Algorithms
演算法

Graphs (1)
— BFS, DFS —

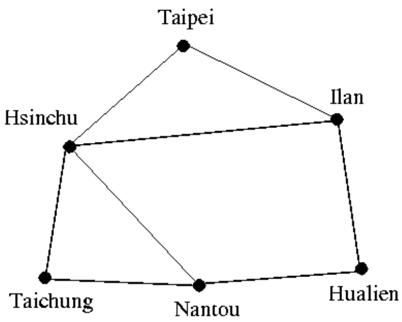
Professor Chien-Mo James Li 李建模
Graduate Institute of Electronics Engineering
National Taiwan University

Outline

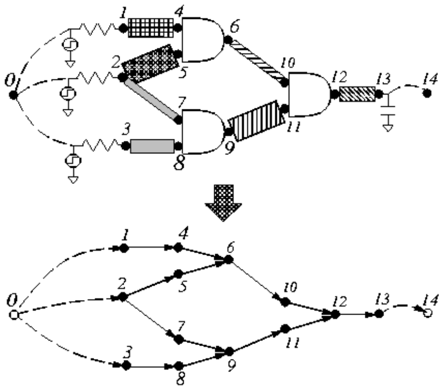
- Elementary Graph Algorithms, CH22
  - Breath First Search
    - application 1: shortest path
    - application 2: Maze router
  - Depth-first Search
    - application 1: topological sort
    - application 2: Strongly connected components
- Minimum Spanning Trees, CH23
- Single Source Shortest Paths, CH24
- All-pairs Shortest Paths, CH25
- Maximum Flow, CH26

Graph

- *Graph:* A mathematical object representing a set of “points” and “interconnections” between them
- Graph has wide applications in computer science
  - Many binary relationship can be modeled as graphs



An undirected graph

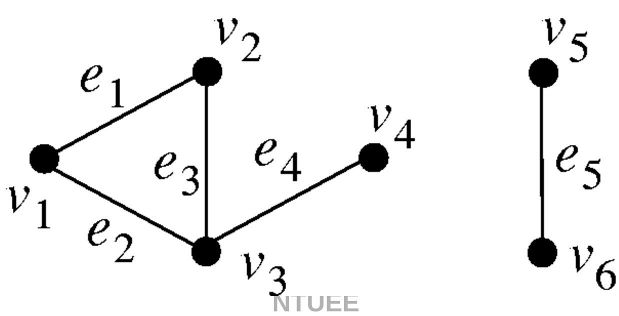


A directed circuit graph

[YW Chang NTU]

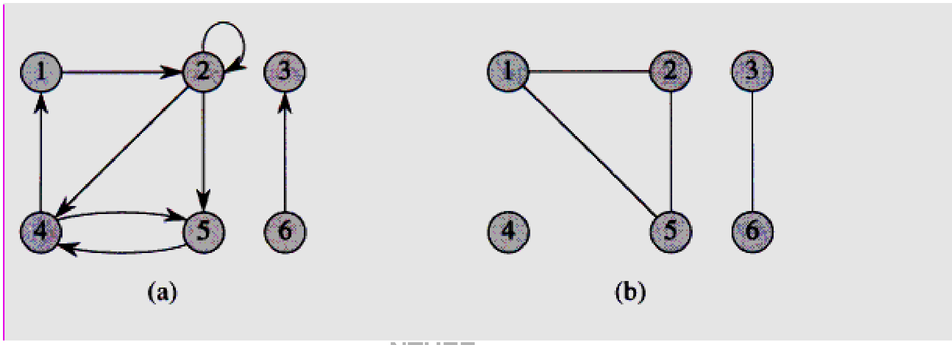
Graph (2)

- A graph $G = (V, E)$ consists of
  - V is the *vertex set*
    - $|V|$ = number of vertices in the set
    - *e.g.* $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, $|V|=6$
  - E is the *edge set*
    - An edge (u, v) connects vertices u and v
      - *directed* or *undirected*
    - $|E|$ = number of edges in the set
    - e.g. $e_1 = (v_1, v_2)$; $E = \{e_1, e_2, e_3, e_4, e_5\}$, $|E|=5$
  - For simplicity, use V for $|V|$ and E for $|E|$



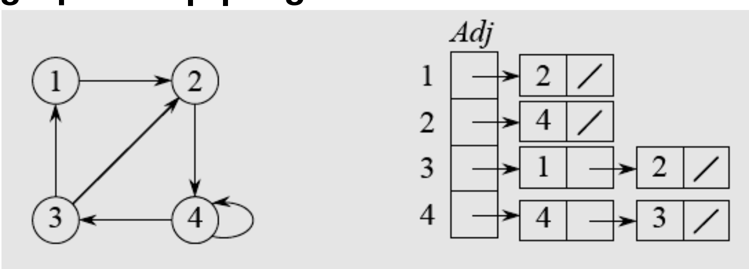
Directed and Undirected Graphs

- *Undirected Graph*: *E* consist of *unordered* pairs of vertices
  - edge (*u*,*v*) is *incident on* vertices *u* and *v*
    - *degree* of vertex *u* = number of edges incident on *u*
- *Directed Graph*: *E* consisted of *ordered* pairs of vertices
  - edge (*u*, *v*) *leaves* vertex *u* and *enters* vertex *v*
  - edge (*u*, *v*) is *incident from u* and is *incident to v*
    - *in-degree* of vertex *u* = number of edges entering *u*
    - *out-degree* of vertex *u* = number of edges leaving *u*
    - *degree* of vertex *u* = in-degree + out-degree

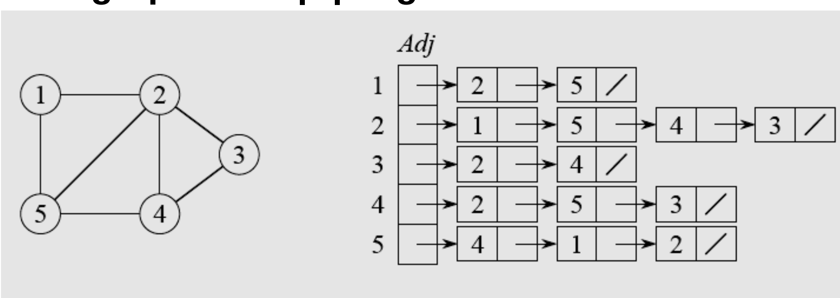


Adjacency List

- Graph *G* consists of array *G.Adj* of |*V*| lists, one per vertex.
  - Each vertex *u* has an *adjacent list: G.Adj[u]*
    - linked list of all vertices *v* such that (*u*, *v*) $\in$ *E*
- directed graph has |*E*| edges

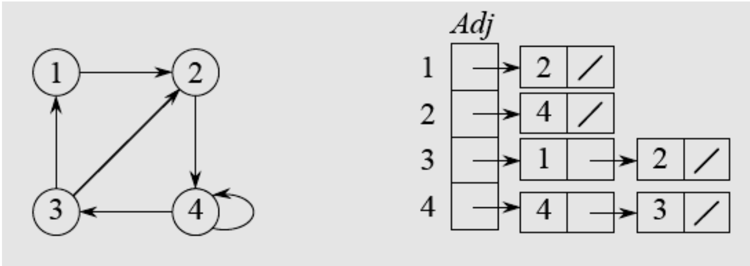


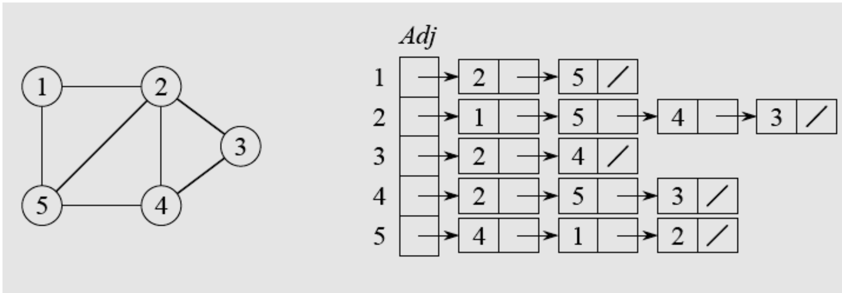
- undirected graph has 2|*E*| edges



Adjacent List (2)

- Storage $\Theta(|V|+|E|) = \Theta(V+E)$
  - good for *sparse graph*, $|E| \ll |V|^2$
- Time to determine if $(u, v) \in E$
  - O(degree(u))



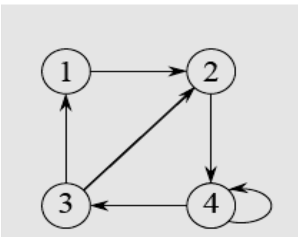


Adjacency Matrix

- Assume vertices are numbered 1, 2, ... $|V|$
- Adjacency matrix A: $|V|$ x $|V|$

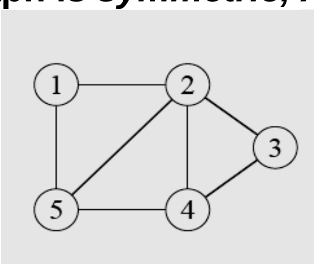
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

- directed graph



	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

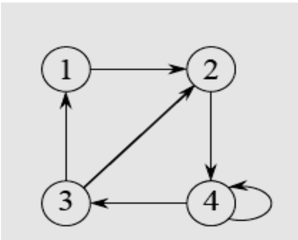
- undirected graph is *symmetric*, $A = A^T$



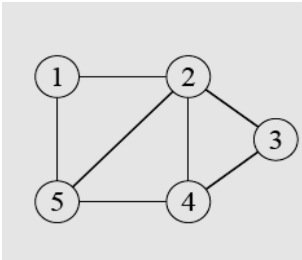
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency Matrix

- Storage required= $\Theta(V^2)$
  - good for *dense graph*, $|E| \cong |V|^2$
- Time to determine if $(u, v) \in E$
  - O(1)
- Faster but larger than adjacency list



	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1



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

Comparison

Comparison	winner
Faster to find an edge?	Matrix
Faster to find vertex out-degree*	List
Faster to traverse the graph?	List O(V+E) vs. matrix O(V²)
Storage for sparse graph?	List O(V+E) vs. matrix O(V²)
Storage for dense graph?	Matrix
Edge insertion or deletion?	Matrix O(1)
Better for most applications?	List

*Q: how about in-degree?

Acknowledgement Prof. YW Chang

- No one best way to implement a graph
  - Depends on the programming language and algorithm
  - But usually adjacency list is better for large problems

Outline

- Elementary Graph Algorithms, CH22
  - Breadth First Search
    - application 1: shortest path
    - application 2: Maze router
  - Depth-first Search
    - application 1: topological sort
    - application 2: Strongly connected components
- Minimum Spanning Trees, CH23
- Single Source Shortest Paths, CH24
- All-pairs Shortest Paths, CH25
- Maximum Flow, CH26

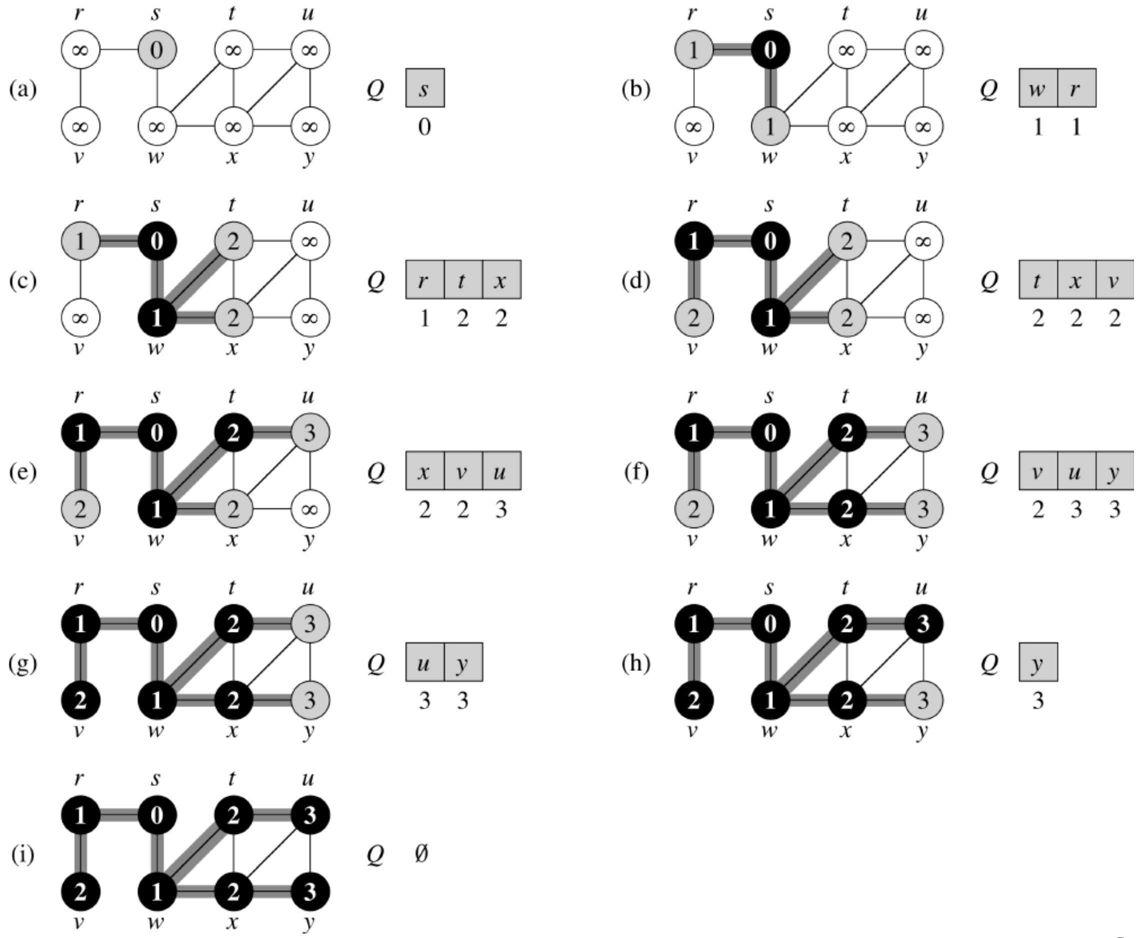
Breadth First Search (BFS)

- *BFS* is one simplest algorithm to search a graph
- Given a graph and a source vertex s,
  - explores edges to discover every vertex that is *reachable* from s

- Input: $G = (V,E)$, directed or undirected, and source vertex $s \in V$
- Output: $v.d$ = distance (smallest # of edges) from s to v for all $v \in V$
  - $v.\pi = (u, v)$ = last edge on shortest path $s \rightsquigarrow v$
    - u is v's *predecessor*
    - set of edges $\{(v.\pi, v): v \neq s\}$ forms a tree

- Idea: Send a wave out from s
  - First hits all vertices 1 edge from s
  - From there, hits all vertices 2 edges from s, *etc*
  - Use queue Q to maintain wavefront
    - $v \in Q$ if and only if wave has hit but has not come out yet

Fig 22.3



BFS

- ***u.color***
  - white (***undiscovered***)
  - gray (***discovered: out edges are being discovered***)
  - black (***explored***: out edges are all discovered)
- ***u.d***
  - distance from source to ***u***
- ***u.π***: predecessor of ***u***
- **G.adj[u] means**
  - set of vertices adjacent to ***u***
- **Store gray vertices in queue Q**
  - **First in first out**

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q,v)
    u.color = BLACK
```

Complexity Analysis

- **Use queue for gray vertices**
  - **Each vertex is enqueued and dequeued once at most**
    - ***O*(*V*) time**
  - **Each edge is considered once at most**
    - ***O*(*E*) time (adjacency list)**
- **Time complexity: O(*V*+*E*)**
- **Aggregate analysis**
  - **CH 17**

```
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
```


Exercise

- **Loop invariant:**
  - **at the test in line 10, queue Q consists of the set of gray vertices**

Outline

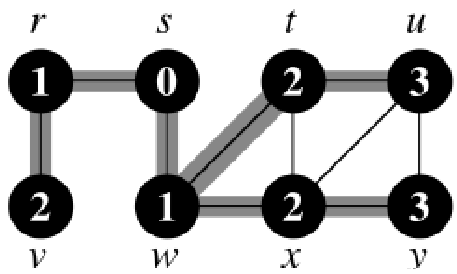
- Elementary Graph Algorithms, CH22
  - Breadth First Search
    - application 1: shortest path
    - application 2: Maze router
  - Depth-first Search
    - application 1: topological sort
    - application 2: Strongly connected components
- Minimum Spanning Trees, CH23
- Single Source Shortest Paths, CH24
- All-pairs Shortest Paths, CH25
- Maximum Flow, CH26

Shortest Path

- BFS finds shortest distance
  - from a given source vertex
  - to each *reachable* vertex
- ***Shortest-path distance*** $\delta(s,v)$:
  - minimum number of edges in any path from vertex s to vertex v
- ***Shortest path*** :
  - a path of length $\delta(s,v)$ from s to v

Breath-first Tree

- ***Predecessor subgraph*** **of** ***G*** **is** $G_\pi = (V_\pi, E_\pi)$
  - $V_\pi = \{v \in V \mid v.\pi \neq \text{NIL}\} \cup \{s\}$
    - $\{s, w, r, t, x, v, u, y\}$
  - $E_\pi = \{(v.\pi, v) \in E \mid v \in V_\pi - \{s\}\}$
    - $\{(s,w), (s,r), (w,t), (w,x), (r,v), (t,u), (x,y)\}$
- **(Lemma 22.6) When applied to a directed or undirected graph** $G=(V,E)$**, BFS constructs** π **so that the predecessor subgraph** G_π **is a** ***breath-first tree***
- ***Example :*** **Fig. 22.3 (i)**
  - **root is** s
- ***Exercise:***
  - **Draw** ***BFT***



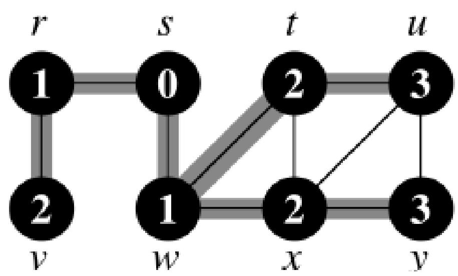
Edge in E_π **are highlighted**

PRINT-PATH

- **Prints out the vertices on a shortest path from** s **to** v

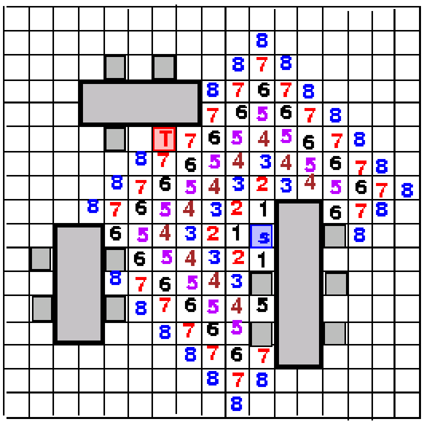
```
PRINT-PATH(G,s,v)
if v == s
      print s
elseif v.π == NIL
      print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
      print v
```

- **Exercise: print pat from** s **to** y

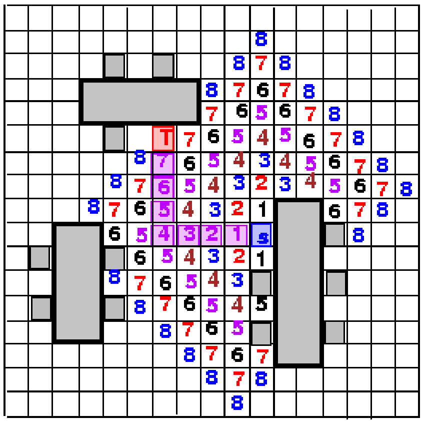


Lee's Maze Router

- Find a path from *S* to *T* by wave propagation
- Discuss mainly on single-layer routing
- Strength: Guarantee to find a minimum-length connection between 2 terminals if it exists
- Weakness: Time & space complexity for an $M \times N$ grid: $O(MN)$
  - huge!
- Note: there is more than one solution



Filing



Retrace

Outline

- Elementary Graph Algorithms, CH22
  - Breath First Search
    - application 1: shortest path
    - application 2: Maze router
  - Depth-first Search
    - application 1: topological sort
    - application 2: Strongly connected components
- Minimum Spanning Trees, CH23
- Single Source Shortest Paths, CH24
- All-pairs Shortest Paths, CH25
- Maximum Flow, CH26

Depth-first Search (DFS)

- DFS explore edges out of the more recently discovered vertex *v*
  - that still have unexplored edges leaving it
- Once all of *v*'s edges have been explored
  - *backtracks* to explore edges leaving the vertex from which *v* was discovered
- Idea: search deeper in the graph whenever possible

DFS

- *u.color*
  - white (undiscovered)
  - gray (discovered: out edges are being discovered)
  - black (explored: out edges are all discovered)
- *u.d*
  - discovery time (gray)
- *u.f*
  - finishing time (black)
- *u.π*
  - predecessor of *u*

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color ==WHITE
    DFS-VISIT(G,v)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY                    // discover u
for each v ∈ G.Adj[u]             // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                        // finish u
```

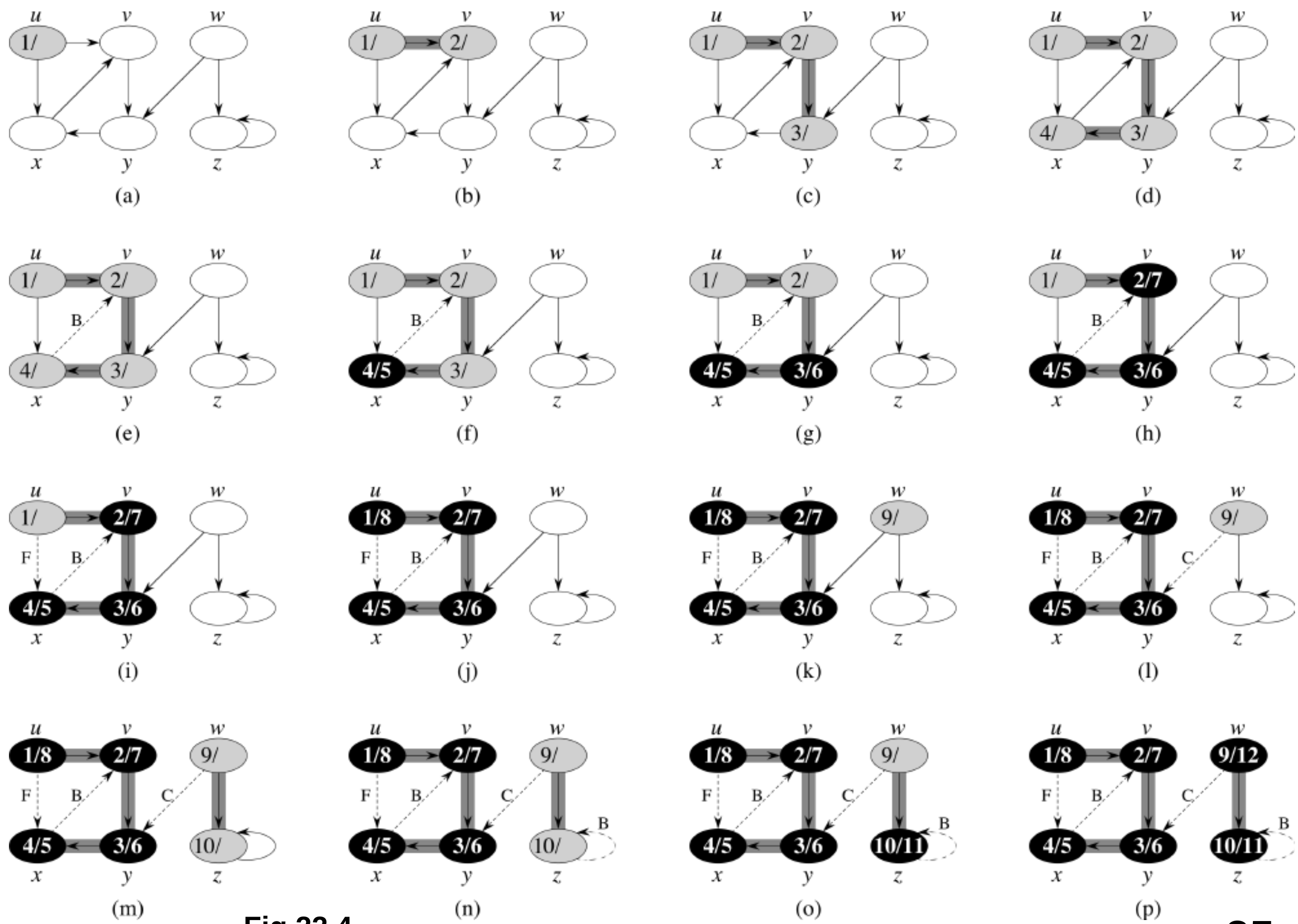


Fig 22.4

Complexity Analysis

- Time complexity
  - DFS line 1-3 $\Theta(V)$
  - DFS line 4-7 $\Theta(V)$
  - DFS-VISIT $\Theta(E)$
  - total: $\Theta(V+E)$

```
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,v)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY                      // discover u
for each v ∈ G.Adj[u]               // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                          // finish u
```

FFT

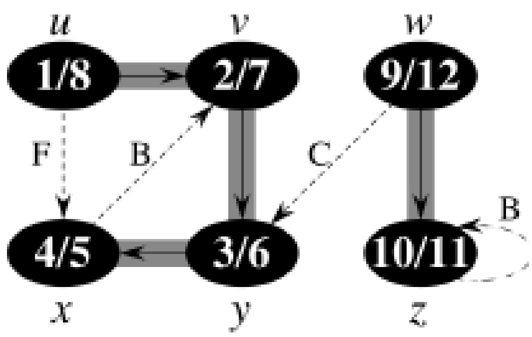
- **Why BFS is *O(V+E)* but DFS is Θ(*V+E*)**

```
DFS(G)
for each vertex u ∈ G.V
      u.color = WHITE
      u.π = NIL
time = 0
for each vertex u ∈ G.V
      if u.color ==WHITE
      DFS-VISIT(G,v)
```

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
      u.color = WHITE
      u.d = ∞
      u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ G.Adj[u]
            if v.color == WHITE
               v.color = GRAY
               v.d = u.d + 1
               v.π = u
               ENQUEUE(Q,v)
      u.color = BLACK
```


Depth-first Forest

- **Predecessor subgraph of G is $G_\pi = (V, E_\pi)$**
  - **$E_\pi = \{(v.\pi, v) \in E \mid v \in V, v.\pi \neq \text{NIL}\}$**
- **G_π forms a *depth-first forest***
  - **edges in E_π are called *tree edges***
- **Example: Fig 22.4 (p)**
  - **depth-first forest {*u*→ *v* → *y* →*x*} {*w* → *z*}**
  - **tree edges are shaded**



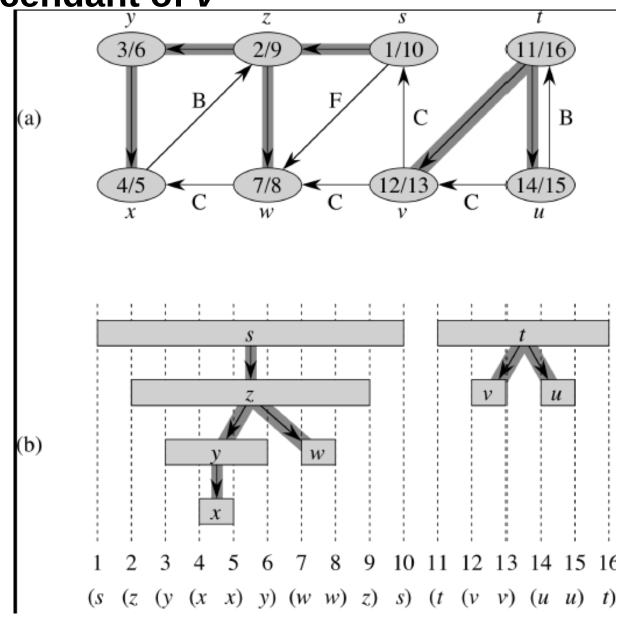
(p)

Edge in E_π are highlighted

Parenthesis Theorem *(Theorem 22.7)*

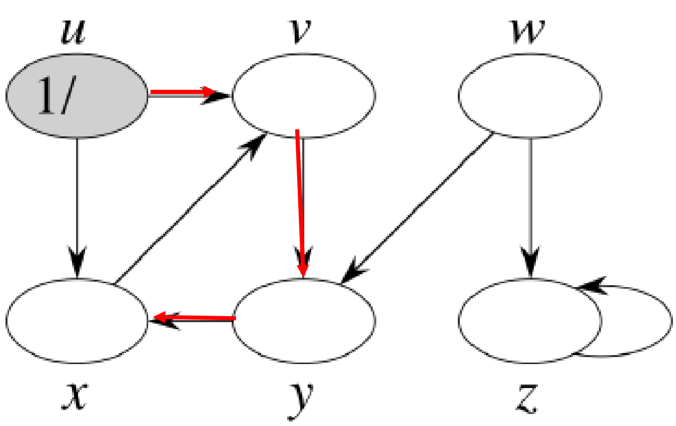
- For any two vertices *u, v*, exactly one of the following holds
- 1. *u.d* < *u.f* < *v.d* < *v.f* or *v.d* < *v.f* < *u.d* < *u.f*
  - intervals entirely disjoint; *u* or *v* is not descendant of each other
- 2. *u.d* < *v.d* < *v.f* < *u.f* and *v* is a descendant of *u*
- 3. *v.d* < *u.d* < *u.f* < *v.f* and *u* is a descendant of *v*

- *u.d* < *v.d* < *u.f* < *v.f* cannot happen
- Like parentheses:
  - OK: ( ), [ ], ([ ]), [( )]
  - No: ([ )], [( ])



White-path Theorem *(Theorem 22.9)*

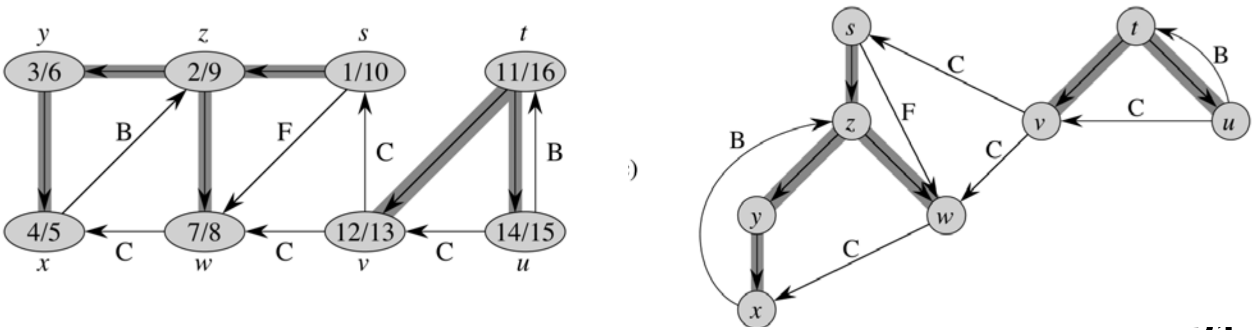
- In a DFS forest of a graph G = (*V*, *E*), vertex *v* is a descendant of vertex *u* if and only if at the time *u.d* that the search discovers *u*, there is a path *u* ~> *v* consisting entirely of white vertices
  - Except for *u*, which was just colored gray
- Fig 22.4



(a)

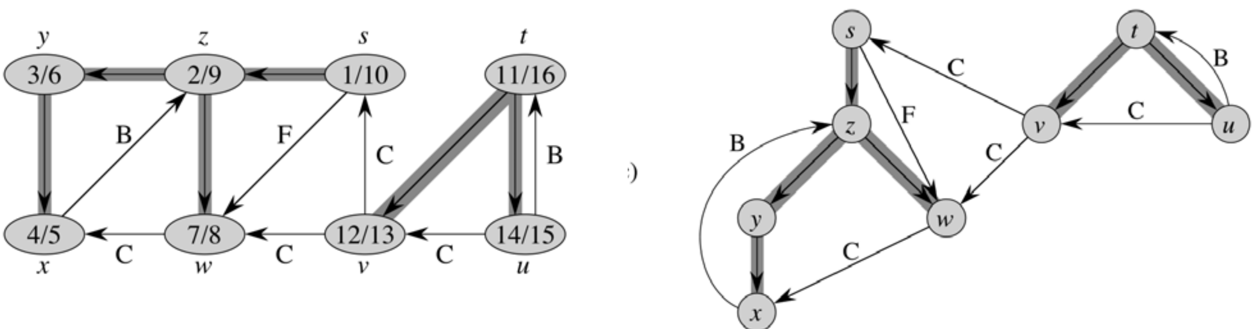
Classification of Edges

- 4 types of edges
  - ***Tree edge***: in the depth-first forest. Found by exploring (*u, v*)
  - ***Back edge***: (*u, v*), where *u* is a descendant of *v*
  - ***Forward edge***: (*u, v*), where *v* is a descendant of *u*
    - but not a tree edge
  - ***Cross edge***: any other edge
    - between vertices in same or in different depth-first trees
- Example Fig. 22.5
  - *F* and *T* are downward. *B* is upward



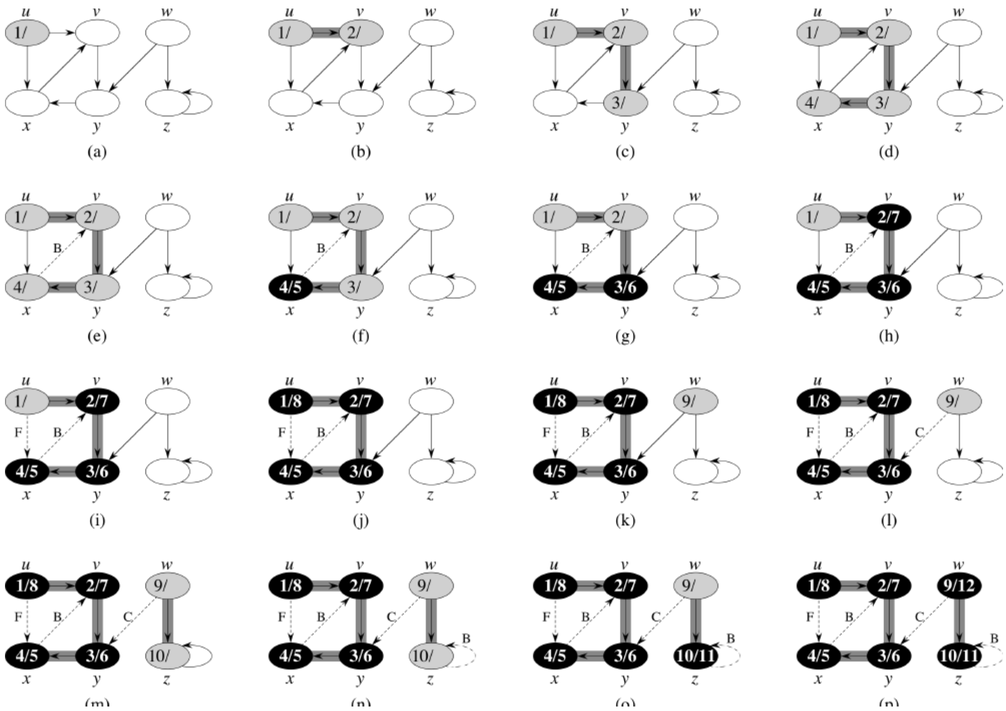
Classification of Edges (2)

- When we first explore an edge (*u,v*), the color of *v* tell us about edge
  - 1. WHITE indicates a tree edge, *e.g.* (*z, y*)
  - 2. GRAY indicates a back edge, *e.g.* (*x, z*)
  - 3. BLACK indicates
    - forward edge (*s, w*)
    - cross edge (*v, w*)



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
  - no cross edge, no forward edge
  - Why?



Summary of Edges

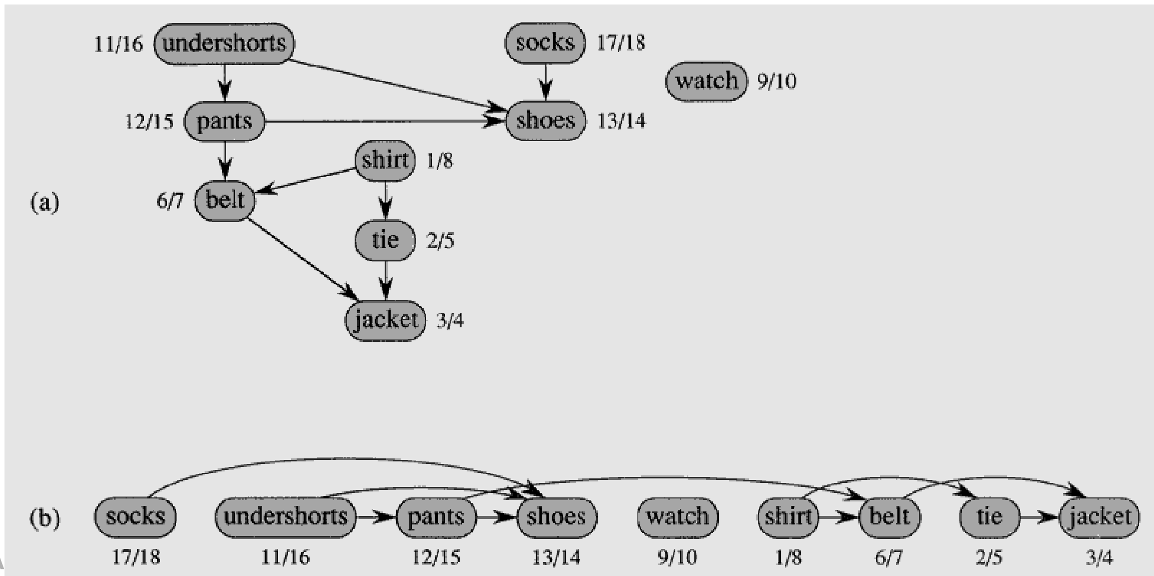
- directed graph
  - tree, forward, back, cross
- directed acyclic graph (dag)
  - tree, forward, cross
- undirected graph
  - tree, back
- Lemma 22.11

Outline

- <u>Elementary Graph Algorithms, CH22</u>
  - Breath First Search
    - application 1: shortest path
    - application 2: Maze router
  - <u>Depth-first Search</u>
    - <u>application 1: topological sort</u>
    - application 2: Strongly connected components
- Minimum Spanning Trees, CH23
- Single Source Shortest Paths, CH24
- All-pairs Shortest Paths, CH25
- Maximum Flow, CH26

Topological Sort

- *dag = directed acyclic graph*
- *Topological sort* of a *dag G* is a linear ordering of all vertices s.t.
  - if *G* contains edge (*u, v*) then *u* appears before *v* in the ordering
- Example: Fig 22.7 getting dressed in the morning
  - All edges goes from left to right



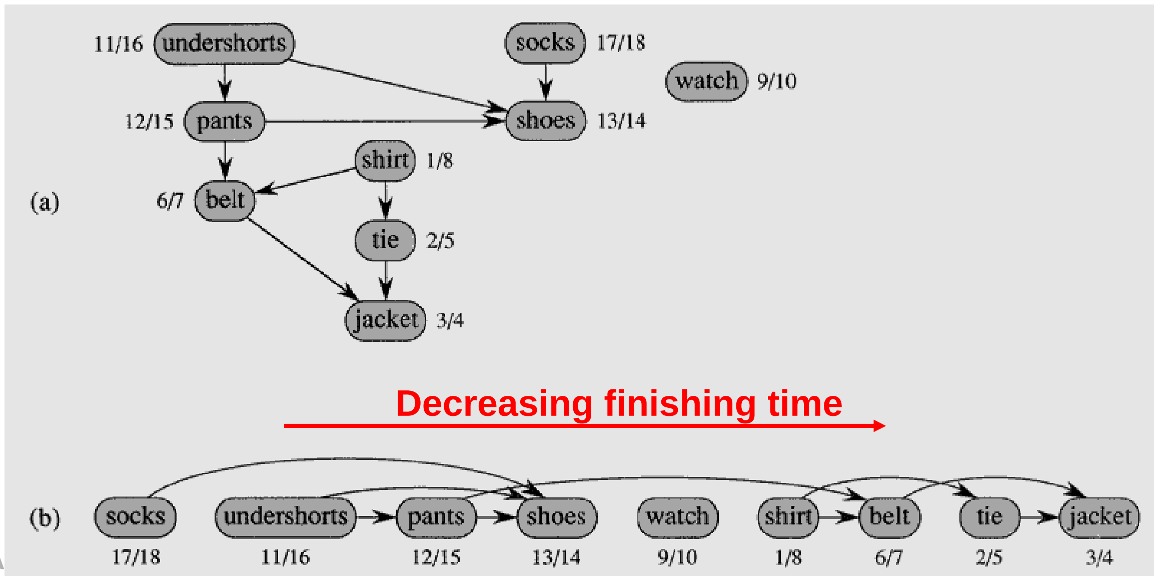
FFT

- Q: can we randomly choose the first vertex ?
- Q: if graph contain cycle, is topological sort possible?

Topological Sort (2)

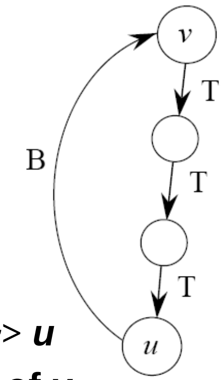
- **Time complexity = ?**

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



Lemma 22.11

- A directed graph *G* is *acyclic* if and only if a depth-first search of *G* yield *NO back edges*
  - Proof →
    - suppose DFS produce a back edge (*u,v*)
    - then *v* is ancestor of *u*. Thus G contains a path *v* ~> *u*
    - *v* ~> *u* and the back edge (*u,v*) complete a cycle
      - this is incorrect
  - Proof ←
    - suppose *G* contains a cycle *c*
    - let *v* be first vertex discovered in *c*
    - let (*u,v*) be the preceding edge in *c*
    - at time *v.d*, the vertices of c form a white path *v* ~> *u*
    - by white-path theorem, *u* becomes a descendant of *v*
    - so (*u, v*) is a back edge
      - this is incorrect



TS Algorithm Is Correct (Theorem 22.12)

- TOPOLOGICAL-SORT produces a topological sort of the dag
  - Proof: Just need to show if (*u, v*)∈ *E*, then *v.f* < *u.f*
  - When we explore (*u, v*), what are the colors of *u* and *v*?
  - *u* is gray
  - Is *v* gray, too?
    - No, because *v* would be ancestor of *u,* (*u,v*) is a back edge. This is a contradiction of previous lemma (dag has no back edges).
  - Is *v* white?
    - Then *v* becomes descendant of *u*
    - By parenthesis theorem, *u.d* < *v.d* < <u>*v.f* < *u.f*</u>
  - Is *v* black?
    - Then *v* is already finished
    - Since we're exploring (*u, v*), we have not yet finished *u*
    - Therefore, <u>*v.f* < *u.f*</u>

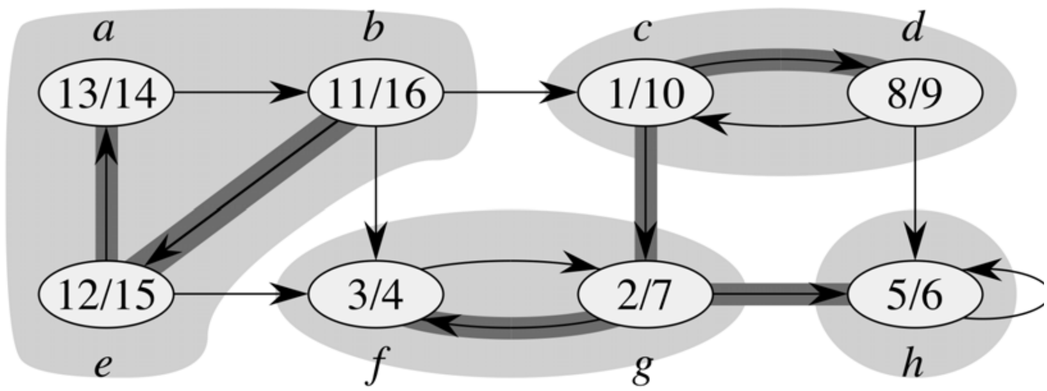
Outline

- <u>Elementary Graph Algorithms, CH22</u>
  - Breath First Search
    - application 1: shortest path
    - application 2: Maze router
  - <u>Depth-first Search</u>
    - application 1: topological sort
    - <u>application 2: Strongly connected components</u>
- Minimum Spanning Trees, CH23
- Single Source Shortest Paths, CH24
- All-pairs Shortest Paths, CH25
- Maximum Flow, CH26

Strongly Connected Components (SCC)

- Given directed graph *G =(V,E)*
  - *SCC* of *G* is a maximal set of vertices $C \subseteq V$ such that
  - For all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$
    - *i.e.* *u* and *v* are *reachable* from each other
- Example : Fig 22.9
  - each shaded region is an *SCC* of *G*

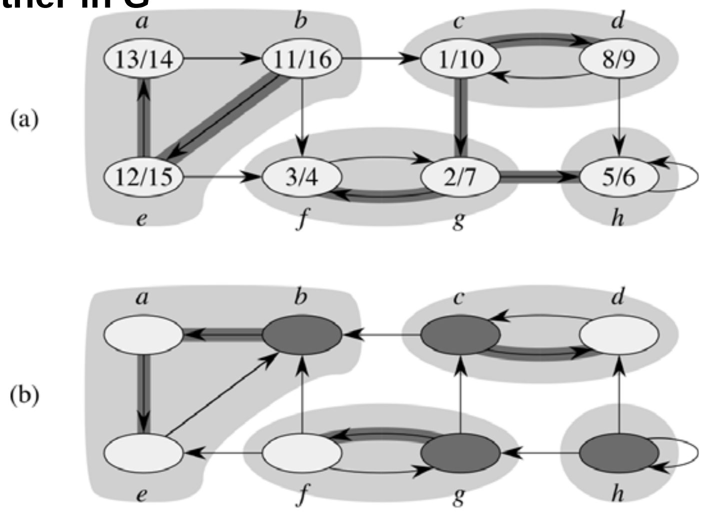
(a)



discovery time / finishing time

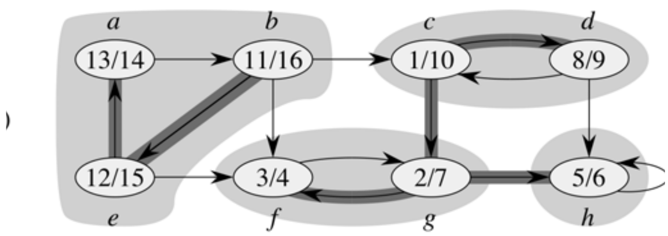
Transpose Graph

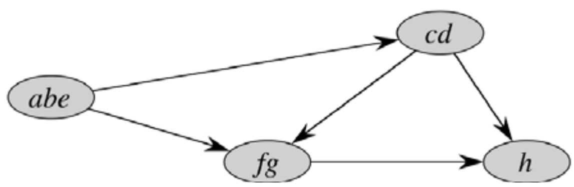
- G^T = *transpose* of G
  - $G^T = (V, E^T)$, $E^T = \{(u,v): (v,u) \in E\}$
  - G^T is G with all edges reversed
  - Can create G^T in $\Theta(V + E)$ time if using adjacency lists
- Obviously, G and G^T have *the same* SCC's
  - u and v are reachable from each other in G if and only if reachable from each other in G^T
- Example:
  - Fig 22.9



Component Graph

- $G^{SCC} = (V^{SCC}, E^{SCC})$
  - V^{SCC} has one vertex for each SCC in G
  - E^{SCC} has an edge if there's an edge between corresponding SCC
- (Lemma 22.13) G^{SCC} is a dag
  - Proof: let C and C' be distinct SCC
  - let $u, v \in C$ let $u', v' \in C'$
  - If there is a path $u \rightsquigarrow u'$, then there is no path $v' \rightsquigarrow v$
    - why? If there is a path $v \rightsquigarrow v'$ in G
    - then there are paths $u \rightsquigarrow u' \rightsquigarrow v' \rightsquigarrow v$
    - u and v' are reachable from each other → contradition!

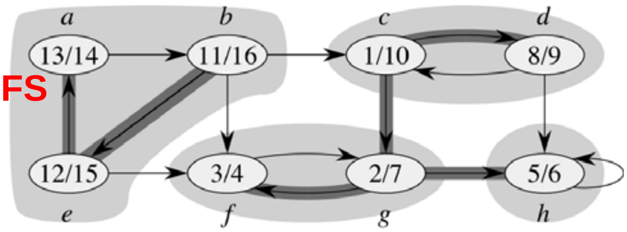


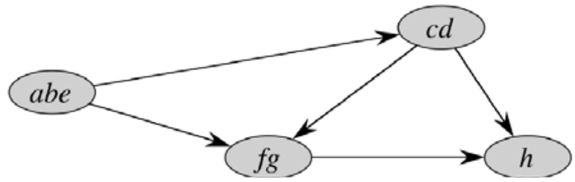


SCC Algorithm

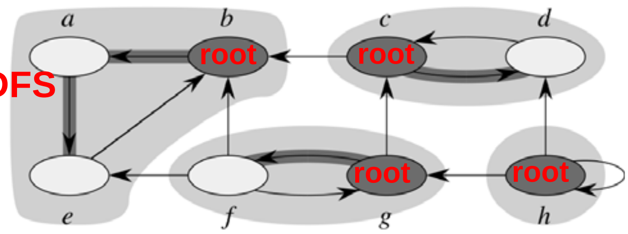
STRONGLY-CONNECTED-COMPONENTS(G)
1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed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

1st DFS
(a)





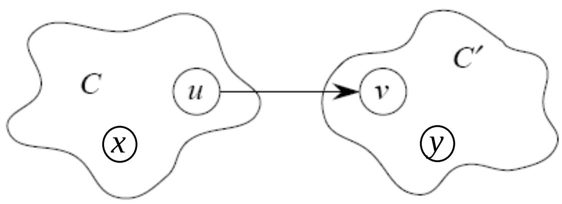
2nd DFS
(b)



consider vertices in decreasing order of finishing time: *b, c, g, h*

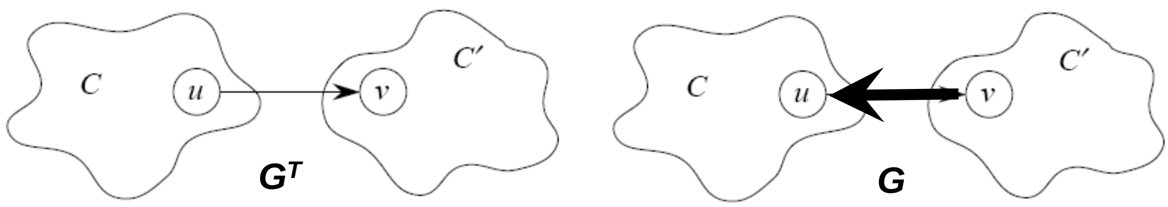
Lemma 22.14

- **For a set of vertices $U \subseteq V$**
  - **$d(U)$ =$\min_{u \in U}$ {$u.d$} (earliest discovery time of all vertices in U)**
  - **$f(U)$= $\max_{u \in U}$ {$u.f$} (latest finishing time of all vertices in U)**
- **(Lemma 22.14) Let C and C' be distinct SCC in G =(V,E). Suppose there is an edge $(u,v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$**
  - **If $d(C) < d(C')$, let x be the first vertex discovered in C**
    - **At time $x.d$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'**
    - **(white-path theorem) all vertices in C and C' are descendants of x**
    - **(parenthesis theorem) $x.f = f(C) > f(C')$**
  - **If $d(C) > d(C')$, let y be the first vertex discovered in C'**
    - **At time $y.d$, all vertices in C' are white and there is a white path from y to each vertex in C'. Again, $y.f = f(C')$**
    - **At time $y.d$, all vertices in C are white.**
    - **(Lemma 22.13) since there is an edge (u,v), we cannot have a path from C' to C. So no vertex in C is reachable from y**
    - **At time $y.f$. all vertices in C are still white. which means $f(C) > f(C')$**



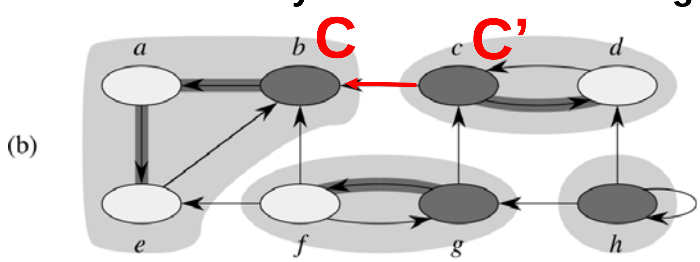
Corollary 22.15

- Let C and C' be distinct SCC in $G = (V,E)$. Suppose there is an edge $(u, v) \in E^T$ such that $u \in C$ and $v \in C'$. Then $f(C) < f(C')$
  - $(u, v) \in E^T$
  - $(v, u) \in E$
  - Since SCC's of G and G^T are the same, $f(C) < f(C')$
- Let C and C' be distinct SCC in G, and $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^T
  - Proof: It's the contrapositive of the previous corollary.



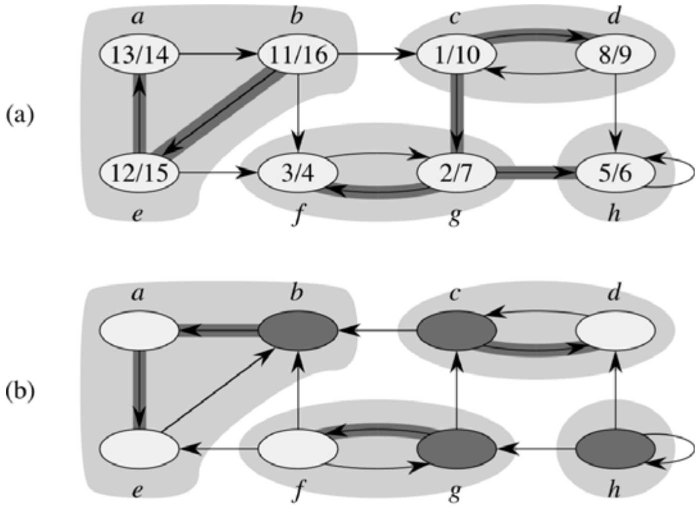
SCC Algorithm Is Correct (Theorem 22.16)

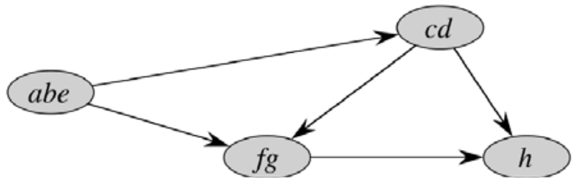
- Second DFS on G^T start with SCC in C such that $f(C)$ is maximum
- Starts from some $x \in C$, and visits all vertices in C
  - $f(C) > f(C')$ for all $C' \neq C$, so no edges from C to C' in G^T (corollary)
  - 2nd DFS will visit only vertices in C
- Next root chosen is in C'
  - such that $f(C')$ is maximum over all SCC other than C
  - 2nd DFS visits all vertices in C',
    - The only edges out of C' go to C
- Repeat ... Each time the second DFS reaches only
  - vertices in its SCC — tree edges
  - vertices in SCC already visited — no tree edges



Another View

- 2^{nd} DFS visits vertices of $(G^T)^{SCC}$ in reverse topologically order
  - because $((G^T)^{SCC})^T = G^{SCC}$ (exercise 22.5-4)
  - 2^{nd} DFS visit the vertices of G^{SCC} in topological order





Topological sort
abe→cd→fg→h

Reading

- CH 22