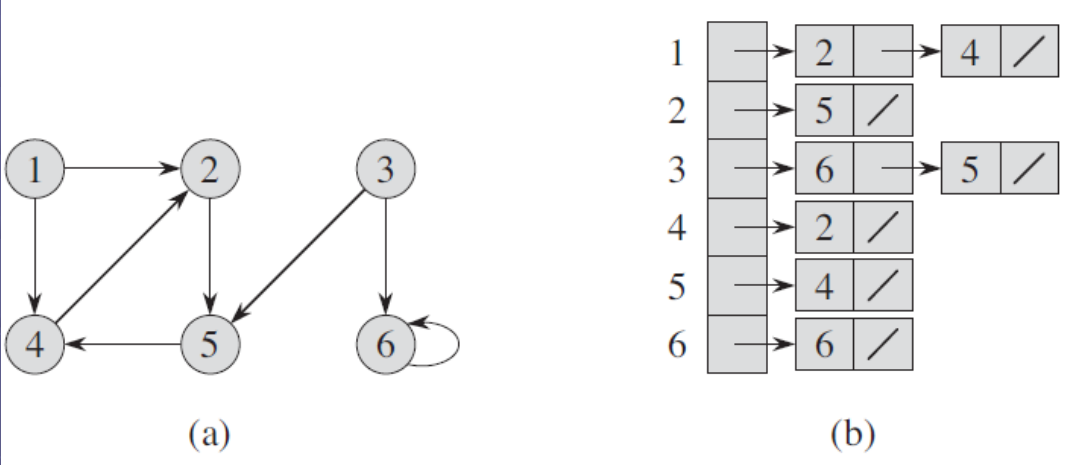
Representations of Graphs[1]

- Adjacency-list representation
  - For sparse graphs
  - A graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V.
  - $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$.
  - Example: Undirected graph



Representations of Graphs[2]

- Example: Directed graph



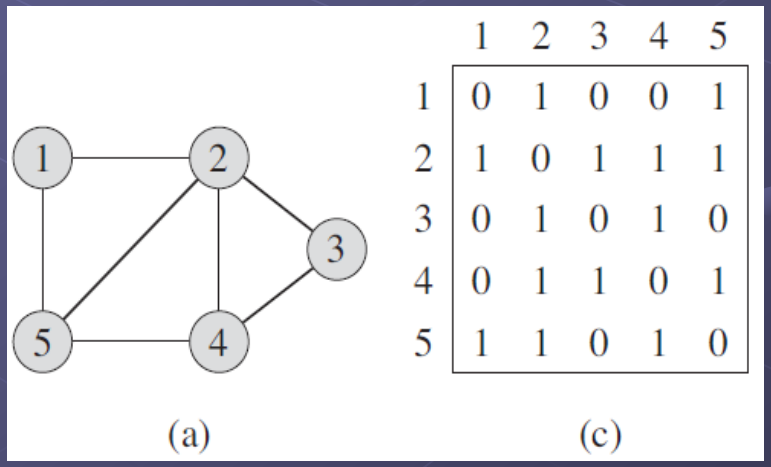
- The amount of memory required is $\Theta(V + E)$.
- Provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$.

Representations of Graphs[3]

- Adjacency-matrix representation
  - A graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

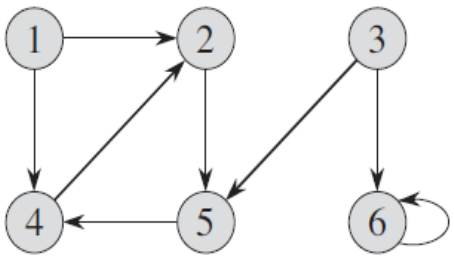
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

  - Example: Undirected graph



Representations of Graphs[4]

- **Example: Directed graph**



(a)

	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)

- **Requires $\Theta(V^2)$ memory.**
- **For undirected graph requires only 1 bit per entry.**

Breadth-first Search[1]

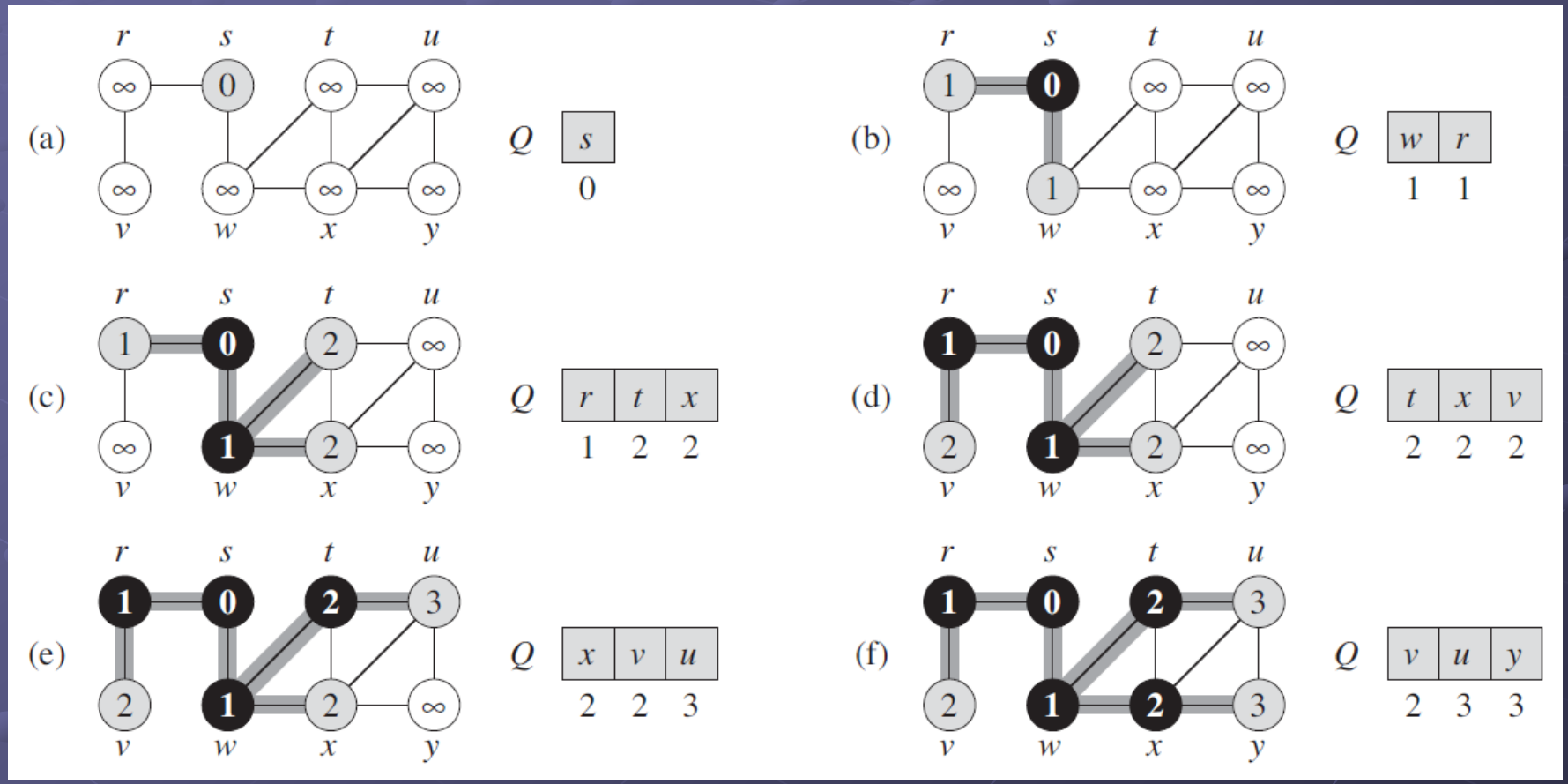
- **Breadth-first search**
  - Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s.
  - Explores the edges of G to "discover" every vertex that is reachable from s.
  - Computes the distance from s to each reachable vertex.
  - Produces the breadth-first tree with root s that contains all reachable vertices.
  - The algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.
  - Breadth-first search colors each vertex white, gray, or black.

Breadth-first Search[2]

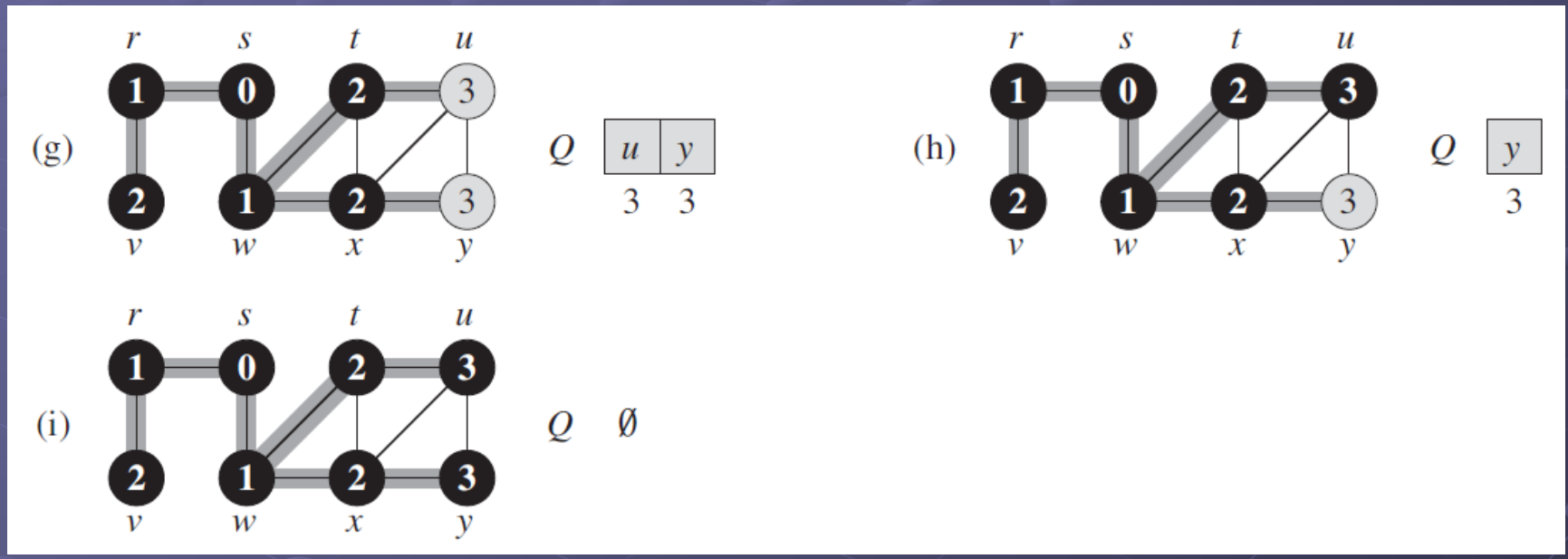
☞ **The algorithm**

```
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
```

Breadth-first Search[3]



Breadth-first Search[4]



- Analysis
  - The total running time of BFS is $O(V + E)$.

Breadth-first Search[5]

- ***Shortest-path distance*** $\delta(s, v)$ **from** s **to** v**.**
  - **The minimum number of edges in any path from vertex** s **to vertex** v**.**
- ***Lemma 22.1***
  - **Let** $G = (V, E)$ **be a directed or undirected graph, and let** $s \in V$ **be an arbitrary vertex. Then, for any edge** $(u, v) \in E$**,**

$$\delta(s, v) \leq \delta(s, u) + 1$$

  - ***Proof:***
    - **Consider** u **is reachable and unreachable from** s**.**

Breadth-first Search[6]

- *Lemma 22.2*
  - Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.
  - *Proof:* By induction on the number of ENQUEUE operations.
- *Lemma 22.3*
  - Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \cdots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \cdots, r - 1$.

Breadth-first Search[7]

- **Proof: By induction on the number of queue operations.**

☞ ***Corollary 22.4***

- **Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.**
- ***Proof:***
  - **Immediate from Lemma 22.3 and the property that each vertex receives a finite d value at most once during the course of BFS.**

Breadth-first Search[8]

☞ ***Theorem 22.5 (Correctness of breadth-first search)***

- **Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.**

Breadth-first Trees[1]

☞ ***Predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ **of** G

- For a graph $G = (V, E)$ with source s

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

and

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\} .$$

- The predecessor subgraph G_π is a ***breadth-first tree*** if
  - V_π consists of the vertices reachable from s and,
  - For all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v in G.
- $|E_\pi| = |V_\pi| - 1$
- ***Tree edges***: The edges in E_π.

Breadth-first Trees[2]

- *Lemma 22.6*
  - When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.
- Prints out the vertices on a shortest path from s to v.

```
PRINT-PATH(G, s, v)
1  if v == s
2       print s
3  elseif v.π == NIL
4       print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6       print v
```

Depth-first Search[1]

- Search "deeper" in the graph whenever possible.
  - Explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.
  - Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.
- Predecessor subgraph
  - May be composed of several trees.

$$G_\pi = (V, E_\pi), \text{ where}$$

$$E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$$

Depth-first Search^2

- Depth-first search colors vertices during the search
  - Initially white
  - Grayed when it is discovered in the search
  - Blackened when it is finished
- Each vertex v has two *timestamps*:
  - The first timestamp $v.d$ records when v is first discovered.
  - The second timestamp $v.f$ records when the search finishes examining v's adjacency list.
  - These timestamps are integers between 1 and $2|V|$.
  - $u.d < u.f$

Depth-first Search[3]

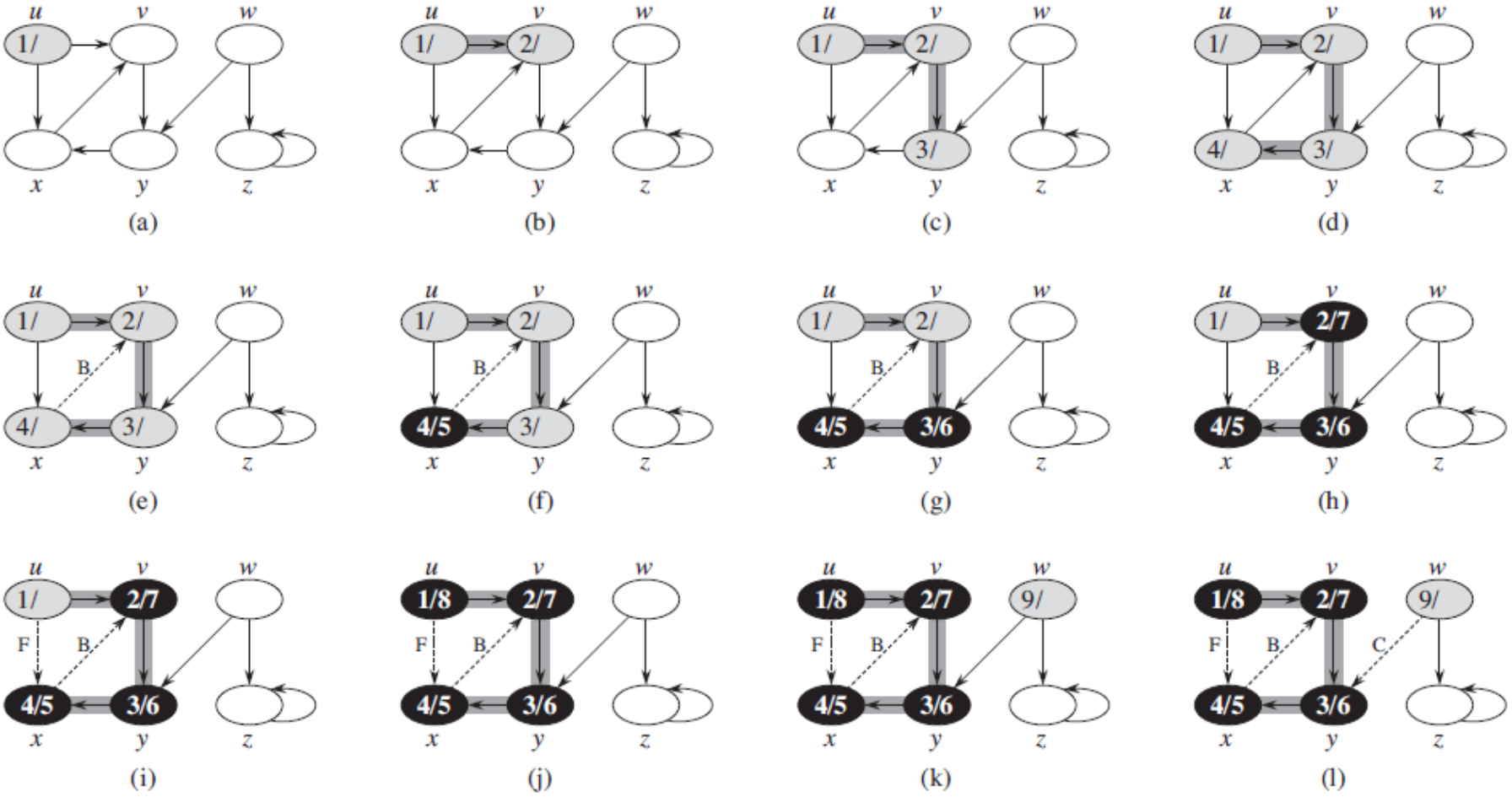
- **The basic depth-first search algorithm**

```
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)
```

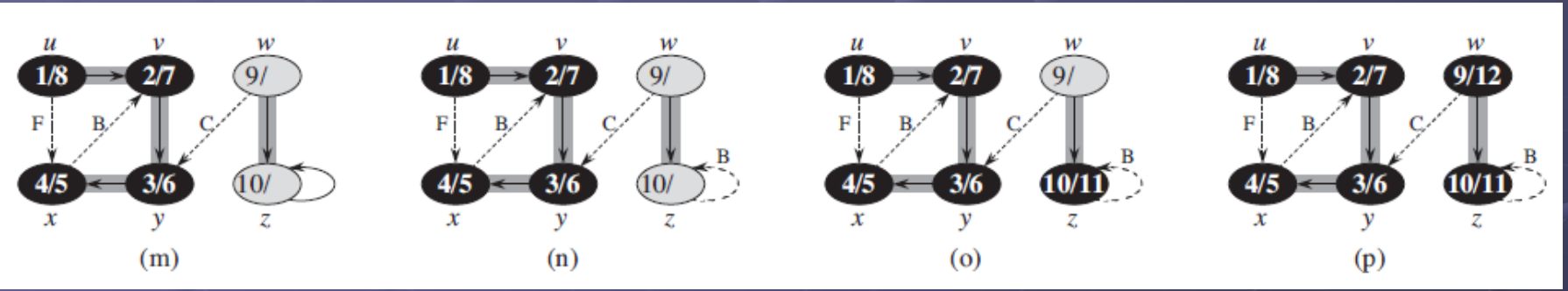
Depth-first Search[4]

```
DFS-VISIT(G, u)
 1  time = time + 1                  // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]            // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK                  // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

Depth-first Search[5]



Depth-first Search[6]



- The running time of DFS is therefore $\Theta(V + E)$.
- Properties of depth-first search
  - DFS yields valuable information about the structure of a graph.
  - The predecessor subgraph G_π does indeed form a forest of trees.

Depth-first Search[7]

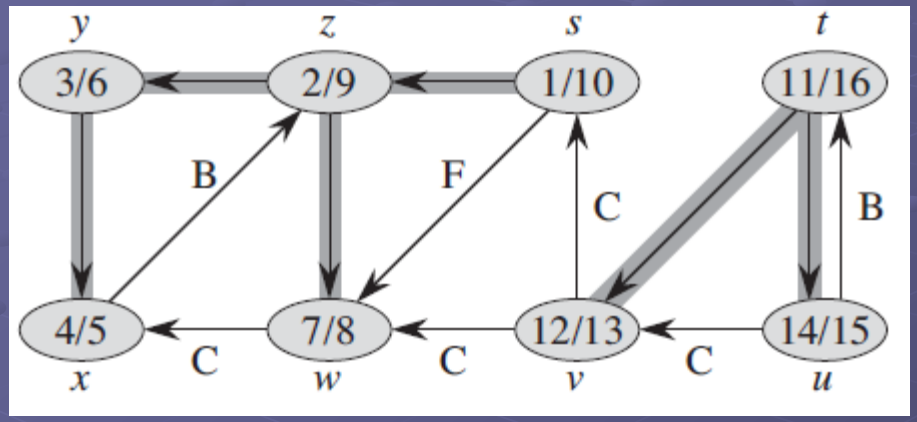
- Vertex v is a descendent of vertex u in the depth-first search if and only if v is discovered during the time in which u is gray.
- Discovery and finishing times have *parenthesis structure*.

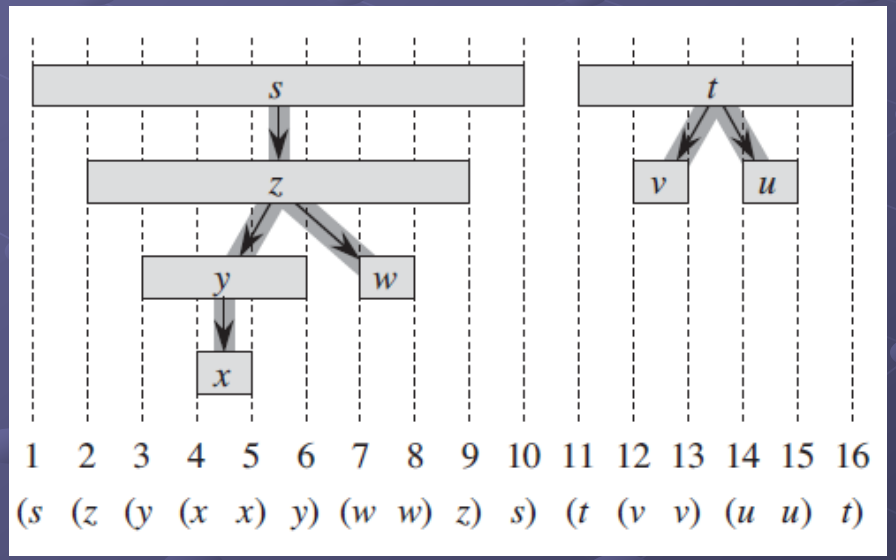
☞ ***Theorem 22.7*** **(*Parenthesis theorem*)**

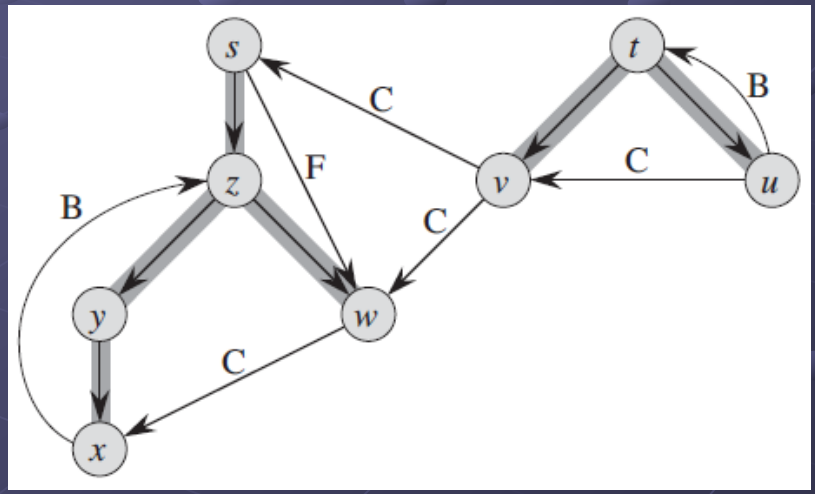
In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

Depth-first Search[8]







Depth-first Search[9]

- ***Corollary 22.8*** **(*Nesting of descendants' intervals*)**
  - **Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $u.d < v.d < v.f < u.f$.**
- ***Theorem 22.9*** **(*White-path theorem*)**

> In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

- **Classification of edges**
  - **A directed graph is acyclic if and only if a depth-first search yields no "*back*" edges.**

Depth-first Search[10]

- **Four edge types**

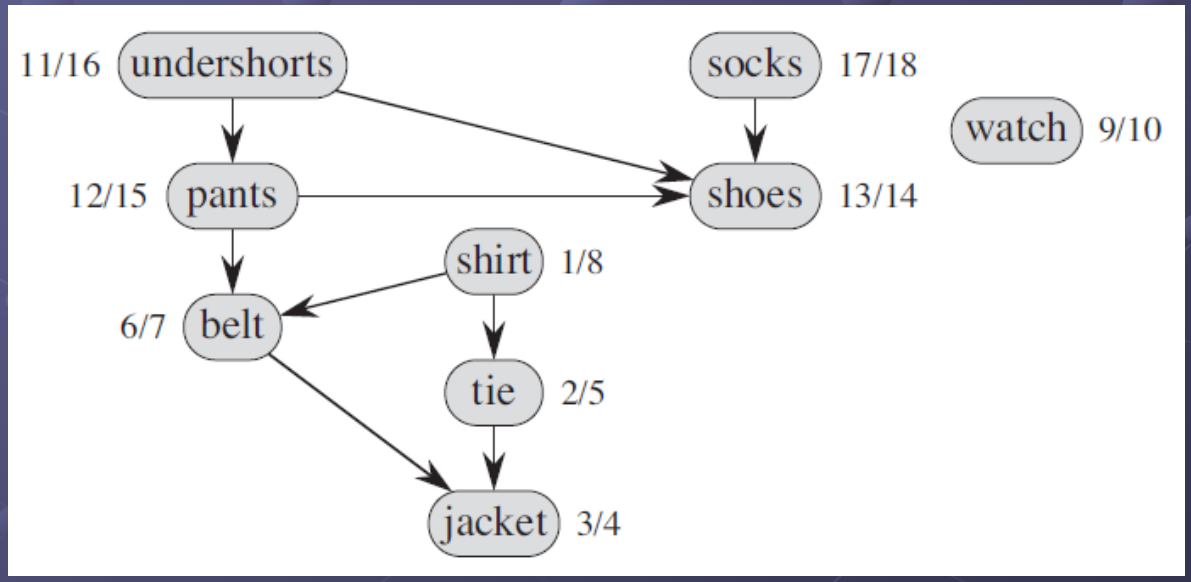
1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

Depth-first Search[11]

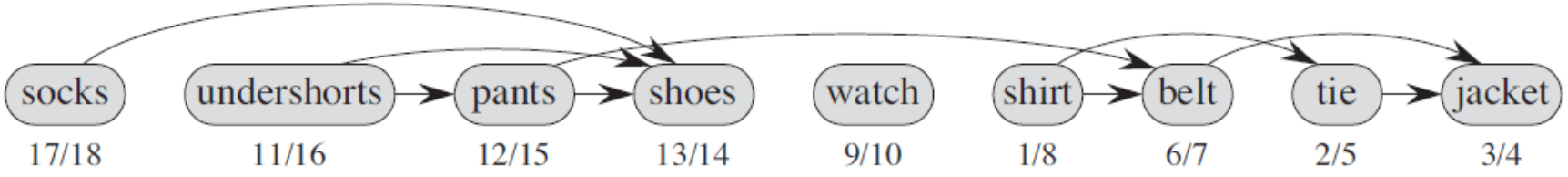
- When we first explore an edge (u, v), the color of vertex v tells us something about the edge:
  - WHITE indicates a *tree edge*,
  - GRAY indicates a *back edge*, and
  - BLACK indicates a *forward* or *cross edge*.
- *Theorem 22.10*
  - In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Topological Sort[1]

- "dag" – directed acyclic graph
- A topological sort of a dag $G = (V, E)$
  - A linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.
- Example:



Topological Sort[2]



TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

☞ **Running time O($V + E$).**

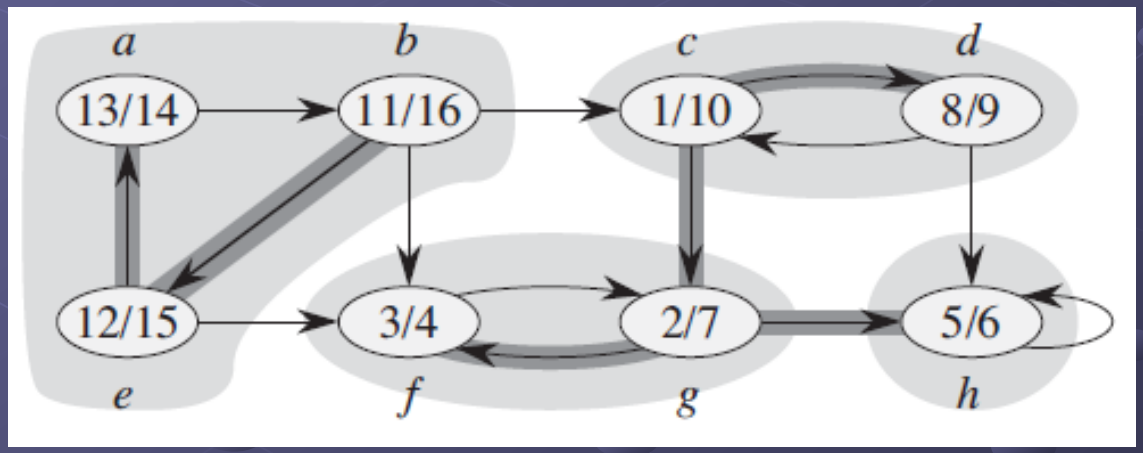
Topological Sort[3]

- ***Lemma 22.11***
  - A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.
- ***Theorem 22.12***
  - TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.

Strongly Connected Components[1]

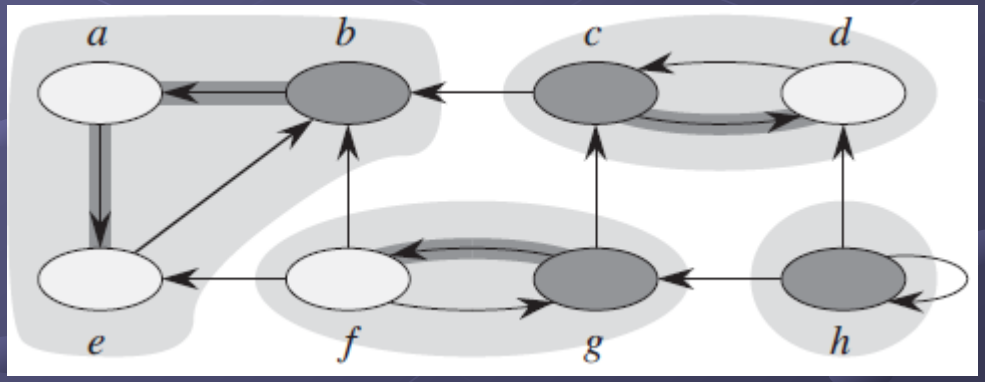
☞ ***Strongly connected component***

- For a directed graph $G = (V, E)$, a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.
- That is, vertices u and v are reachable from each other.



Strongly Connected Components[2]

- The *transpose* of G
  - $G^T = (V, E^T)$ where $E^T = \{(u, v): (v, u) \in E\}$.
  - That is, E^T consists of the edges of G with their directions reversed.
  - The time to create G^T is $O(V + E)$.
  - G and G^T have exactly the same strongly connected components.



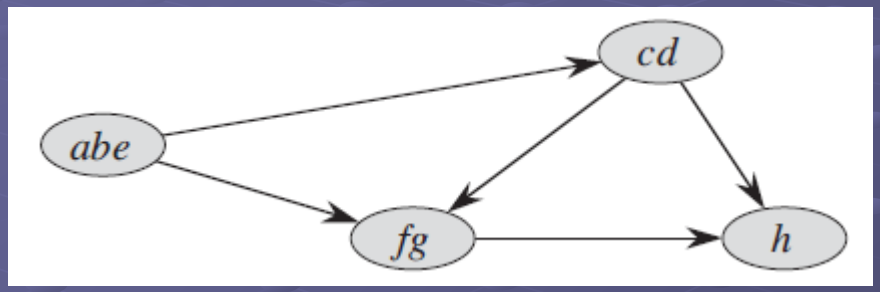
Strongly Connected Components[3]

- **A linear-time algorithm computes the strongly connected components of a directed graph.**

```
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
```

- ***Component graph*** $G^{SCC} = (V^{SCC}, E^{SCC})$.

Strongly Connected Components[4]



- *Lemma 22.13*
  - Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.
- If $U \subseteq V$, then we define

$$d(U) = \min_{u \in U} \{u.d\}$$
$$f(U) = \max_{u \in U} \{u.f\}$$

Strongly Connected Components[5]

- ***Lemma 22.14***
  - Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.
- ***Corollary 22.15***
  - Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.
    - This corollary tells us that each edge in G^T that goes between different strongly connected components goes from a component with an earlier finishing time to a component with a later finishing time.

Strongly Connected Components[6]

- **Theorem 22.16**
  - **The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the strongly connected components of the directed graph *G* provided as its input.**