طراحی الگوریتم

۲۸ و ۳۰ مهر ۹۸

ملکی مجد

Topic	Reference
Recursion and Backtracking	Ch.1 and Ch.2 JeffE
Dynamic Programming	Ch.3 JeffE and Ch.15 CLRS
Greedy Algorithms	Ch.4 JeffE and Ch.16 CLRS
Amortized Analysis	Ch.17 CLRS
Elementary Graph algorithms	Ch.6 JeffE and Ch.22 CLRS
Minimum Spanning Trees	Ch.7 JeffE and Ch.23 CLRS
Single-Source Shortest Paths	Ch.8 JeffE and Ch.24 CLRS
All-Pairs Shortest Paths	Ch.9 JeffE and Ch.25 CLRS
Maximum Flow	Ch.10 JeffE and Ch.26 CLRS
String Matching	Ch.32 CLRS
NP-Completeness	Ch.12 JeffE and Ch.34 CLRS

- Talk about graph
- DFS
- BFS

Running time of a graph algorithm

- the size of the input of the graph
  - the number of vertices $|V|$
  - and the number of edges $|E|$

- Inside asymptotic notation (such as O-notation or Θ-notation)
  - the symbol V denotes $|V|$ and the symbol E denotes $|E|$

- the pseudocode views vertex and edge sets as attributes of a graph
  - denote the vertex set of a graph G by $V[G]$ and its edge set by $E[G]$

Two representations of graphs

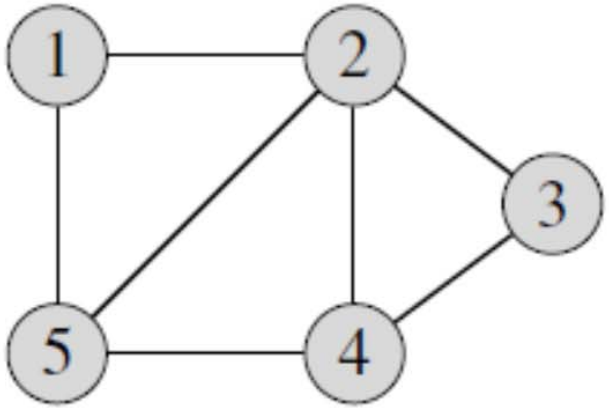
- Either way is applicable to both directed and undirected graphs

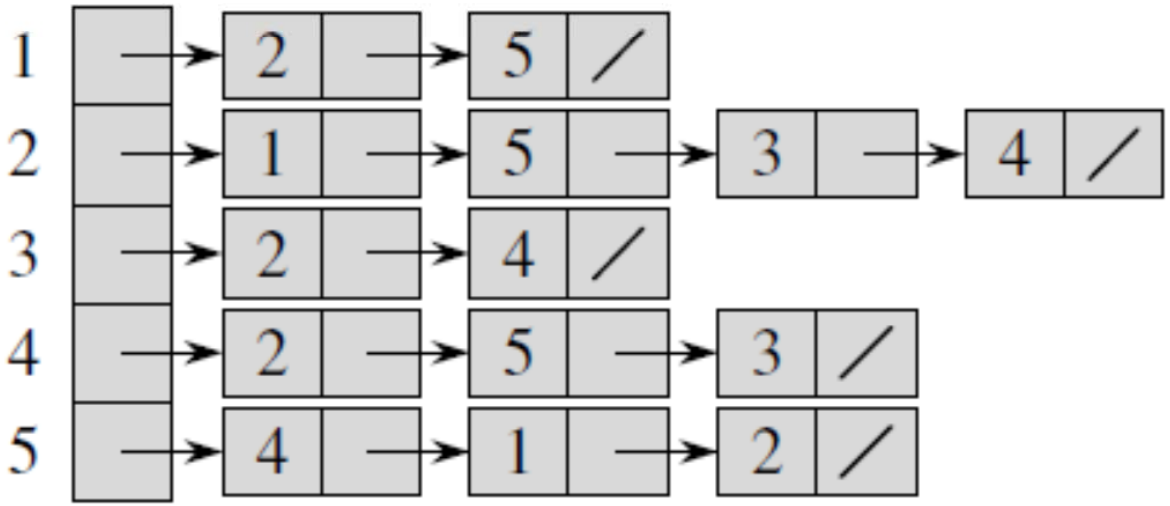
- as **adjacency lists**
  - provides a compact way to represent ***sparse*** graph($|E|$ is much less than $|V|^2$)
  - Assumed by most of the presented graph algorithms in this course

- as **adjacency matrices**
  - the graph is ***dense***($|E|$ is close to $|V|^2$)
  - or when we need to be able to tell quickly if there is an edge connecting two given vertices
  - Rather than using one word of computer memory for each matrix entry, the adjacency matrix uses only one bit per entry.

Example of undirected graph

- Two representations of an undirected graph in next slide.
  - An undirected graph *G* having five vertices and seven edges.
  - An adjacency-list representation of *G*.
    - consists of an array *Adj* of $|V|$ lists, one for each vertex in *V*.
    - For each $u \in V$, the adjacency list *Adj*[*u*] contains all the vertices *v* such that there is an edge $(u, v) \in E$.
    - The vertices in each adjacency list are typically stored in an arbitrary order
  - The adjacency-matrix representation of *G*.



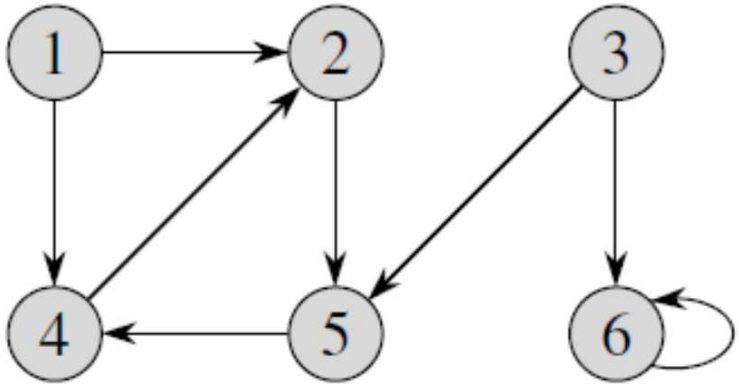


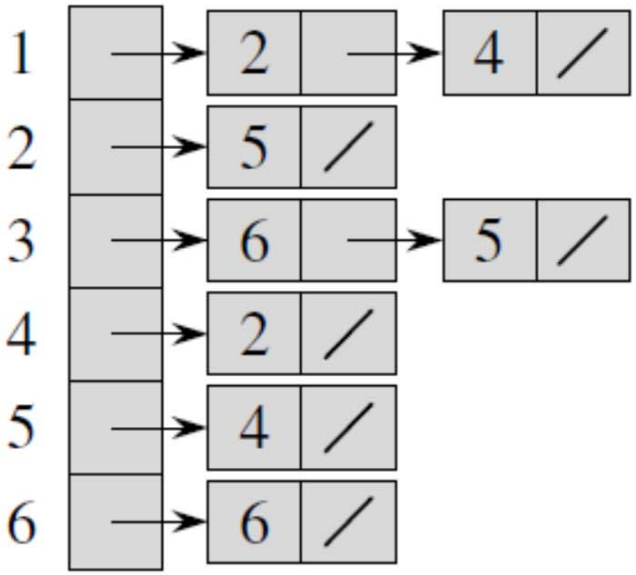
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Example of directed graph

- Two representations of a directed graph in next slide.
  - A directed graph G having six vertices and eight edges.
  - An adjacency-list representation of G.
  - The adjacency-matrix representation of G.

- For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$.





	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

- the sum of the lengths of all the adjacency lists
  - directed graph : $|E|$
  - undirected graph : $2|E|$
- the amount of memory adjacency-list representation requires
  - $\Theta(V + E)$
- the amount of memory *adjacency-matrix* representation requires
  - $\Theta(V^2)$
  - independent of the number of edges in the graph

Transpose of a matrix *A*

- the *transpose* of a matrix $A = (a_{ij})$ is the matrix $A^T = (a_{ij}^T)$ given by $a_{ij}^T = a_{ji}$
- the adjacency matrix *A* of an undirected graph is its own transpose

Weighted graphs

- let $G = (V, E)$ be a weighted graph with weight function $w : E \to R$.
- The weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored with vertex v in u's adjacency list
- the weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored as the entry in row u and column v of the adjacency matrix

- درجه خروجی راس های یک گراف جهت دار را در چه زمانی می توانیم حساب کنیم؟
- درجه ورودی راس های یک گراف جهت دار را در چه زمانی می توانیم حساب کنیم؟

- زمان مورد نیاز برای پیدا کردن گره universal sink چقدر است (گراف با ماتریس مجاورت نمایش داده شده)؟
  - (گره ای با درجه خروجی $|V| - 1$ و درجه ورودی صفر)
  - در زمان $O(V)$ می شود؟

Breadth-first search

- one of the simplest algorithms for searching a graph
- archetype for many important graph algorithms
  - Prim
  - Dijkstra,

- **breadth-first search** systematically explores the edges of G to "discover" every vertex that is reachable from s
- It computes the distance (smallest number of edges) from s to each reachable vertex
- produces a "breadth-first tree" with root s that contains all reachable vertices
- for any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G

Breadth

- the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$

- To keep track of progress, breadth-first search colors each vertex **white, gray, or black**.
- All vertices start out **white** and may later become **gray** and then **black**.

- A vertex is ***discovered*** the first time it is encountered during the search
  - at which time it becomes nonwhite
  - all vertices adjacent to black vertices have been discovered
  - Gray vertices may have some adjacent white vertices

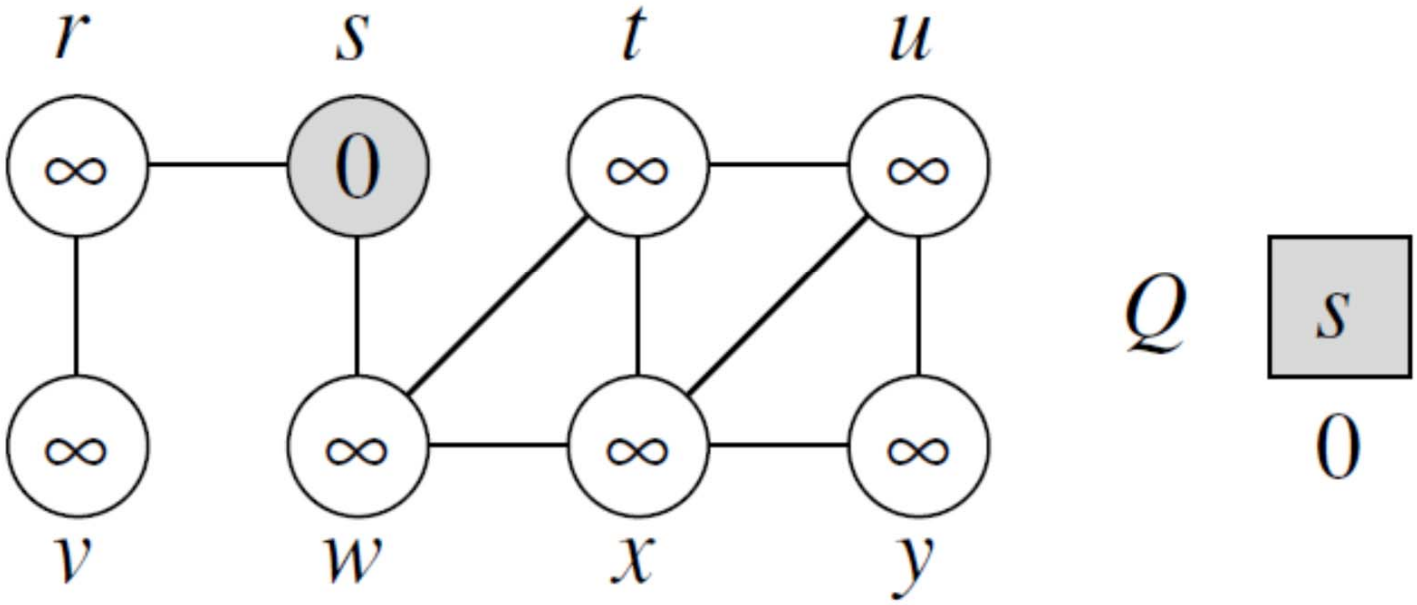
Breadth-first search

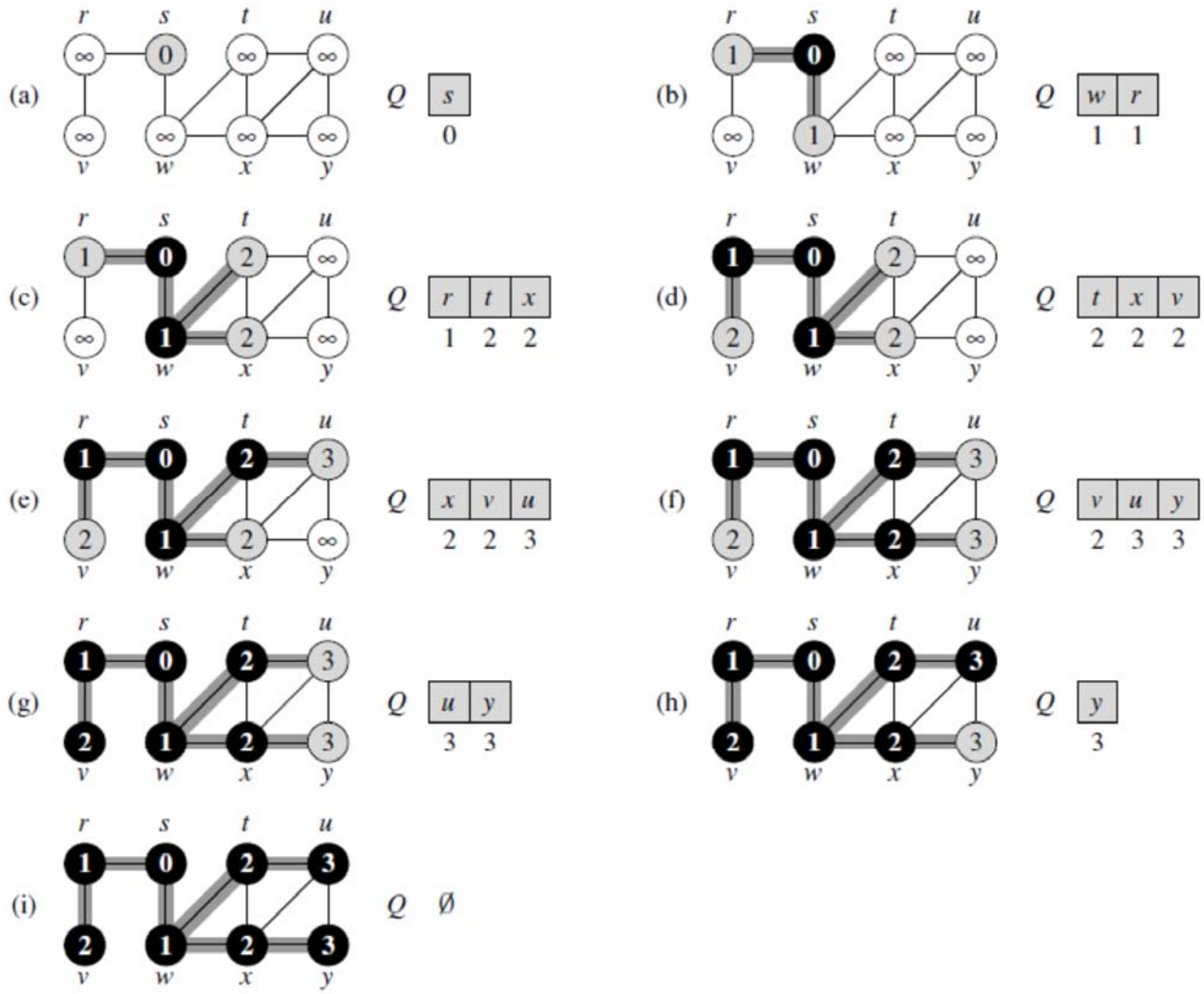
- constructs a breadth-first tree,
  - initially containing only its root, which is the source vertex *s*.
- Whenever a white vertex *v* is discovered in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree.
- u is the ***predecessor*** or ***parent*** of v in the breadth-first tree.
  - Since a vertex is discovered at most once, it has at most one parent.
  - Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s

- The breadth-first-search procedure BFS uses **adjacency lists**
- The **color** of each vertex $u \in V$ is stored in the variable $color[u]$
- The **predecessor** of u is stored in the variable $\pi[u]$.
  - If u has no predecessor (root and not discovered), then $\pi[u] = NIL$
- The **distance** from the source s to vertex u computed by the algorithm is stored in $d[u]$.
- The algorithm also uses a **first-in, first-out queue** Q to manage the set of gray vertices

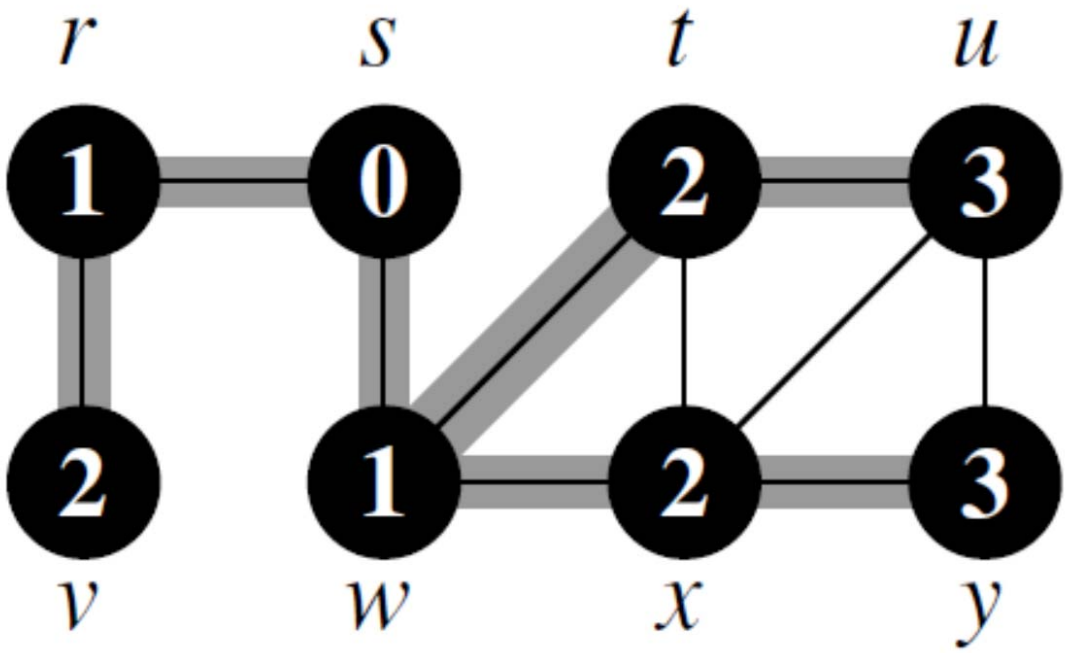
```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v]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

Compute BFS tree?




(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q ∅

Result:



- The results of **breadth-first search** may depend upon the **order** in which the neighbors of a given vertex are visited in line 12
- the breadth-first tree may vary, but the **distances** d computed by the algorithm **will not vary**

Analyzing the running time of BFS

- The overhead for initialization is $\boldsymbol{O(V)}$
- Each vertex is enqueued at most once
  - and hence dequeued at most once
- The operations of enqueuing and dequeuing take $O(1)$ time
  - the total time devoted to queue operations is ***O(V)***.
- Each adjacency list is scanned at most once
  - sum of the lengths of all the adjacency lists is $\boldsymbol{\Theta(E)}$,
- The total running time of BFS is ***O(V + E)***
  - runs in time linear in the size of the adjacency-list representation

Shortest paths

- we claimed that breadth-first search finds the distance to each reachable vertex in a graph $G = (V, E)$ from a given source vertex $s \in V$.
- Define the ***shortest-path distance*** $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v
  - if there is no path from s to v, then $\delta(s, v) = \infty$.
- A path of length $\delta(s, v)$ from s to v is said to be a ***shortest path*** from s to v.

Lemma 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: u is reachable from s?

u is not reachable from s?

- We want to show that BFS properly computes $d[v] = \delta(s, v)$ for each vertex $v \in V$. We first show that $d[v]$ bounds $\delta(s, v)$ from above

Lemma 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 $d[v]$ computed by BFS satisfies $d[v] \geq \delta(s, v)$.

Proof: induction on the number of ENQUEUE operations

- basis *: $d[v] \geq \delta(s, v)$* for all $v \in V$ *($d[s] = 0 = \delta(s, s)$* and $d[v] = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$.)
- inductive step :

Lemma 2

Proof: induction on the number of ENQUEUE operations

- inductive step :

consider a white vertex v that is discovered during the search from a vertex u

inductive hypothesis implies that $d[u] \geq \delta(s,u)$

from lemma 1 and line 15:

$$d[v] = d[u] + 1 \geq \delta(s,u) + 1 \geq \delta(s,v)$$

$d[v]$ never changes again (because never enqueued again)

- To prove that $d[v] = \delta(s, v)$, we must first show more precisely how the queue Q operates during the course of BFS.
- The next lemma shows that at all times, there are **at most two distinct d** values in the queue.

Lemma 3

Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $v_1, v_2, \ldots, v_r$, where v_1 is the head of Q and v_r is the tail.

Then,

$d[v_r] \leq d[v_1] + 1$ and

$d[v_i] \leq d[v_{i+1}]$ for $i = 1, 2, \ldots, r - 1$.

Proof: induction on the number of queue operations

Lemma 3

Proof: induction on the number of queue operations

- basis *:* Initially, when the queue contains only s, the lemma certainly holds.
- inductive step : we must prove that the lemma holds after both dequeuing and enqueuing a vertex.

Corollary 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 $d[v_i] \leq d[v_j]$ at the time that v_j is enqueued.

- ***Proof:***
  - Lemma 3
  - Each vertex receives a finite d value at most once during the algorithm

Theorem 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, $d[v] = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \in s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by the edge $(\pi[v], v)$.
- Proof: contradiction
  - A vertex receives a *d* value not equal to its shortest path distance!

Breadth-first trees

- $V_\pi = \{v \in V : \pi[v] = NIL\} \cup \{s\}$
- $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$

- The predecessor subgraph G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G.

Lemma 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.

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

DFS

depth-first search

- to search **deeper** in the graph whenever possible

- Two applications of depth-first search
  - Sorting a directed acyclic graph (DAG)
  - Finding the strongly connected components of a directed graph

Insight of DFS

In depth-first search,

- edges are explored out of the **most recently** discovered vertex v that still has unexplored edges leaving it.
- When **all** of v's edges have been explored, the search **backtracks** to explore edges leaving the vertex from which v was discovered (**predecessor**)
- This process **continues** until we have discovered all the vertices that are reachable from the original source vertex.
- If **any undiscovered** vertices remain, then one of them is selected as a **new source** and the search is repeated from that source.

From one vertex

```
DFS-VISIT(u)
1  color[u] ← GRAY   ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]        ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK           ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

coloring technique

- Each vertex is **initially white**, is **grayed** when it is ***discovered*** in the search, and is **blackened** when it is ***finished***

- This technique guarantees that each vertex ends up in exactly one depth-first tree

timestamps

- Each vertex *v* has two timestamps:
- the **first timestamp *d*[*v*]** records when *v* is first discovered (and grayed),
- and the second **timestamp *f* [*v*]** records when the search finishes examining *v*'s adjacency list (and blackens *v*)

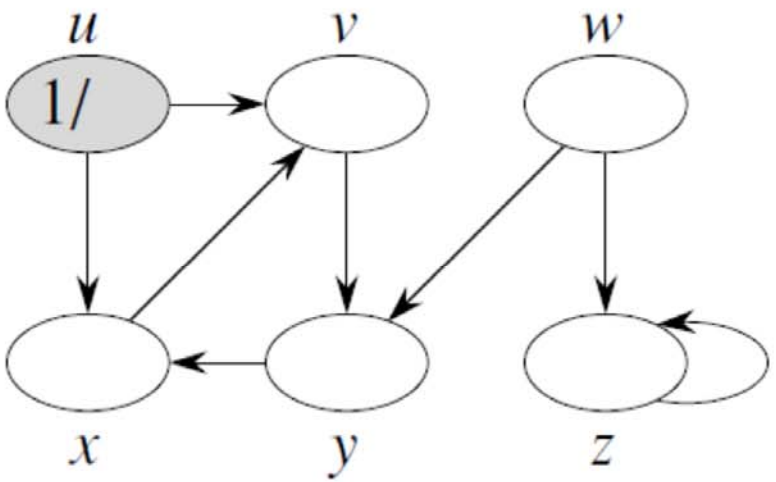
- These timestamps are integers between 1 and 2 |*V*|

- the predecessor subgraph produced by a depth-first search may be composed of **several trees**
  - search may be repeated from multiple sources

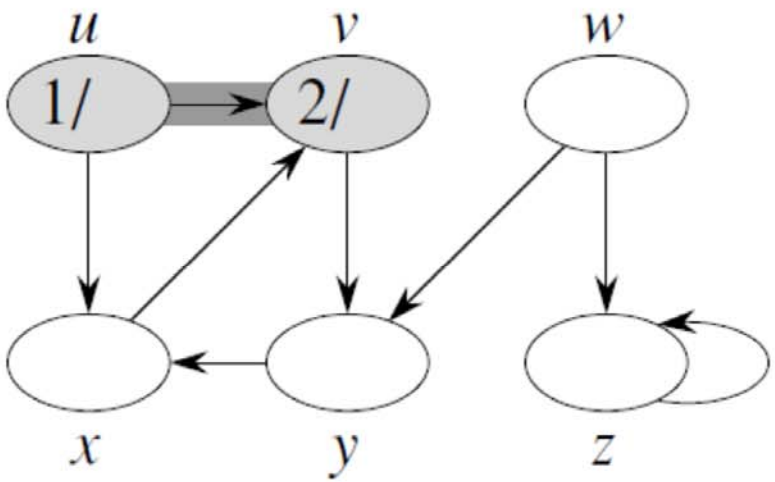
Whole graph

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7          then DFS-VISIT(u)
```

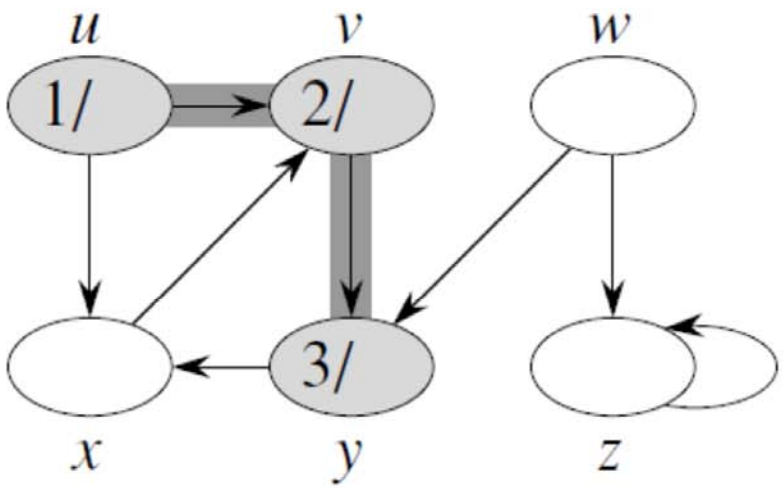
Example



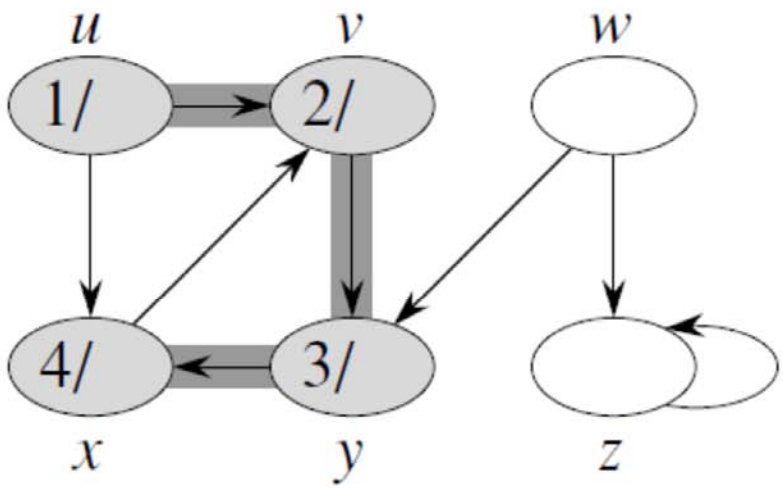
(a)



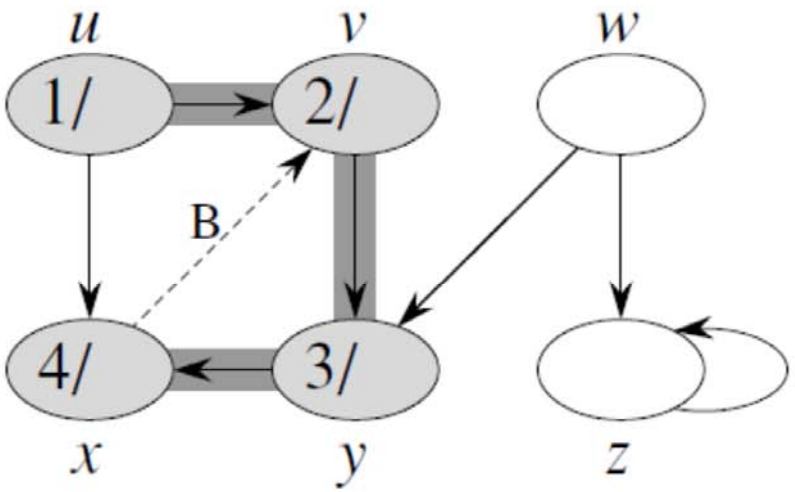
(b)



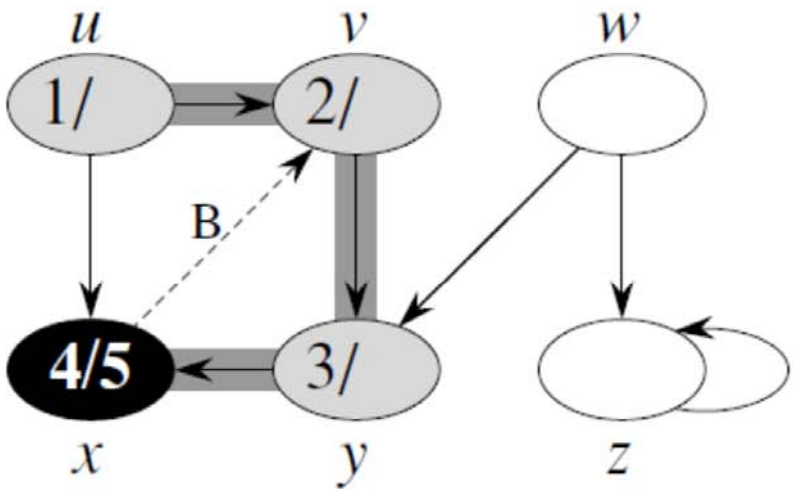
(c)



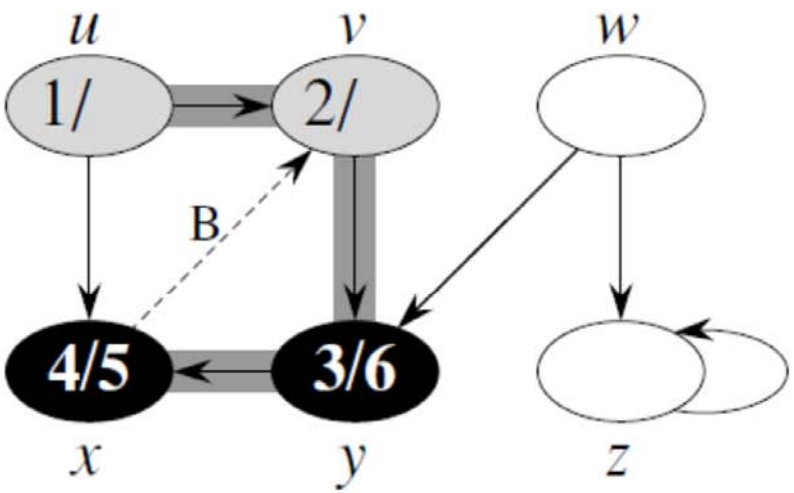
(d)



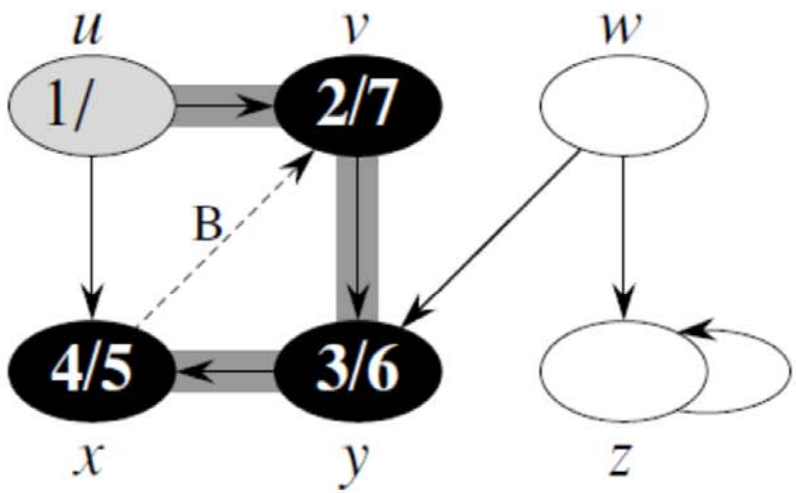
(e)



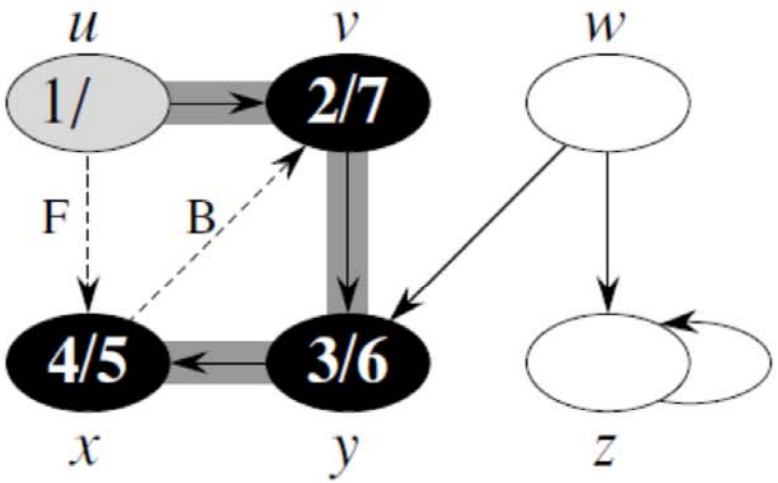
(f)



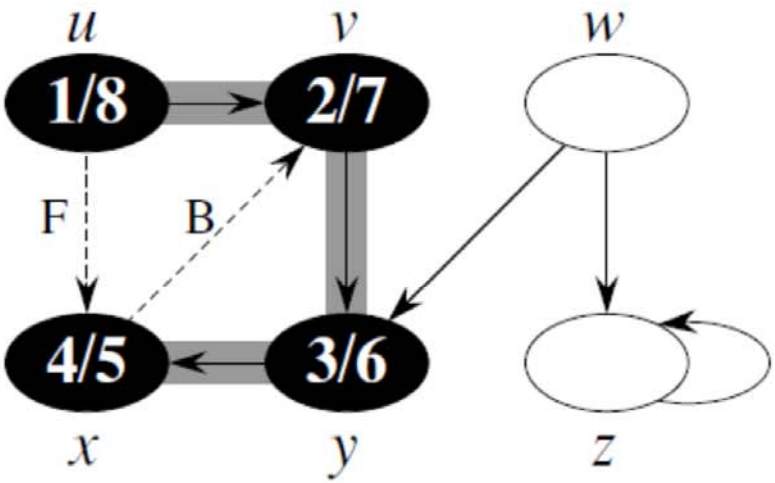
(g)



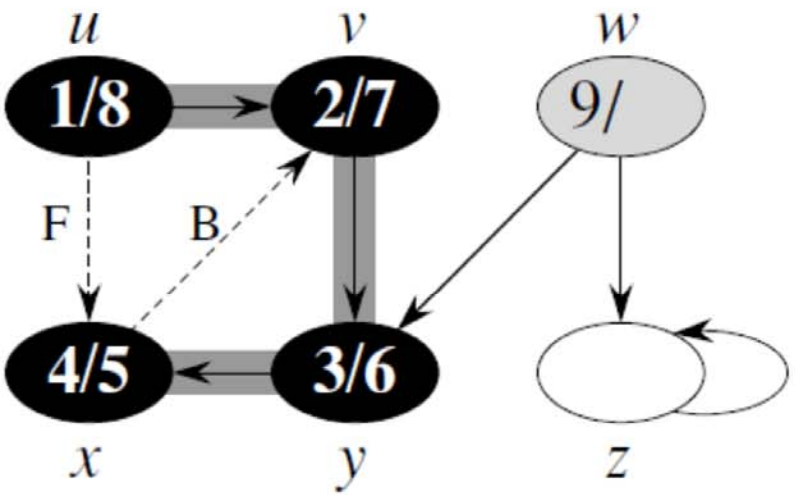
(h)



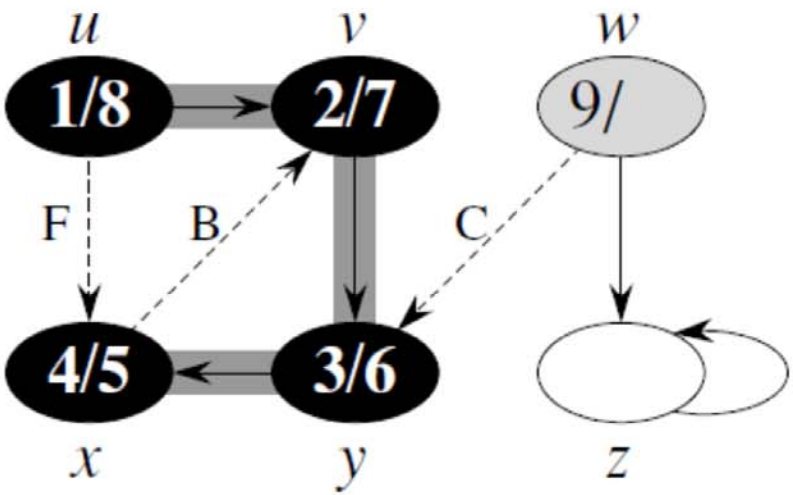
(i)



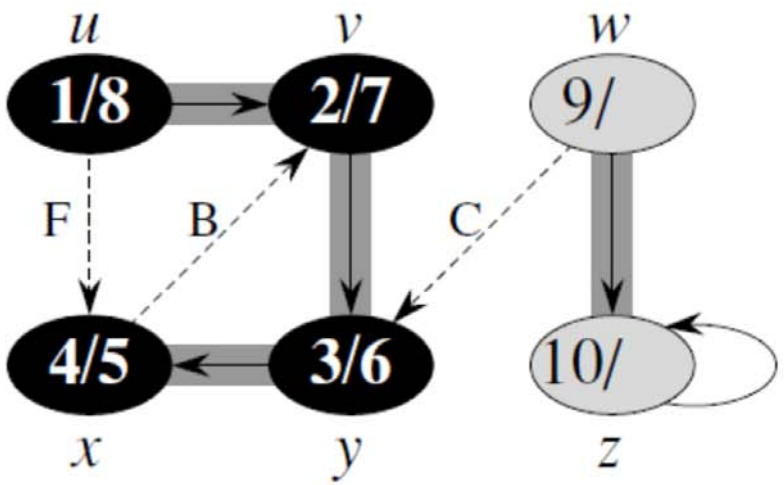
(j)



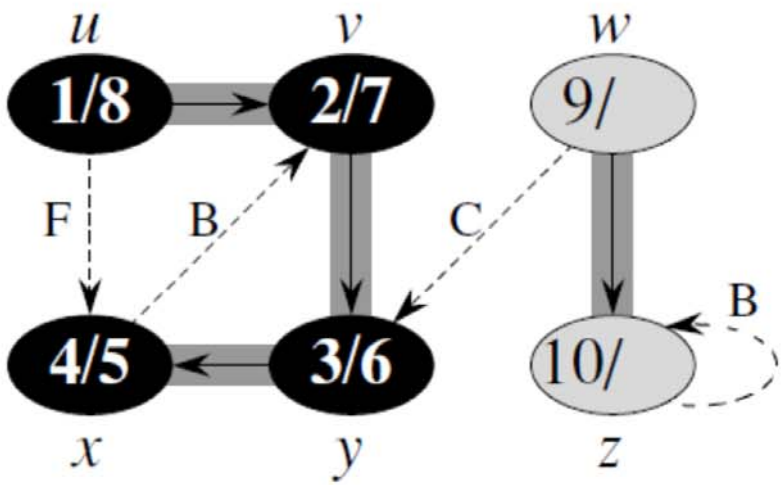
(k)



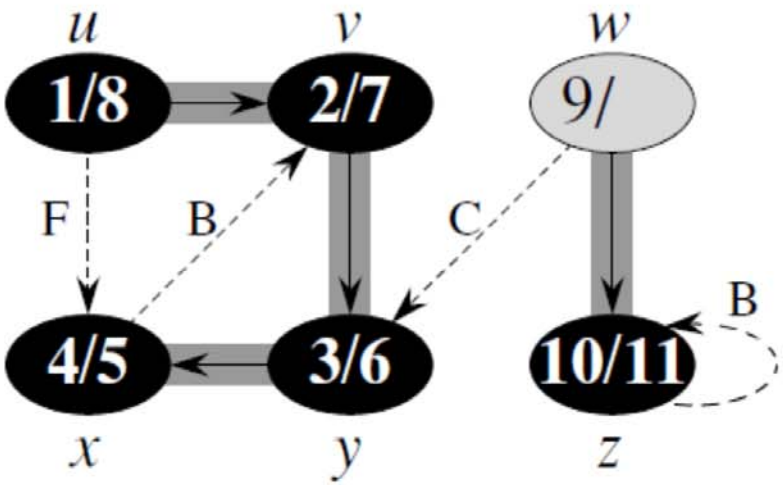
(l)



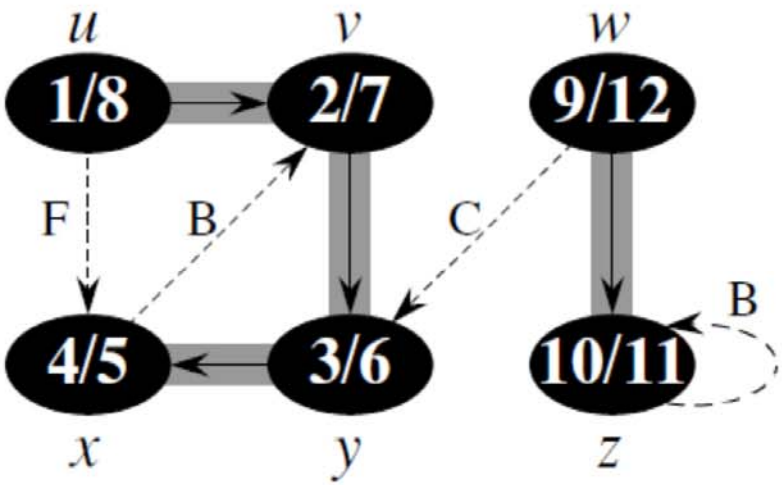
(m)



(n)



(o)



(p)

Running time of DFS

```
DFS-VISIT(u)
1  color[u] ← GRAY   ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]        ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6            then π[v] ← u
7                 DFS-VISIT(v)
8  color[u] ← BLACK           ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7            then DFS-VISIT(u)
```

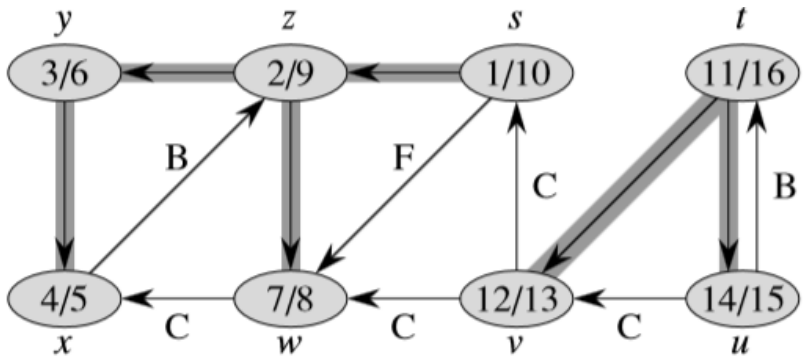
Properties of depth-first search

- u = π[v] if and only if DFS-VISIT(v) was called during a search of u's adjacency list
- vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray
- If we represent the discovery of vertex u with a left parenthesis "(u" and represent its finishing by a right parenthesis "u)", then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested

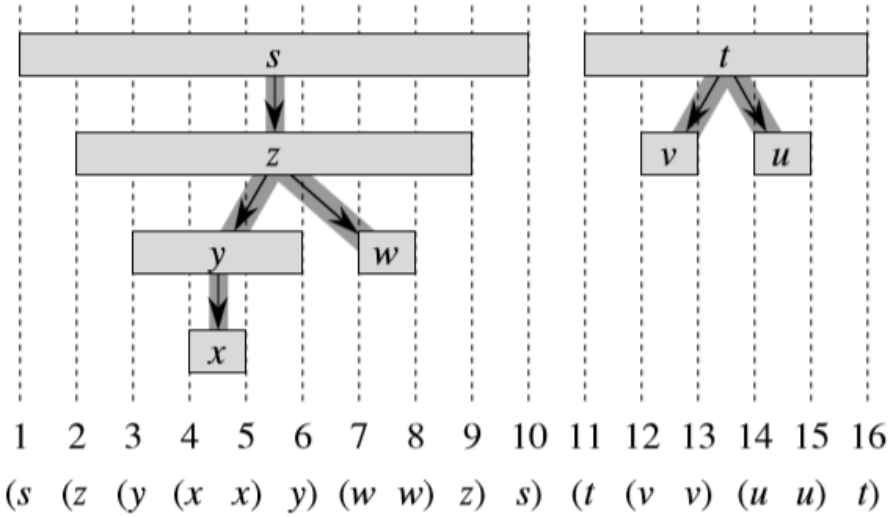
Theorem 22.7 (Parenthesis theorem)

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

(a)


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u

(b)


s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

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 d[u] < d[v] < f[v] < f[u].

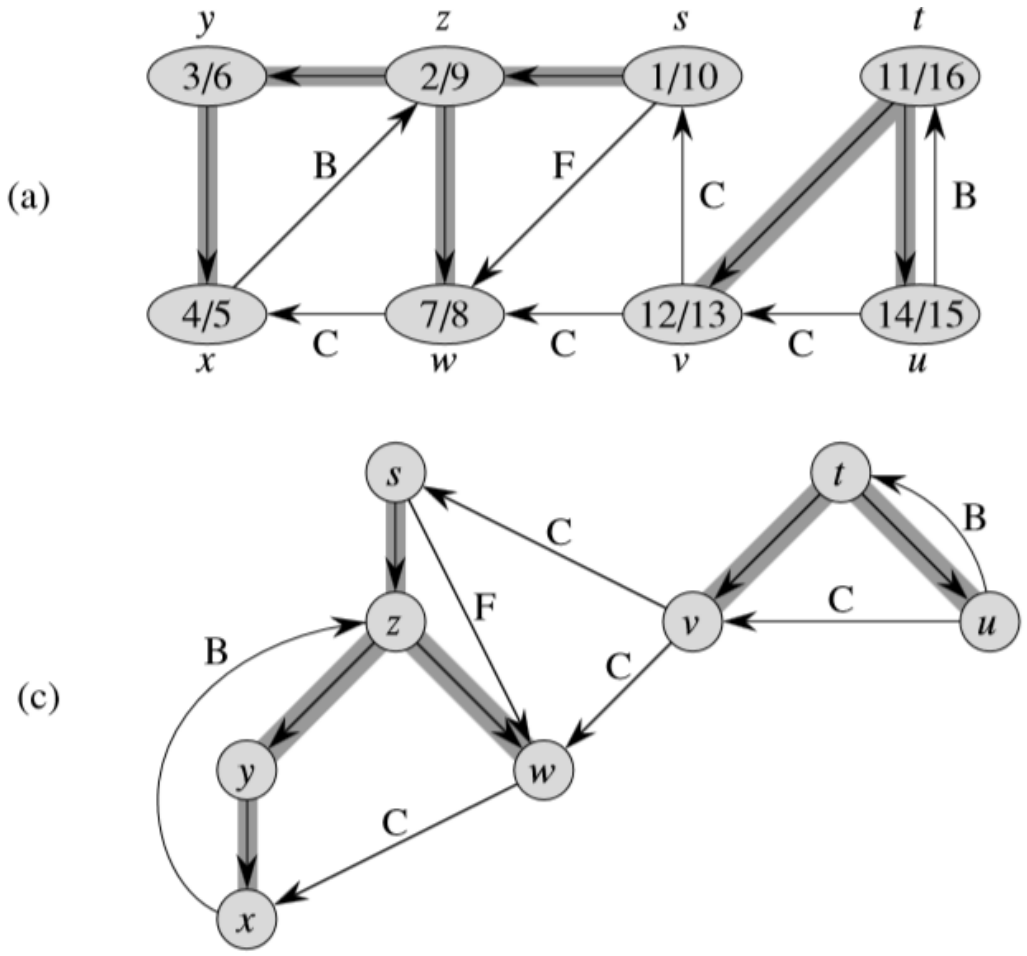
White-path theorem

- 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 d[u] that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Classification of edges

- **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) .
- **Back edges** are those edges (u,v)connecting a vertex u to an ancestor v in a depth-first tree.
  - Self-loops, which may occur in directed graphs, are considered to be back edges
- **Forward edges** are those nontree edges (u,v)connecting a vertex u to a descendant v in a depth-first tree
- **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

- a directed graph is acyclic if and only if a depth-first search yields no "back" edges


(a)
y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u
(c)
s
t
B
C
z
F
v
C
u
B
C
y
w
C
x

- The DFS algorithm can be modified to classify edges as it encounters them
- The key idea is that each edge (u,v) can be classified by the color of the vertex v that is reached when the edge is first explored (except that forward and cross edges are not distinguished):

1) WHITE indicates a tree edge
2) GRAY indicates a back edge, and
3) BLACK indicates a forward or cross edge.
    - it can be shown that such an edge (u,v) is a forward edge if d[u] < d[v] and a cross edge if d[u] > d[v].

Theorem

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