Introduction to Algorithms

Chapter 22 : Elementary Graph Algorithms

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2022

Outline

This chapter presents methods for representing a graph and for searching a graph.

- Section 22.1 discusses the two most common computational representations of graphs: as adjacency lists and as adjacency matrices.
- Section 22.2 presents breadth-first search.
- Section 22.3 presents depth-first search.
- Section 22.4 topologically sorting a directed acyclic graph.
- Section 22.5 finding the strongly connected components of a directed graph

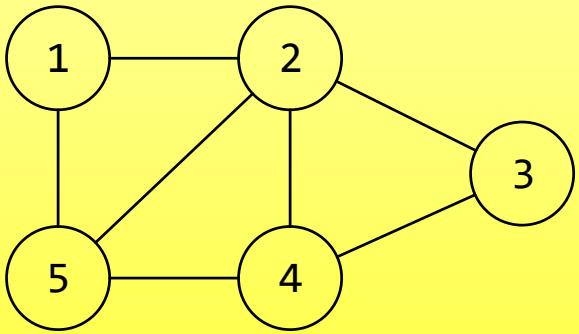
Representations of Graphs

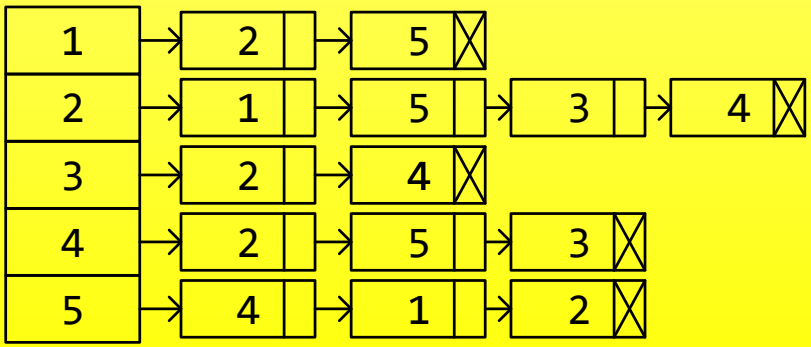
Two standard ways to represent a graph $G = (V, E)$

- **adjacency-lists:** Because the adjacency-list representation provides a compact way to represent sparse graphs—those for which $|E|$ is much less than $|V|^2$—it is usually the method of choice.
- **adjacency-matrix:** When 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.

Two Representations of Graphs

Undirected Graph

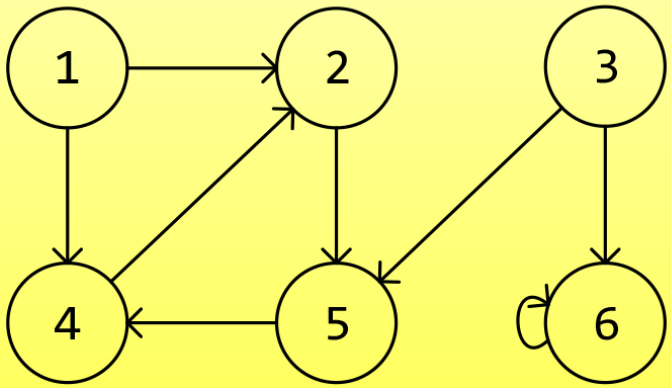


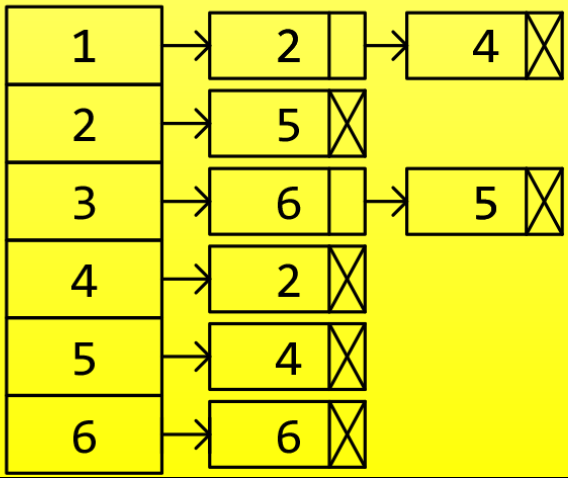


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Two Representations of Graphs

Directed Graph





	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

Adjacency Lists

Memory:

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

Weighted graphs:

We can store the weight $w(u,v)$ of the edge $(u,v) \in E$ with vertex v in u's adjacency list.

Disadvantage:

Adjacency-list provides no quicker way to determine whether a given edge (u,v) is present in graph than to search for v in the adjacency list $u.Adj$.

Adjacency Matrix

Memory:

The adjacency matrix of a graph requires $\Theta(V^2)$ memory.

Weighted graphs:

We can simply store the weight $w(u,v)$ of the edge $(u,v) \in E$ as the entry in row u and column v of the adjacency matrix. If an edge does not exist, we can store a NIL, 0 or ∞. For unweighted, adjacency matrix require only one bit per entry.

Disadvantage:

More memory required if the graph is sparse.

Breadth First Search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.

Given a graph $G = (V, E)$ and a distinguished **source** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex at distance k from s before discovering any vertices at distance $k+1$ until it finds every vertex that is reachable from s. It also produces a "breadth-first tree" with root s that contains all reachable vertices.

Color of Nodes

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 **gray**.

A **black** node means all its adjacent nodes have been discovered.

Breadth First Tree

Breadth-first search constructs a **breadth-first tree**, initially containing only its root, which is the source vertex s.

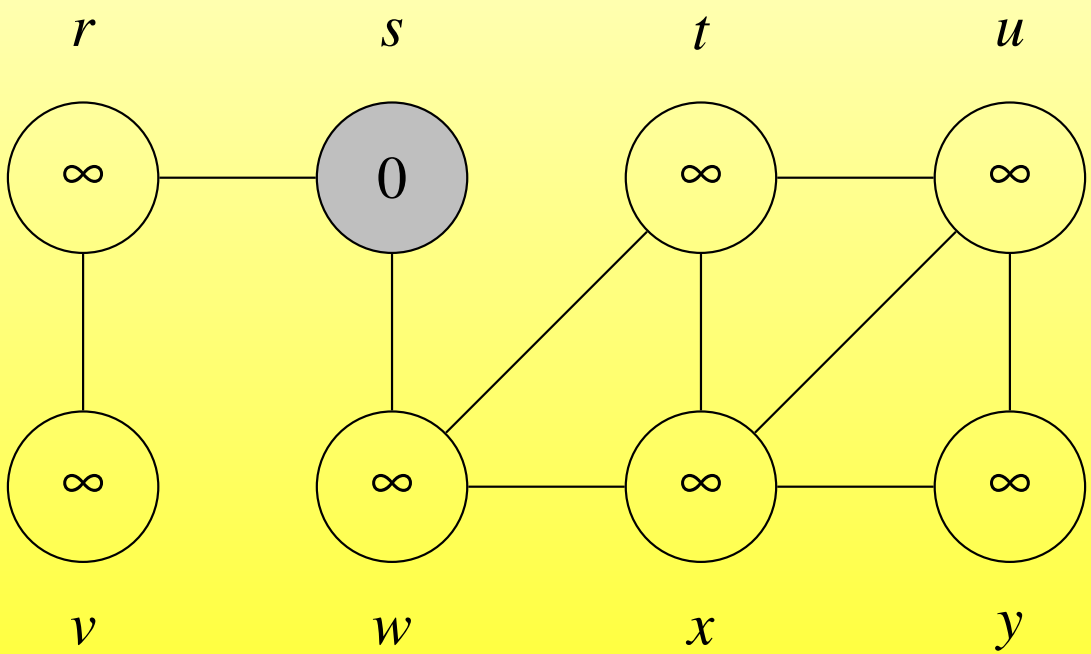
Whenever the search discovers a white vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u,v) are added to the tree. We say that u is the **predecessor** or **parent** of v in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent.

We store the color of each vertex $u \in V$ in the attribute $u.color$ and the predecessor of u in the attribute $u.\pi$. If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $u.\pi = NIL$. The attribute $u.d$ holds the distance from the sources to vertex u computed by the algorithm.

Breadth First Search

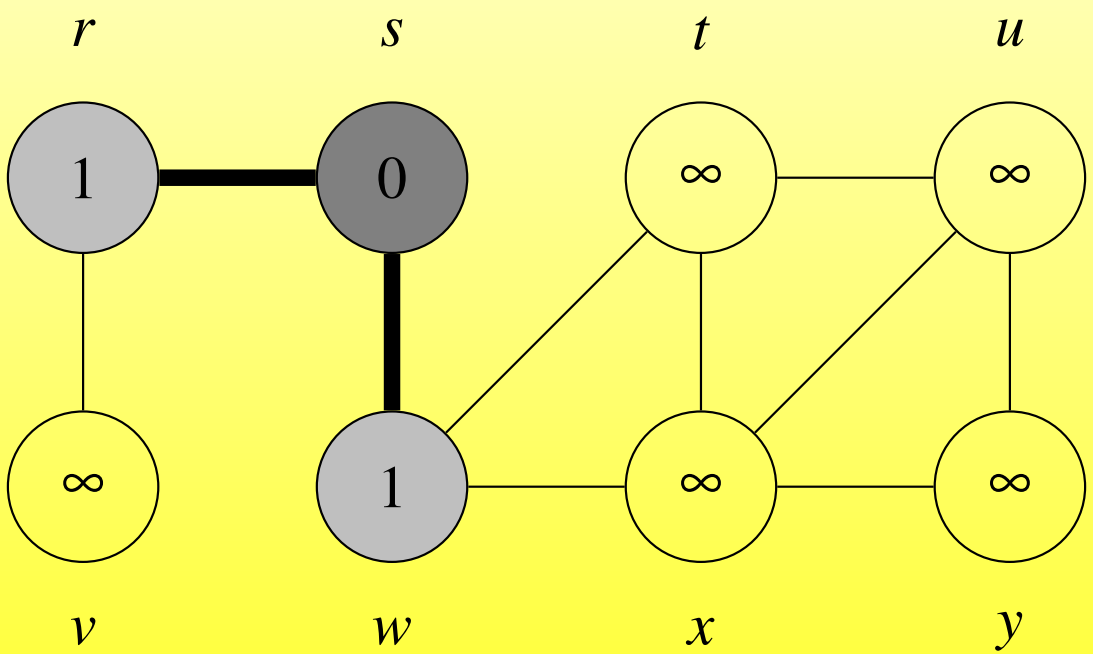
```
BFS(G, s)
1:  for each u ∈ G.V − {s} do
2:      u.color = WHITE
3:      u.d = ∞
4:      u.π = NIL
5:  s.color = GRAY
6:  s.d = 0
7:  s.π = NIL
8:  Q = ∅
9:  ENQUEUE(Q,S)
10: while Q ≠ ∅ do
11:     u =DEQUEUE(Q)
12:     for each v ∈ G.Adj[u] do
13:         If v.color == WHITE
14:             v.color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

Breadth First Search



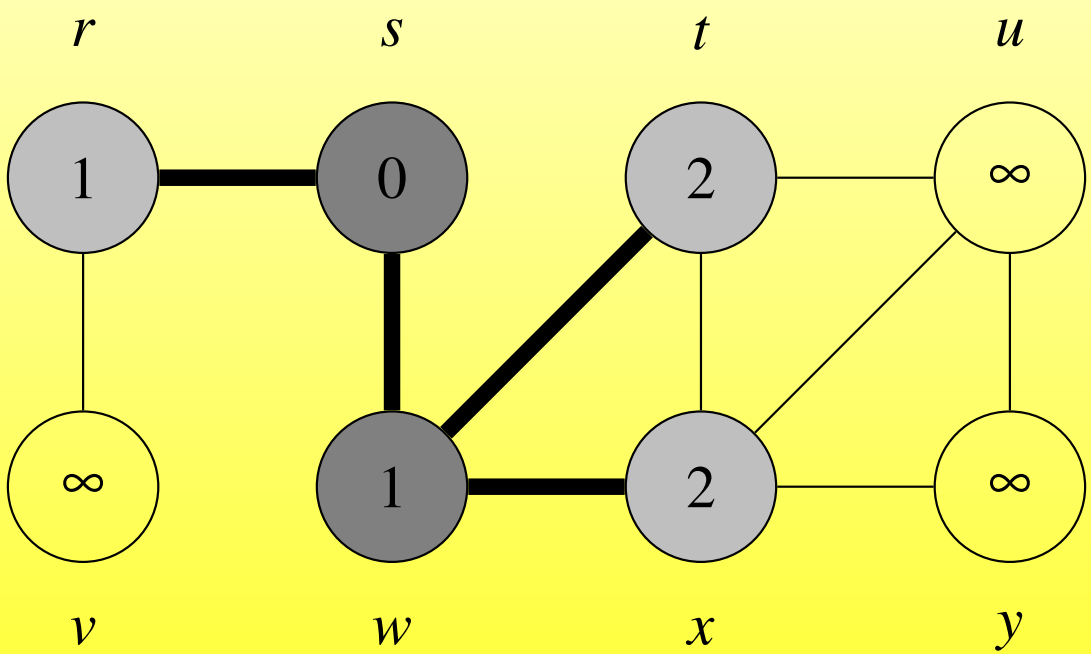
Q	s
d	0
π	nil

Breadth First Search



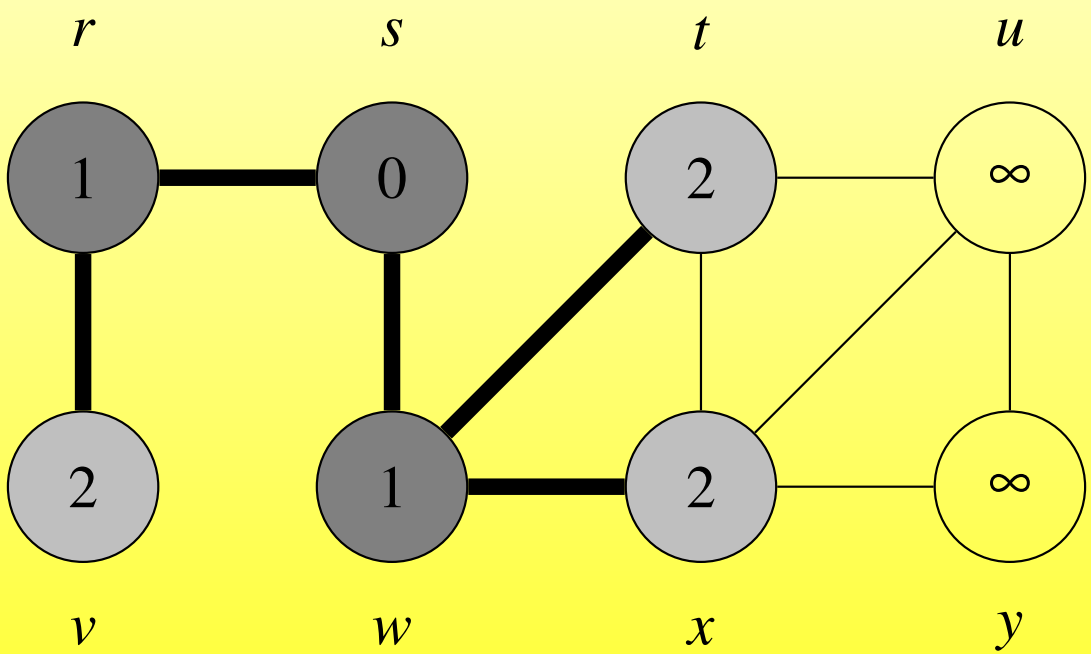
Q	w	r
d	1	1
π	s	s

Breadth First Search



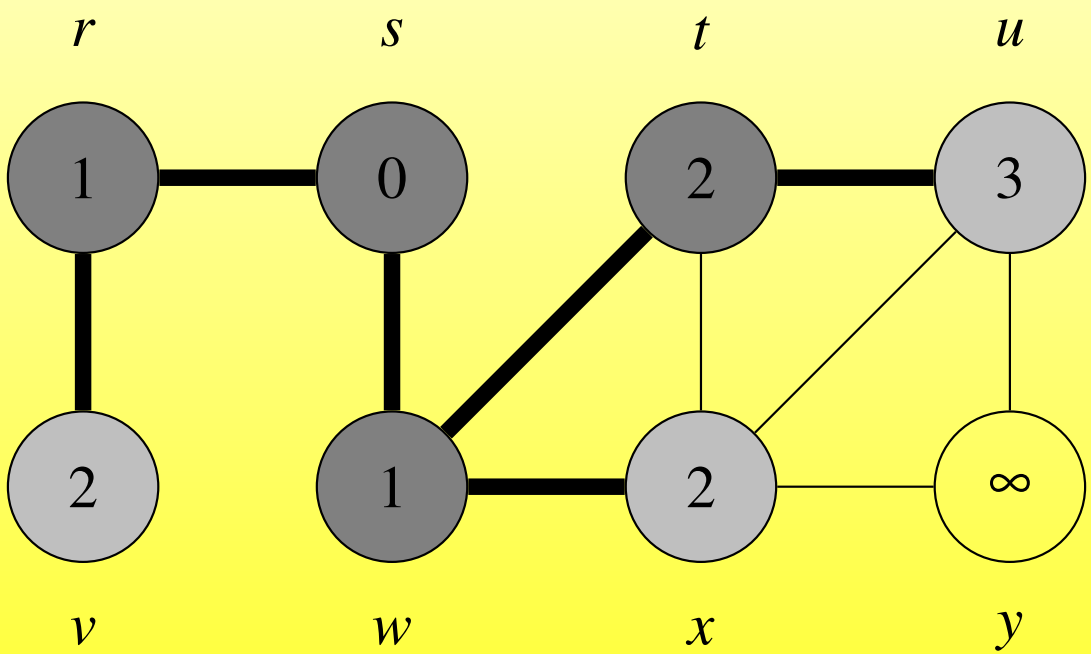
Q	r	t	x
d	1	2	2
π	s	w	w

Breadth First Search



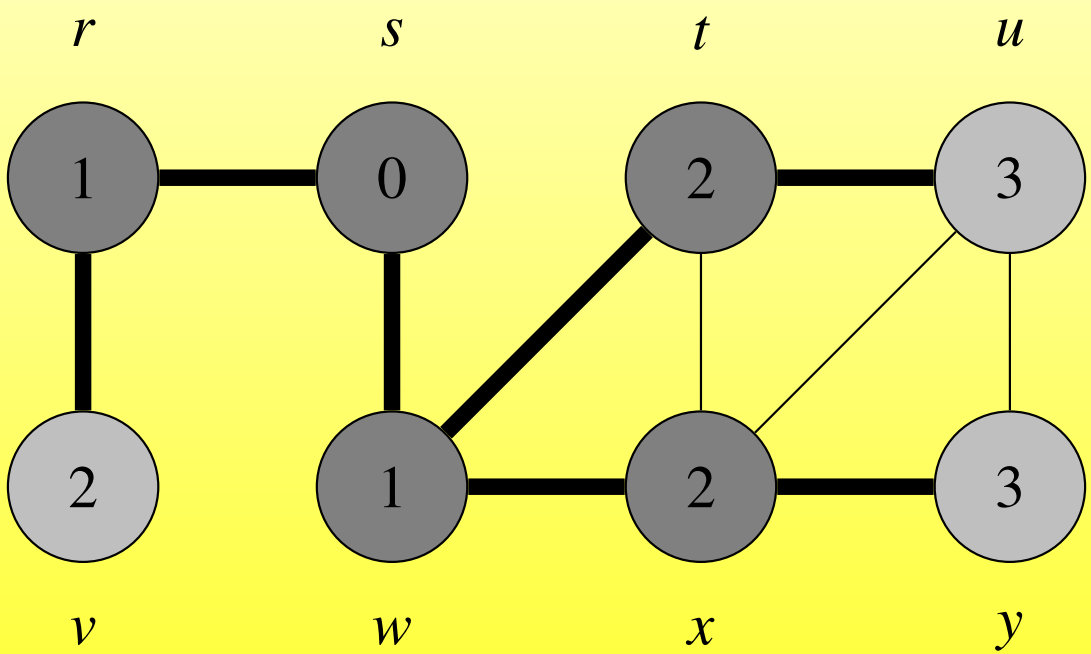
Q	t	x	v
d	2	2	2
π	w	w	r

Breadth First Search



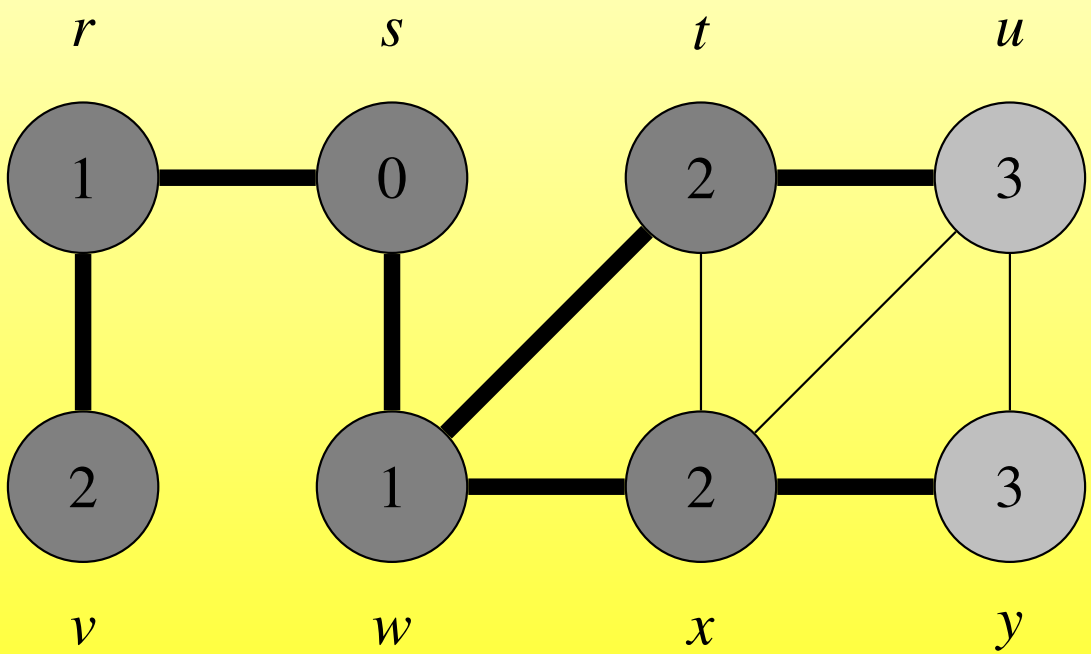
Q	x	v	u
d	2	2	3
π	w	r	t

Breadth First Search



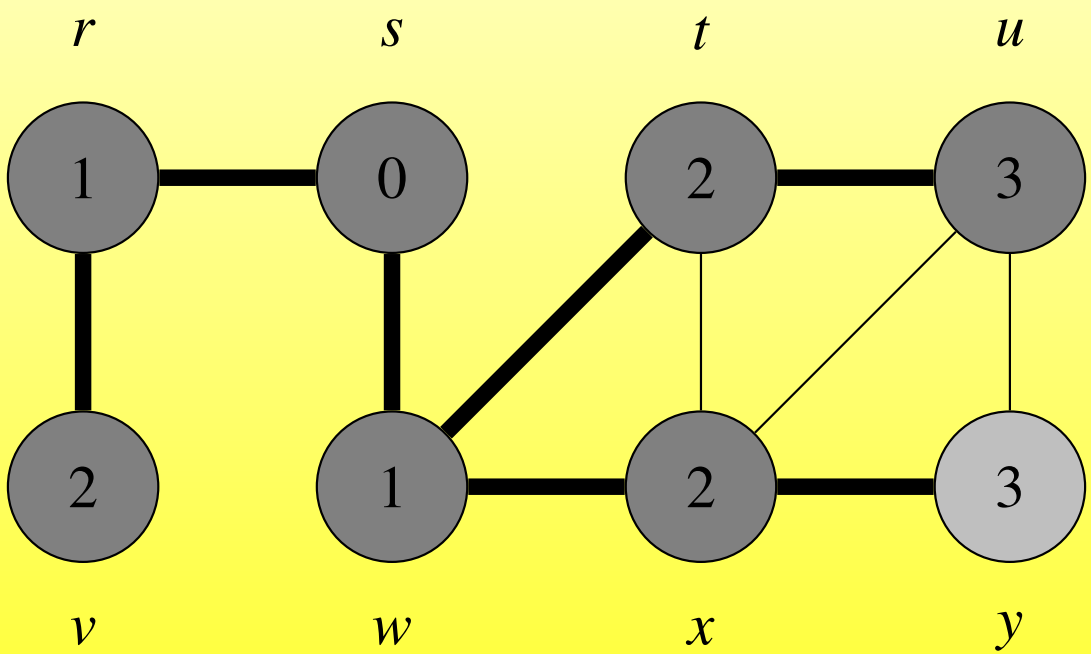
Q	v	u	y
d	2	3	3
π	r	t	x

Breadth First Search



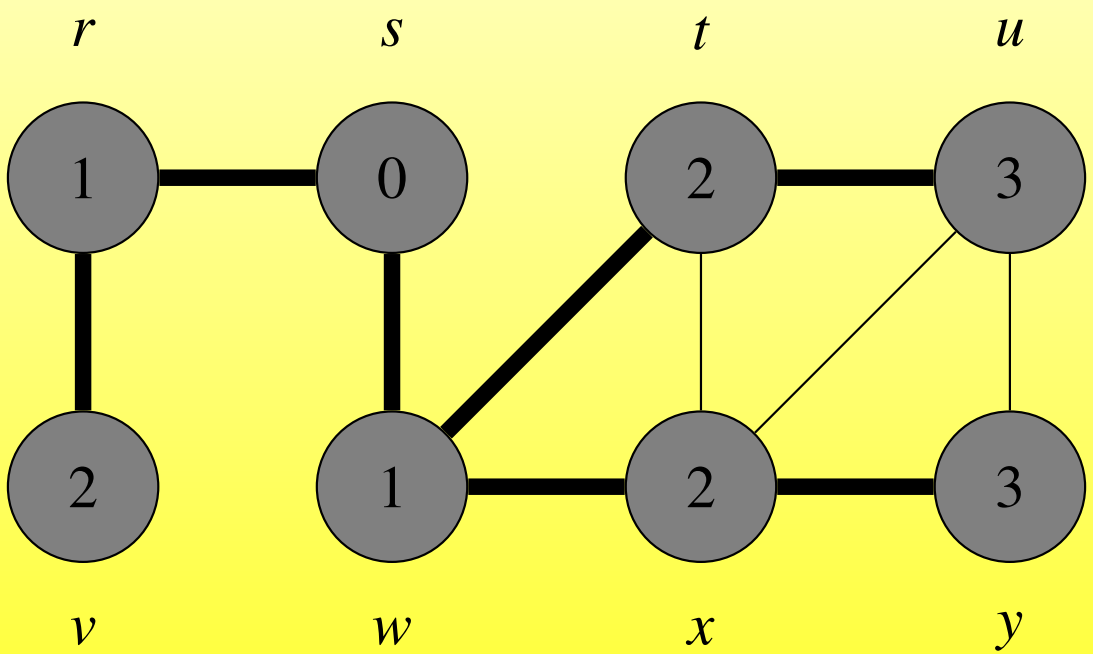
Q	u	y
d	3	3
π	t	x

Breadth First Search



Q	y
d	3
π	x

Breadth First Search



Q
d
π

Breadth First Search

BFS(G, s)
1: **for** each $u \in G.V - \{s\}$ **do**
2: $u.color = WHITE$
3: $u.d = \infty$
4: $u.\pi = NIL$
5: $s.color = GRAY$
6: $s.d = 0$
7: $s.\pi = NIL$
8: $Q = \varnothing$
9: ENQUEUE(Q,S)
10: **while** $Q \neq \varnothing$ **do**
11: $u =$DEQUEUE(Q)
12: **for** each $v \in G.Adj[u]$ **do**
13: **If** $v.color == WHITE$
14: $v.color = GRAY$
15: $v.d = u.d + 1$
16: $v.\pi = u$
17: ENQUEUE(Q, v)
18: $u.color = BLACK$

Breadth First Search - Analysis

Aggregate Analysis

After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once, and so the total time devoted to queue operations is $O(V)$.

Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued. The total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V+E)$.

Shortest Paths

Shortest path distance: The shortest-path distance $\delta(s,v)$ from s to v is the **minimum** number of edges in any path from vertex s to vertex v, if there is no path from s to v, then $\delta(s,v) = \infty$.
Shortest path: A path of length $\delta(s,v)$ from s to v is said to be a shortest path from s to v.

Shortest Paths

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

The procedure BFS builds a breadth-first tree as it searches the graph. The tree corresponds to the π attributes. More formally, for a graph $G = (V, E)$ with source s, we define the **predecessor subgraph** of G as $G_\pi(V_\pi, E_\pi)$, where $V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$ 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 $v \in V_\pi$, G_π contains a unique simple path from s to v that is also a shortest path from s to v in G. A breadth-first tree is in fact a tree, since it is connected and $|E_\pi| = |V_\pi| - 1$. We call the edges in E_π tree edges.

Breadth First Trees

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.

Proof: Line 16 of BFS sets $v.\pi = u$ if and only if $(u, v) \in E$ and $\delta(s, v) < \infty$, and thus V_π consists of the vertices in V reachable from s. Since G_π from a tree, it contains a unique simple path from s to each vertex in V_π. By applying Theorem 22.5 inductively, we conclude that every such path is a shortest path in G.

Breadth-first trees

PRINT-PATH(G, s, v)
1: **if** $v == s$ **then**
2: print s
3: return
4: **if** $v.\pi == NIL$ **then**
5: print “no path from “s” to ” v “exists”
6: **else**
7: PRINT-PATH($G, s, v.\pi$)

Depth-first search

Strategy: to search deeper in the graph whenever possible.

Depth-first search 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. This process continues until we have discovered all the vertices that are reachable from the original source vertex.

If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Depth First Trees

The predecessor subgraph of a depth-first search: $G_\pi = (V, E_\pi)$ where $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq NIL\}$
The predecessor subgraph of a depth-first search forms a **depth-first forest** comprising several **depth-first trees**. The edges in E_π are tree edges.
Vertices are colored during the search to indicate their state.

- Each vertex is initially white.
- It is grayed when it is discovered in the search.
- It is blackened when it is finished, that is, when its adjacency list has been examined completely.

This technique guarantees that each vertex ends up in exactly one depth-first tree, so these trees are disjoint.

Depth First Trees

Besides creating a depth-first forest, depth-first search also timestamps each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v).

These timestamps are integers between 1 and $2|V|$. For every vertex u, vertex u is **WHITE** before time $d.u$, **GRAY** between time $d.u$ and time $d.f$, and **BLACK** thereafter.

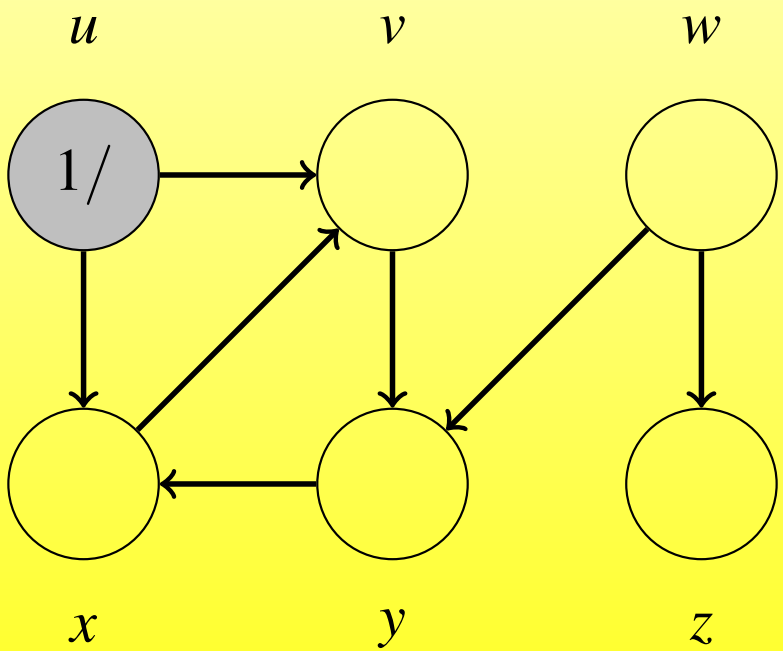
The procedure DFS below records when it discovers vertex u in the variable $u.d$ and when it finishes vertex u in the variable $u.f$.

Depth First Search

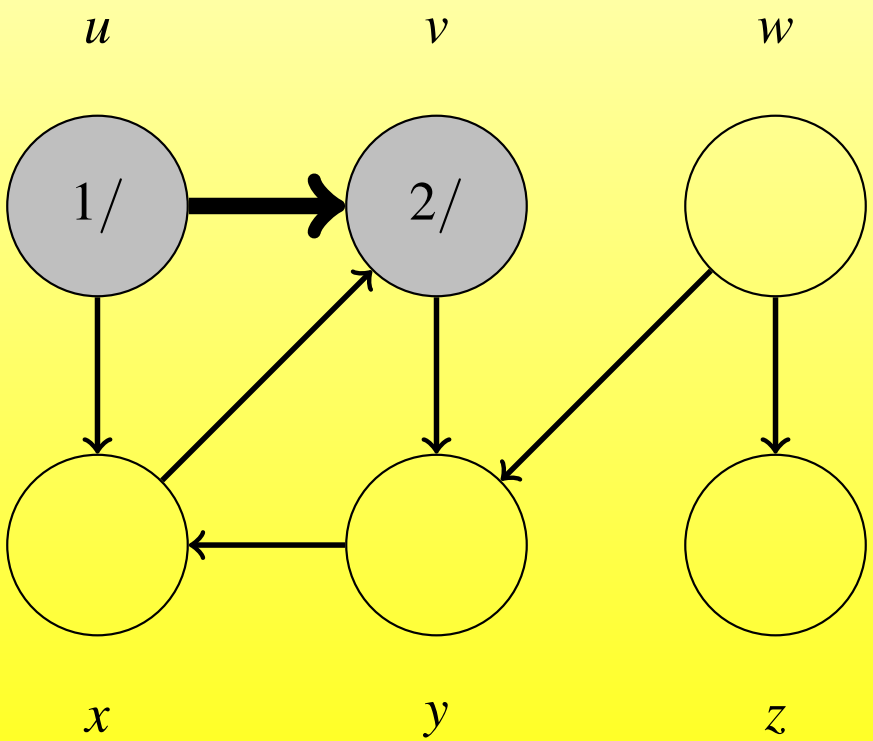
DFS(G)
1: **for** *each vertex* $u \in G.V$ **do**
2: $u.color = WHITE$
3: $u.\pi = NIL$
4: time=0
5: **for** *each vertex* $u \in G.V$ **do**
6: **if** $u.color == WHITE$ **then**
7: DFS-VISIT(G, u)

DFS-VISIT(G, u)
1: $time = time + 1$
2: $u.d = time$
3: $u.color = GRAY$
4: **for** *each* $v \in G.Adj[u]$ **do**
5: **if** $v.color == WHITE$ **then**
6: $v.\pi = u$
7: DFS-VISIT(G, v)
8: $u.color = BLACK$
9: $time = time + 1$
10: $u.f = time$

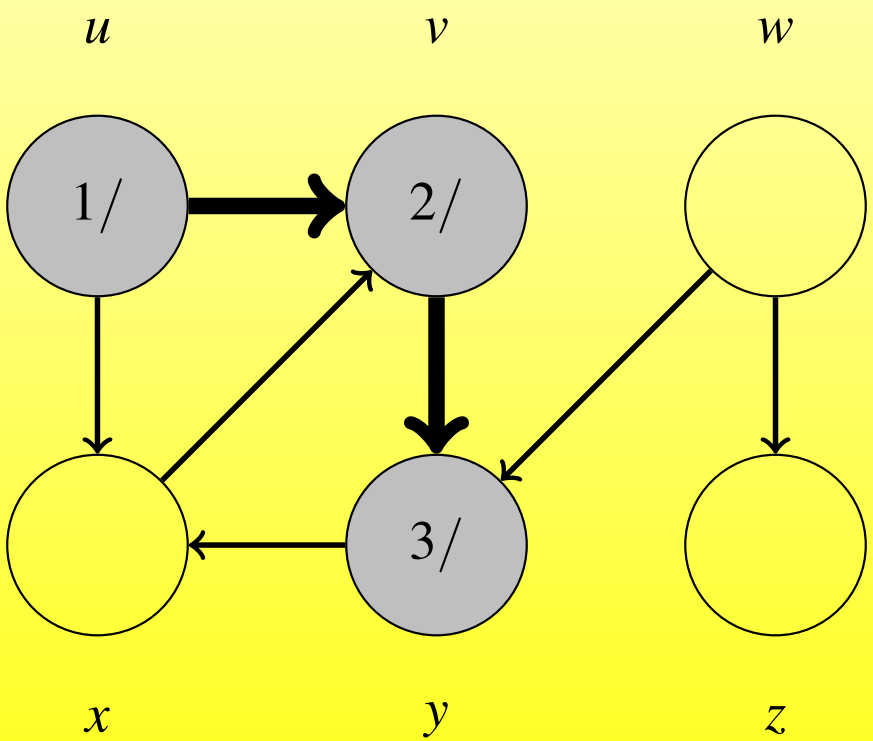
Depth First Search - Example



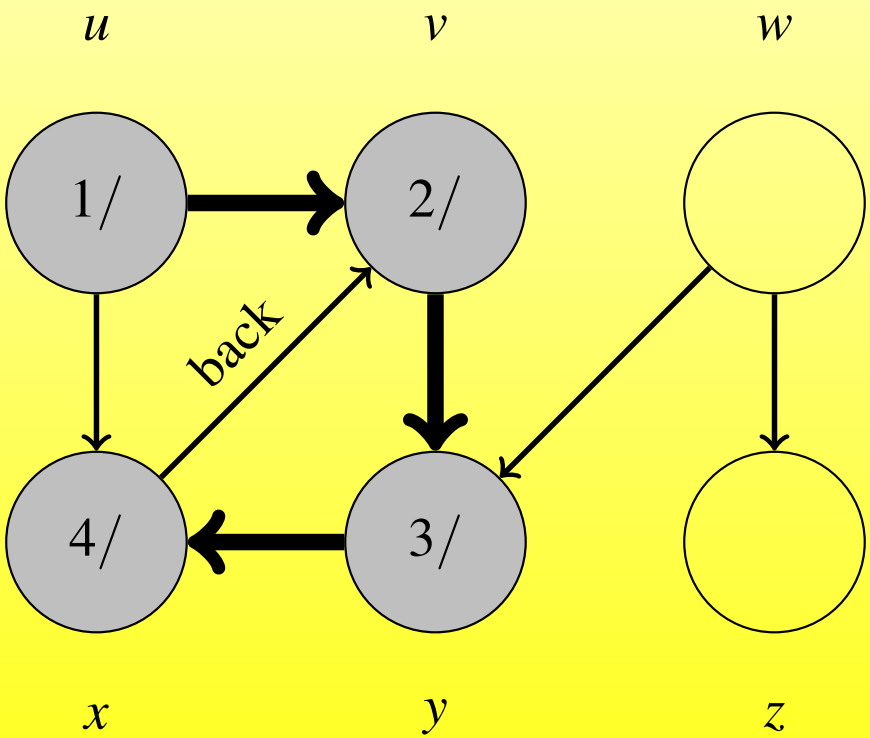
Depth First Search - Example



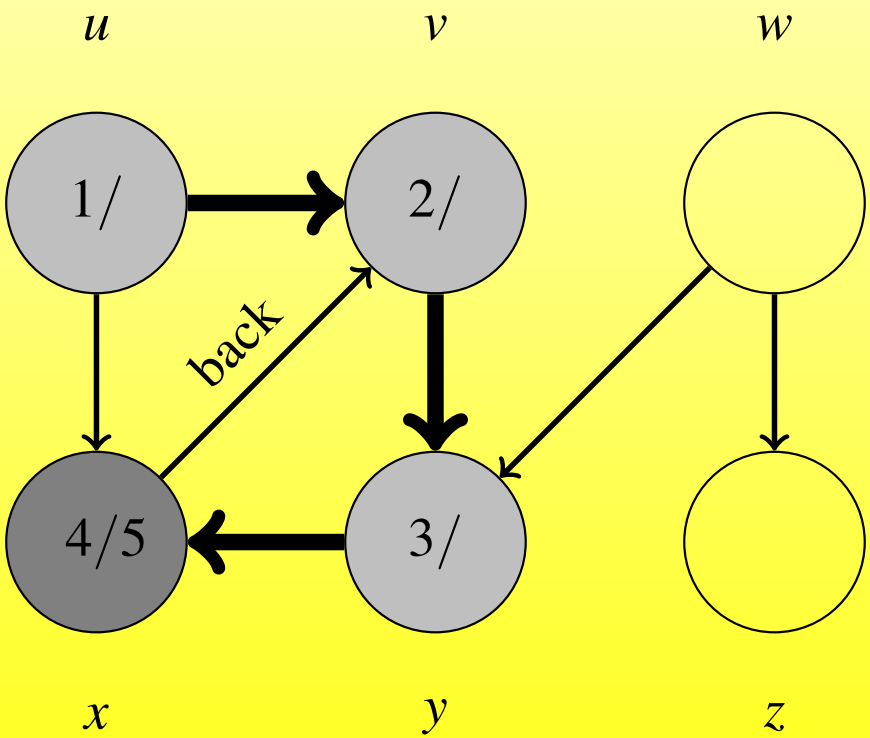
Depth First Search - Example



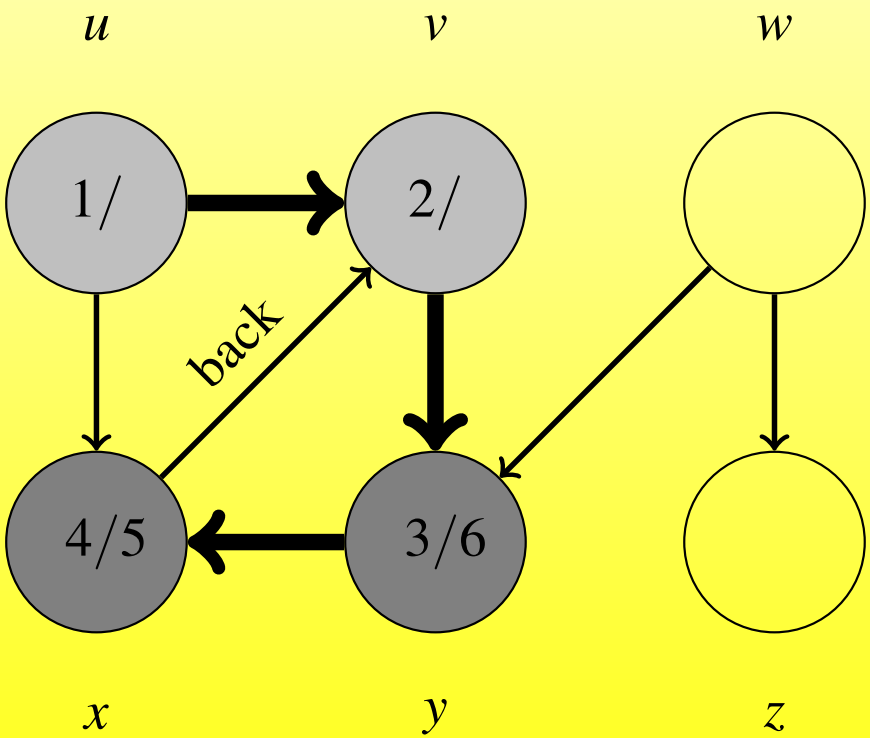
Depth First Search - Example



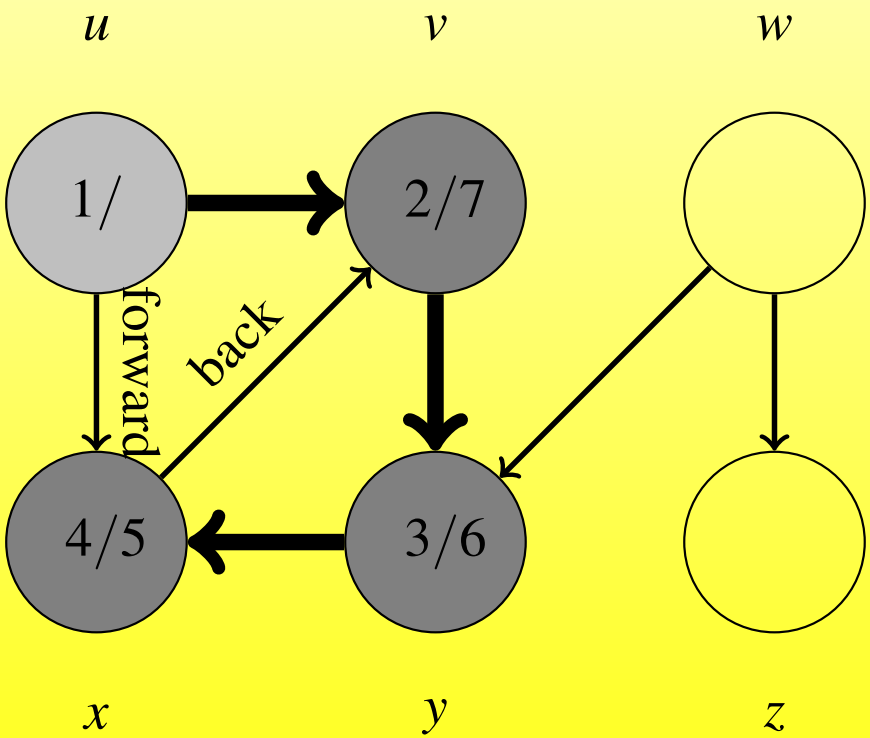
Depth First Search - Example



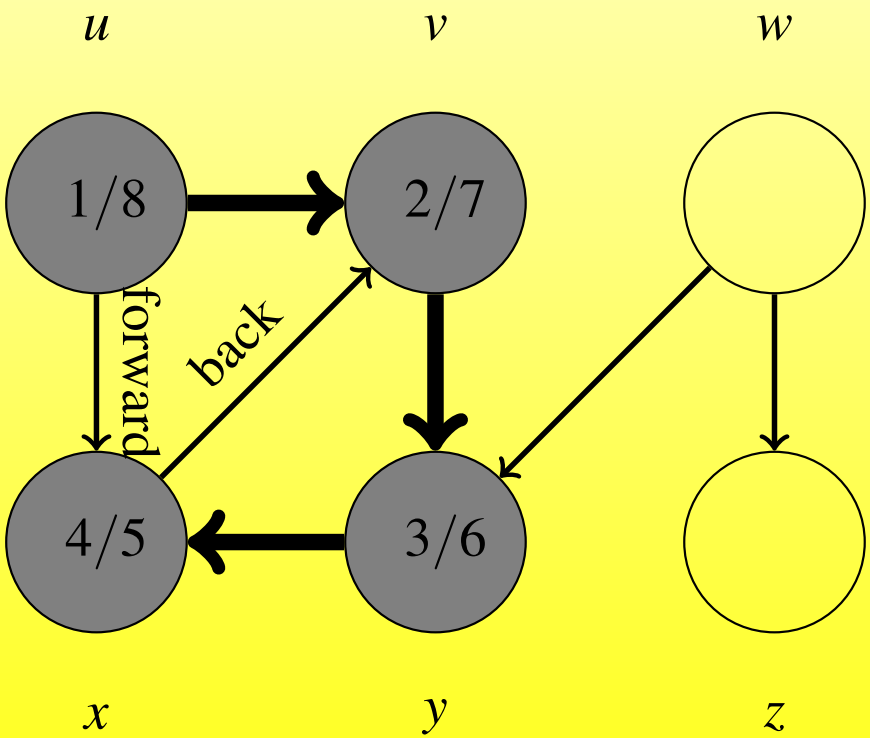
Depth First Search - Example



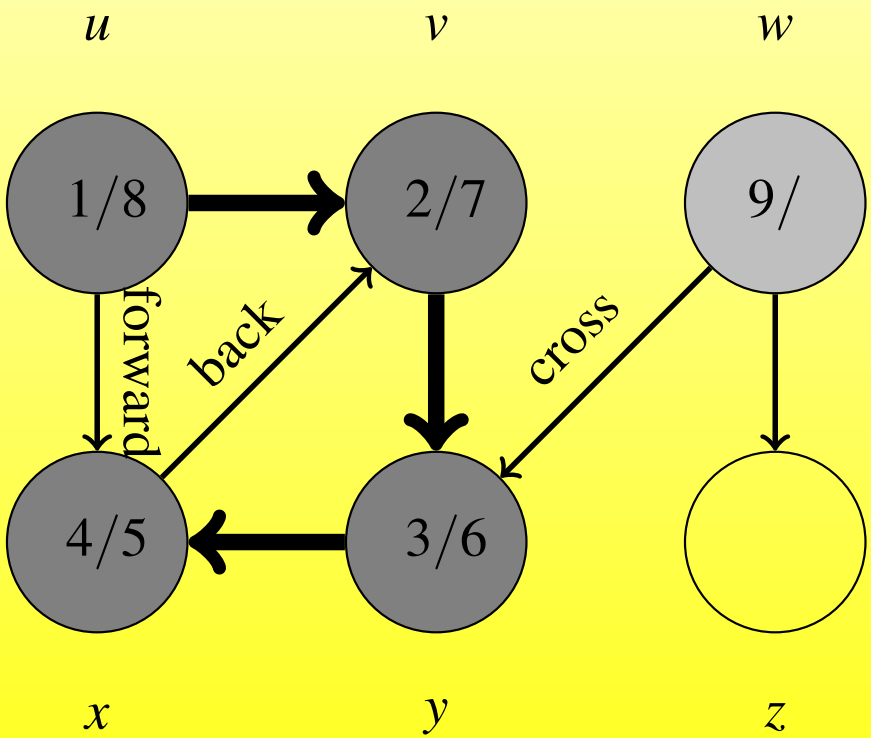
Depth First Search - Example



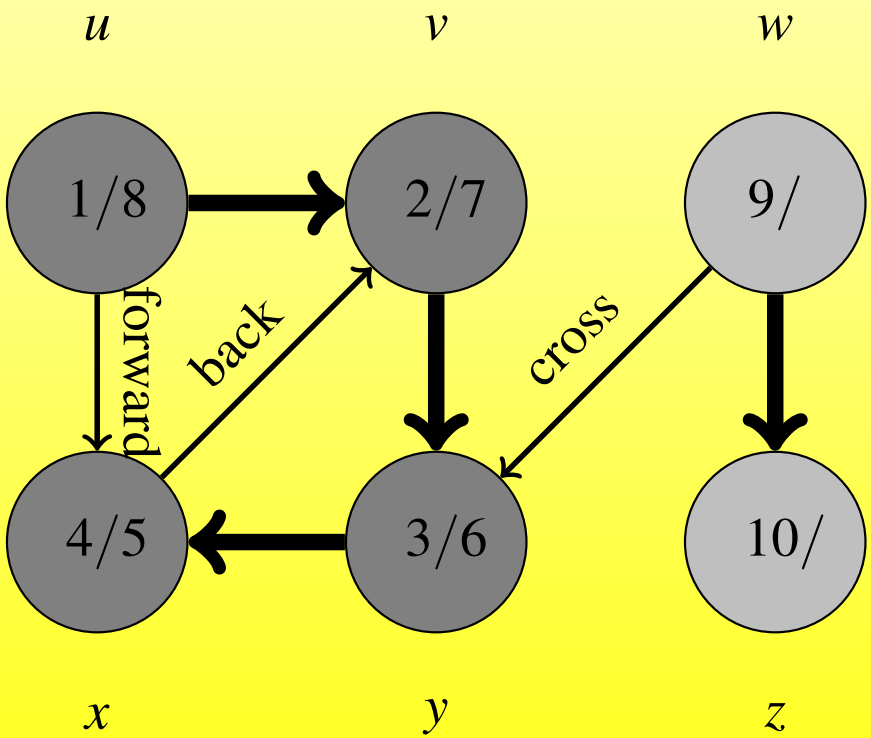
Depth First Search - Example



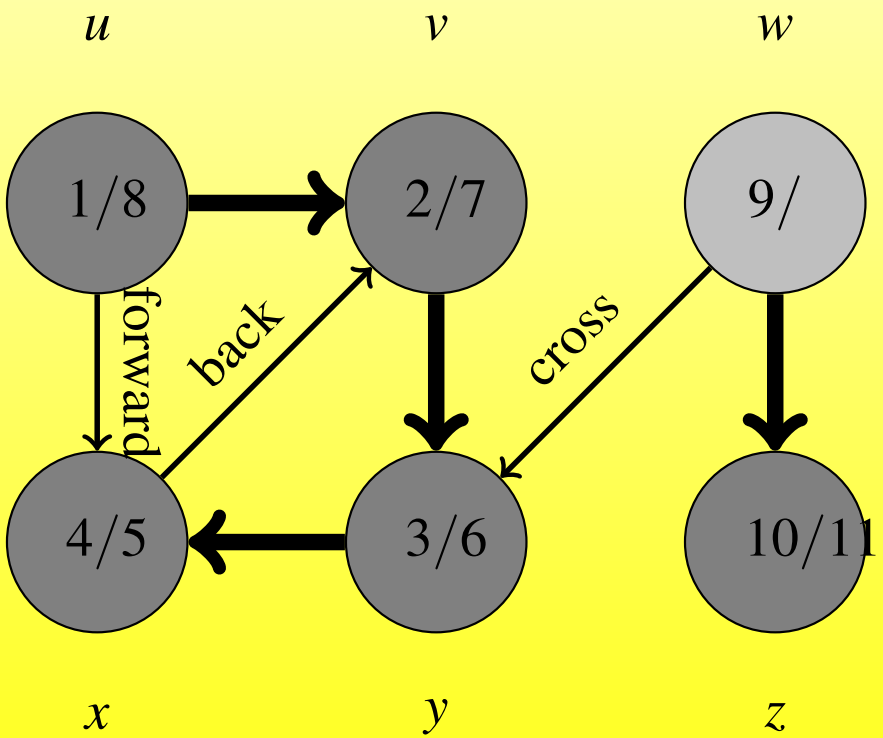
Depth First Search - Example



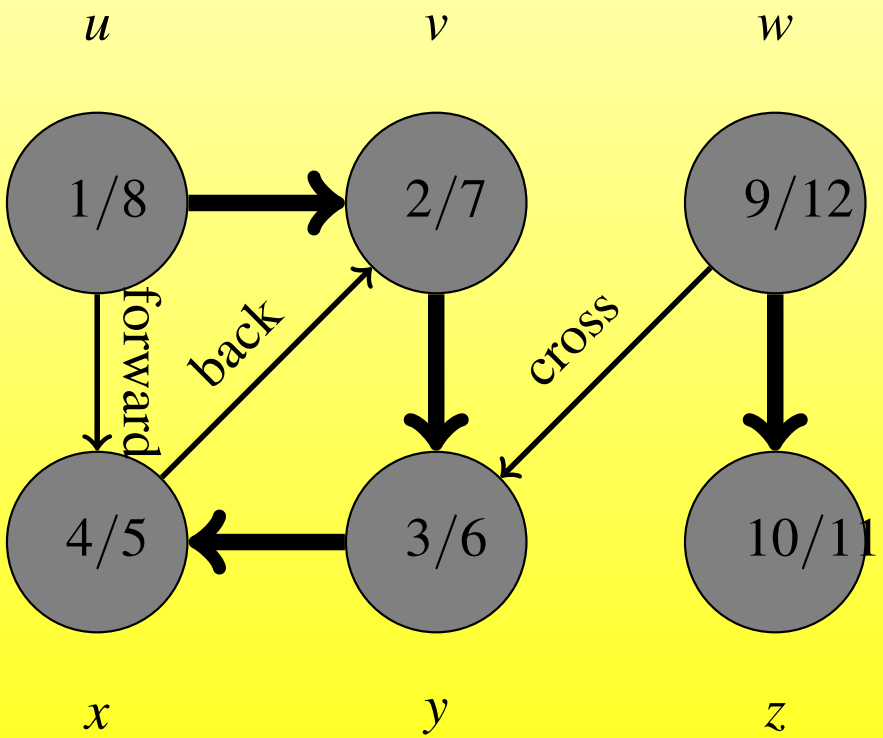
Depth First Search - Example



Depth First Search - Example



Depth First Search - Example



Depth First Search

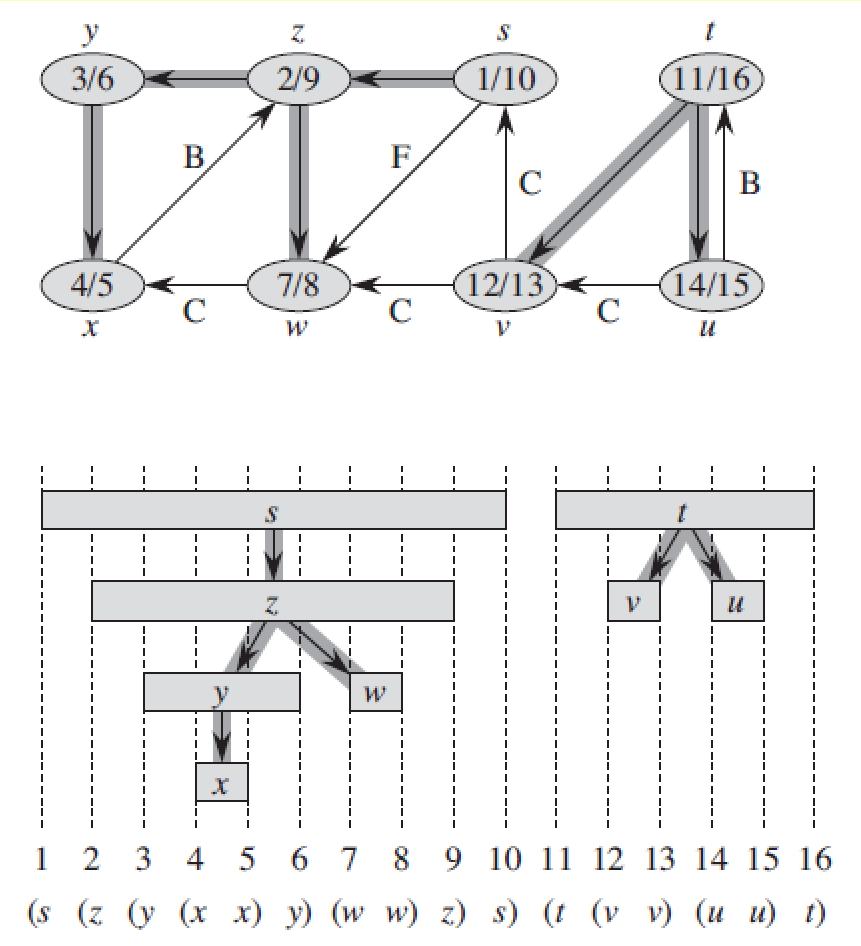
```
DFS(G)
1: for each vertex u ∈ G.V do
2:     u.color = WHITE
3:     u.π = NIL
4: time=0
5: for each vertex u ∈ G.V do
6:     if u.color == WHITE then
7:         DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u)
 1: time = time + 1
 2: u.d = time
 3: u.color = GRAY
 4: for each v ∈ G.Adj[u] do
 5:     if v.color == WHITE then
 6:         v.π = u
 7:         DFS-VISIT(G,v)
 8: u.color = BLACK
 9: time = time + 1
10: u.f = time
```

Depth First Search - Analysis

Time complexity: The loops on lines 1-3 and lines 5-7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT s invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G,v), the loop on lines 4-7 executes $|Adj[v]|$ times. Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$, the total cost of executing lines 4-7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V+E)$.

Depth First Search - Analysis



Properties of Depth First Search

The most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT.

Another important property of depth-first search is that discovery and finishing times have **parenthesis structure**. 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.

Properties of Depth First Search

Theorem 22.7: 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 v nor u 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.

Properties of Depth First Search

Proof: We discuss the following cases:

- $u.d < v.d$: There are two subcases to consider
  - $v.d < u.f$: In this case, v is a descendant of u, then the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$
  - $u.f < v.d$: The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint, so neither vertex was discovered while the other was gray, and neither vertex is a descendant of the other.
- $v.d < u.d$: This case is similar, with the roles of u and v reversed in the above argument.

Properties of Depth First Search

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 $u.d < v.d < v.f < u.f$.

Proof: Immediate from Theorem 22.7.

The next theorem gives another important characterization of when one vertex is a descendant of another in the depth-first forest.

Properties of Depth First Search

Corollary 22.8:(White-path theorem)

In a depth-first forest of a (directed or undirected) graph G, 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.

Properties of Depth First Search

Proof $\Rightarrow$**:** If $v == u$, then the path from u to v contains just vertex u, which is still white when we set the value of $u.d$. Now, suppose that v is a proper descendant of u in the depth-first forest. By Corollary 22.8, $u.d < v.d$, and so v is white at time $u.d$. Since v can be any descendant of u, all vertices on the unique simple path from u to v in the depth-first forest are white at time $u.d$.

Properties of Depth First Search

$\Leftarrow$: Suppose that there is a path of white vertices from u to v at time $u.d$, but v does not become a descendant of u in the depth-first tree. Assume that every vertex other than v along the path becomes a descendant of u. Let w be the predecessor of v on the path, so that w is a descendant of u. By Corollary 22.8, $w.f < u.f$. Because v must be discovered after u is discovered, but before w is finished, we have $u.d < v.d < w.f < u.f$. Theorem 22.7 then implies that the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$. By Corollary 22.8, v must after all be a descendant of u.

Classification of Edge

The depth-first search can be used to classify the edges of the input graph $G = (V, E)$. Four edge types are defined in terms of the depth-first forest G_π produced by a depth-first search on G.

- 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. We consider self-loops, which may occur in directed graphs, 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.

Classification of Edge

The DFS algorithm has enough information to classify some 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):

- WHITE indicates a tree edge,
- GRAY indicates a back edge, and
- BLACK indicates a forward or cross edge.

Classification of Edge

The first case is immediate from the specification of the algorithm.

The gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations. Exploration always proceeds from the deepest gray vertex, so an edge that reaches another gray vertex reaches an ancestor.

The third case handles the remaining possibility; an edge (u, v) is a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$.

Classification of Edge

In an undirected graph, there may be some ambiguity in the type classification, since (u,v) and (v,u) are really the same edge.

In such a case, the edge is classified as the first type in the classification list that applies.

Forward and cross edges never occur in a depth-first search of an undirected graph.

Classification of 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.

Proof: Let (u,v) be an arbitrary edge of G, and suppose without loss of generality that $u.d < v.d$. Then the search must discover and finish v before it finishes u (while u is gray), since v is on u's adjacency list. If the first time that the search explores edge (u,v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise, the search would have explored this edge already in the direction from v to u. Thus, (u,v) becomes a tree edge. If the search explores (u,v) first in the direction from v to u, then (u,v) is a back edge, since u is still gray at the time the edge is first explored.

Topological Sort

DAG: a directed acyclic graph

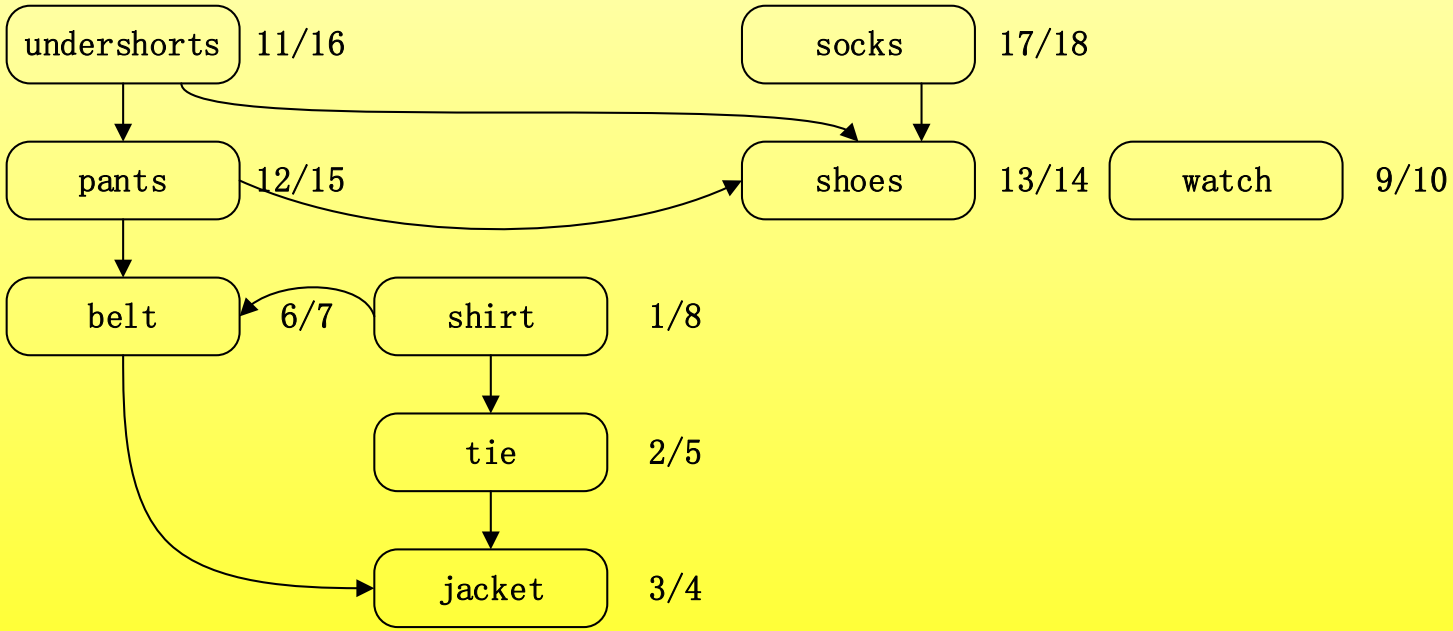
A **topological sort** of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

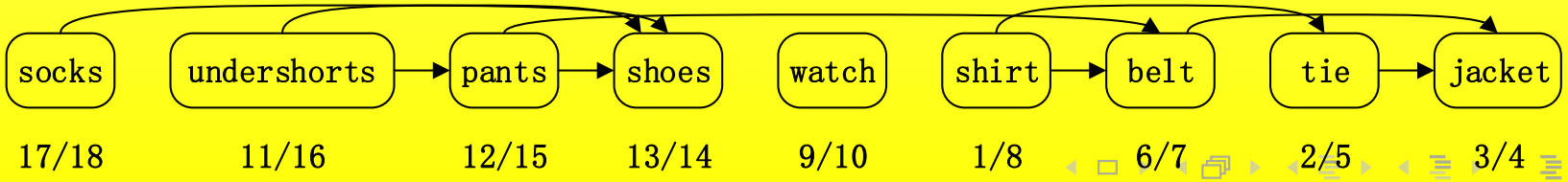
TOPOLOGICAL-SORT(G)

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: We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

Topological Sort





Topological Sort

Lemma 22.11: A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

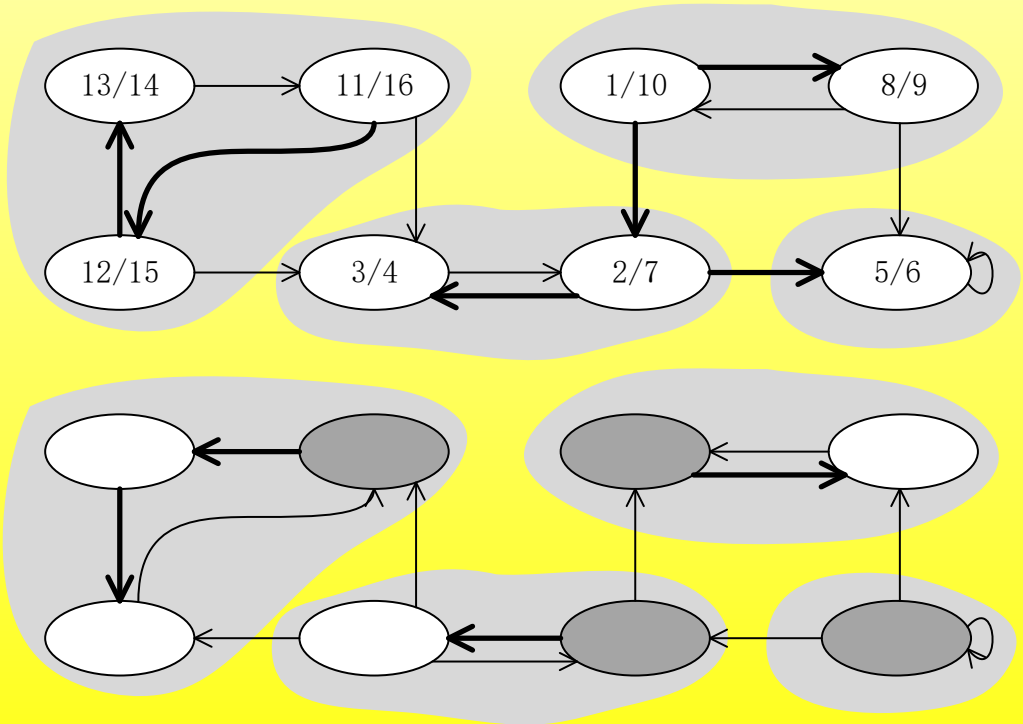
Theorem 22.12: TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.
Hint: When exploring $(u,v) \in E$, v can not be gray, or else it will be a back edge. So, we must have $v.f < u.f$, which means v will be sorted righthand side to (i.e., after) u.

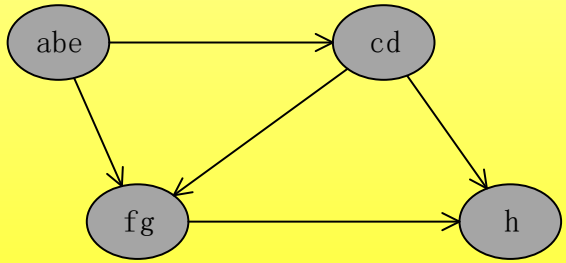
Strongly Connected Components

We now consider a classic application of depth-first search: **decomposing a directed graph** into its strongly connected components. This section shows how to do so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition.

A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subset V$ 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





Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS(G)

1: call DFS(G) to compute finishing time $u.f$ 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

Strongly Connected Components

The idea behind this algorithm comes from a key property of the **component graph** $G^{SCC} = (V^{SCC}, E^{SCC})$, which we define as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, v_2, v_3, \ldots, v_k\}$, and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E_{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$.

Strongly Connected Components

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 $v' \rightsquigarrow v$.

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 $f(C) < f(C')$.

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