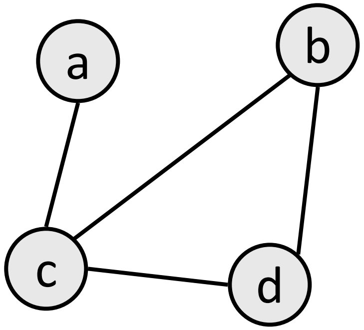
Graph Theory

Design and Analysis of Algorithms

Fall 2022

Graphs

- **graph**: A data structure containing:
  - a set of **vertices** V, *(sometimes called nodes)*
  - a set of **edges** E, where an edge represents a connection between 2 vertices.
    - Graph $G = (V, E)$
    - an edge is a pair (v, w) where v, w are in V

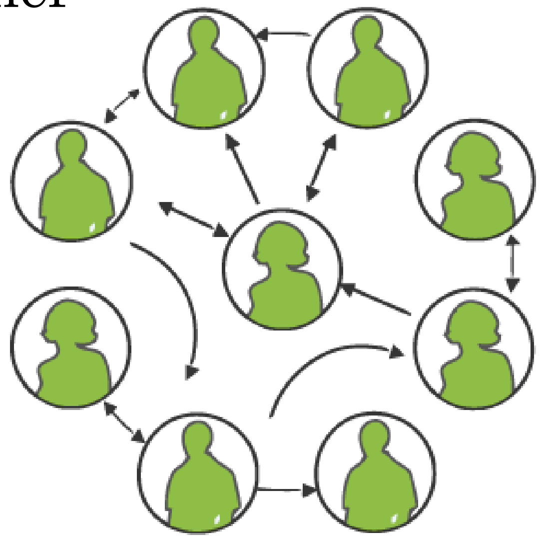


- the graph at right:
  - $V = \{a, b, c, d\}$
  - $E = \{(a, c), (b, c), (b, d), (c, d)\}$

- **degree**: number of edges touching a given vertex.
  - at right: a=1, b=2, c=3, d=2

Graph examples

- For each, what are the vertices and what are the edges?
  - Web pages with links
  - Network broadcast routing
  - Web crawling
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Facebook friends
  - Course pre-requisites
  - Family trees
  - Paths through a maze
  - Solving puzzles and games



Applications of Graphs

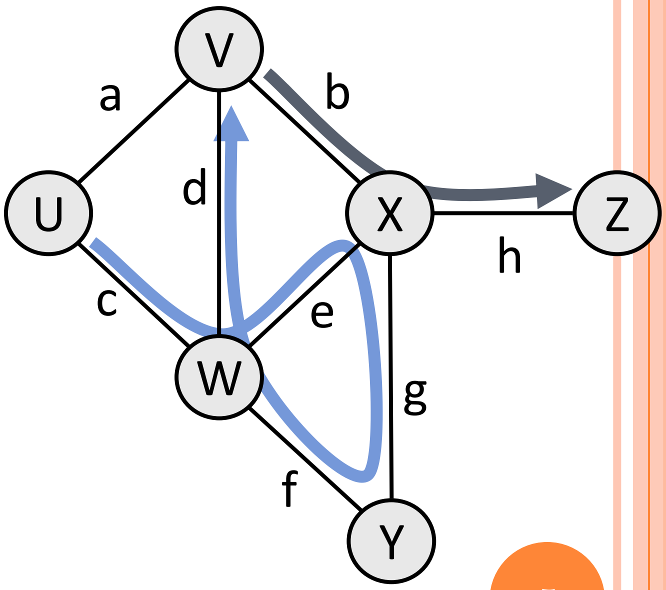
- Driving Map
  - Edge = Road
  - Vertex = Intersection
  - Edge weight = Time required to cover the road
- Airline Traffic
  - Vertex = Cities serviced by the airline
  - Edge = Flight exists between two cities
  - Edge weight = Flight time or flight cost or both
- **Computer networks**
  - Vertex = Server nodes
  - Edge = Data link
  - Edge weight = Connection speed

Paths

- **path**: A path from vertex a to b is a sequence of edges that can be followed starting from a to reach b.
  - can be represented as vertices visited, or edges taken
  - example, one path from V to Z: {b, h} or {V, X, Z}
  - What are two paths from U to Y?

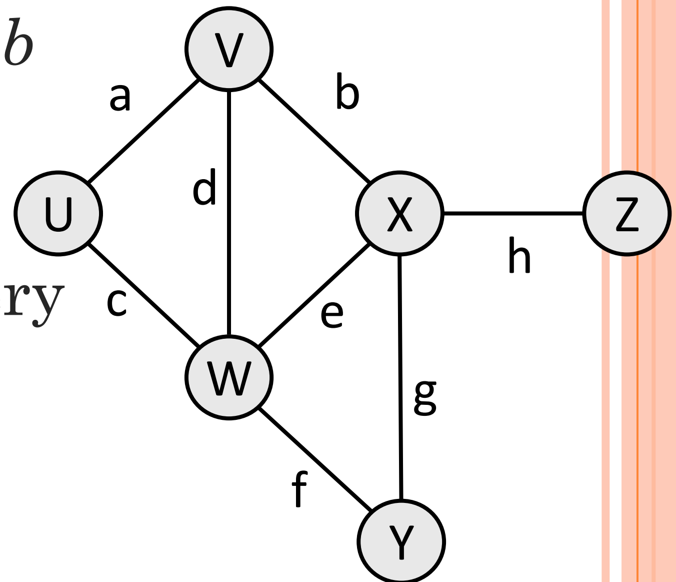
- **path length**: Number of edges
- contained in the path.

- **neighbor** or **adjacent**: Two vertices connected directly by an edge.
  - example: V and X



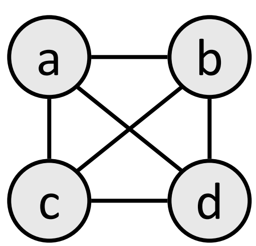
Reachability, connectedness

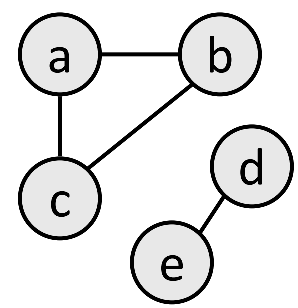
- **reachable**: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.



- **connected**: A graph is *connected* if every vertex is reachable from any other.
  - Is the graph at top right connected?

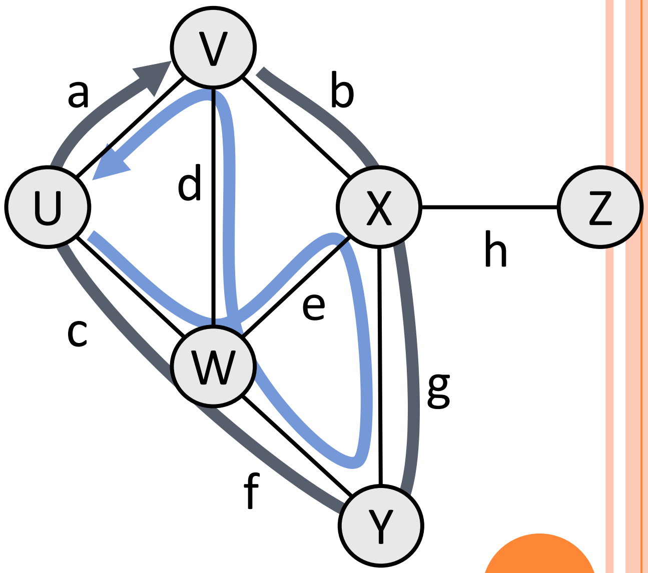
- **strongly connected**: When every vertex has an edge to every other vertex.





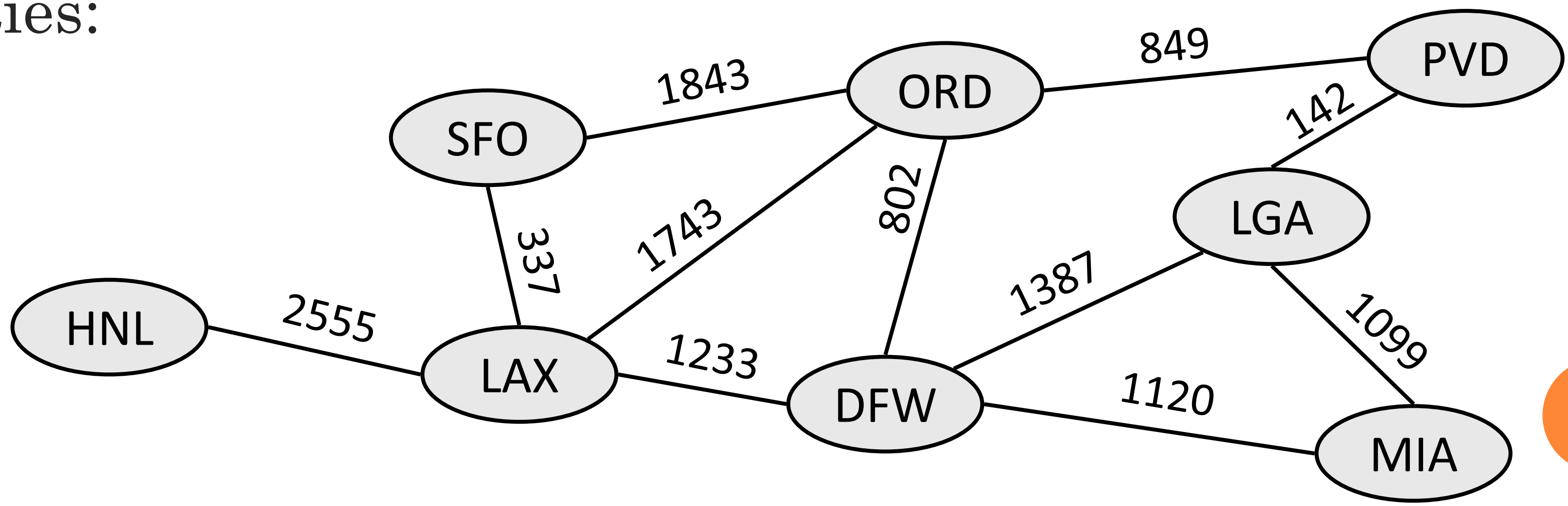
Loops and cycles

- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.
  - **acyclic graph**: One that does not contain any cycles.
- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.



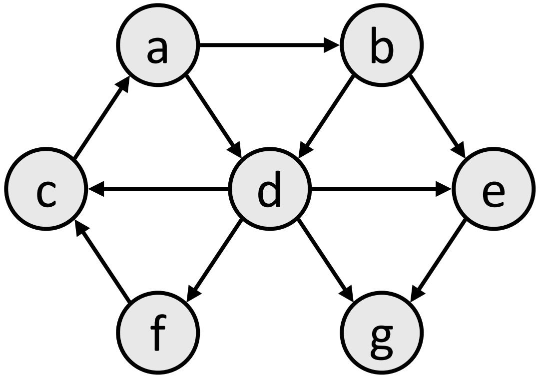
Weighted graphs

- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.
- *example*: graph of airline flights, weighted by miles between cities:



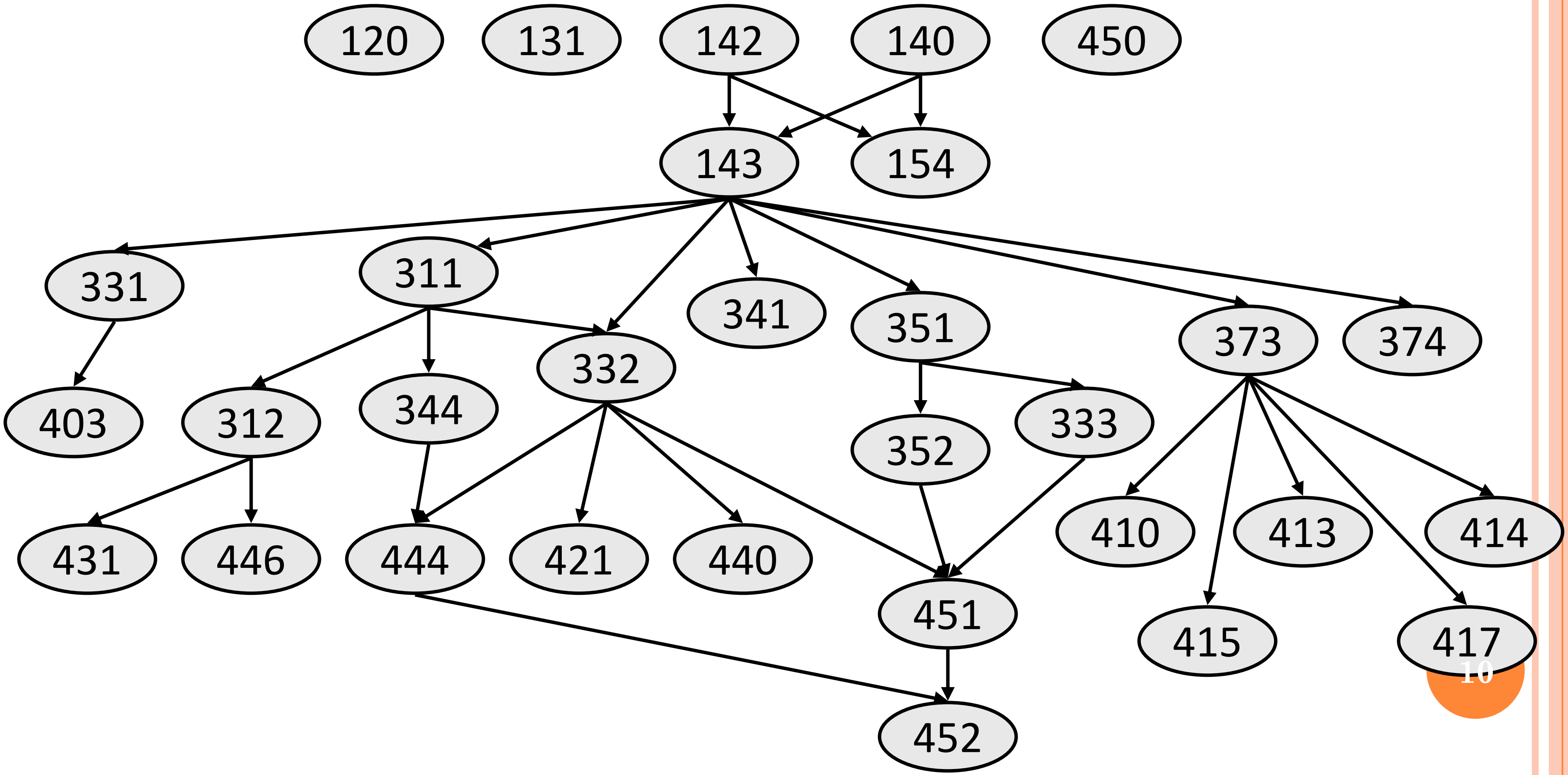
DIRECTED GRAPHS

- **directed graph** ("digraph"): One where edges are *one-way* connections between vertices.
  - If graph is directed, a vertex has a separate in/out degree.
  - A digraph can be weighted or unweighted.
  - Is the graph below connected? Why or why not?



Digraph example

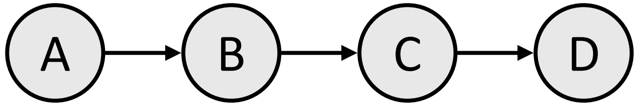
- Vertices= University courses (incomplete list)
- Edge (a, b) = *a* is a prerequisite for *b*

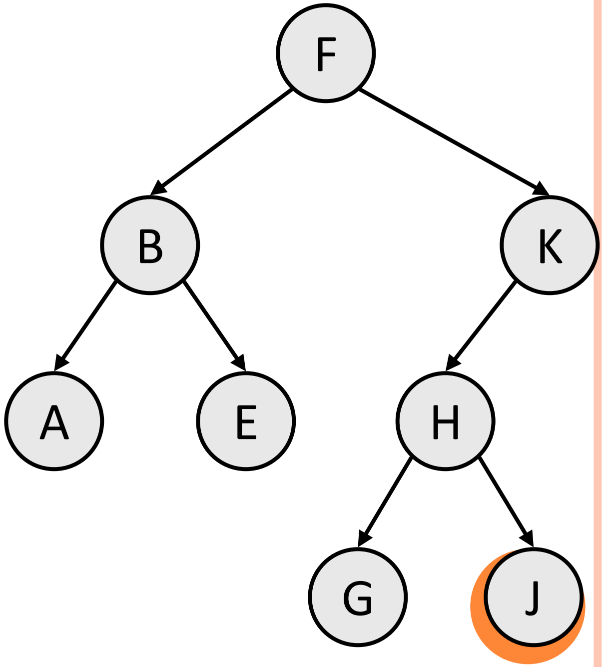


Linked Lists, Trees, Graphs

- **A *binary tree* is a graph with some restrictions:**
  - The tree is an unweighted, directed, acyclic graph (DAG).
  - Each node's in-degree is at most 1, and out-degree is at most 2.
  - There is exactly one path from the root to every node.

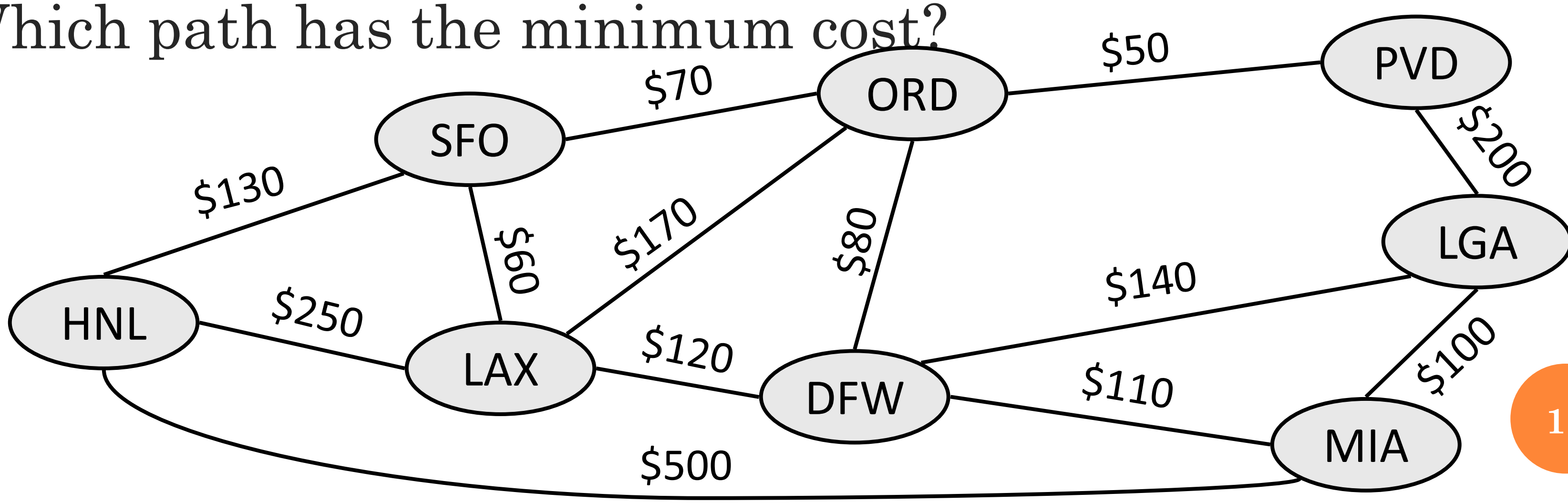
- A *linked list* is also a graph:
  - Unweighted DAG.
  - In/out degree of at most 1 for all nodes.





SEARCHING FOR PATHS

- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).
- What is the shortest path from MIA to SFO? Which path has the minimum cost?



GRAPHS

Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

In either case, we have $|E| = O(V^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$, which implies that $\lg |E| = \Theta(\lg V)$.

(Review CLRS, Appendix B.)

ADJACENCY-MATRIX REPRESENTATION

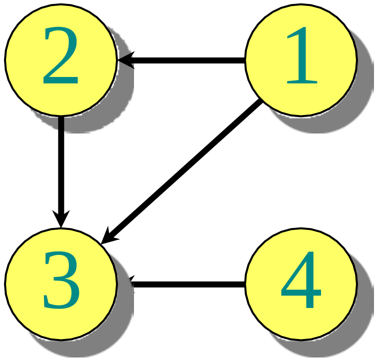
The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in \mathrm{E}, \\ 0 & \text{if } (i, j) \notin \mathrm{E}. \end{cases}$$

ADJACENCY-MATRIX REPRESENTATION

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1\,.\,.\,n, 1\,.\,.\,n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

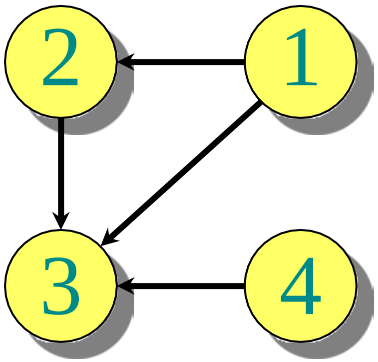


A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(V^2)$ storage $\Rightarrow$ ***dense*** representation.

ADJACENCY-LIST REPRESENTATION

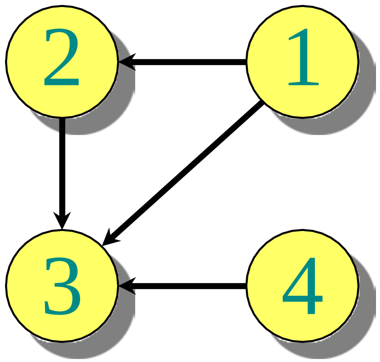
An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

ADJACENCY-LIST REPRESENTATION

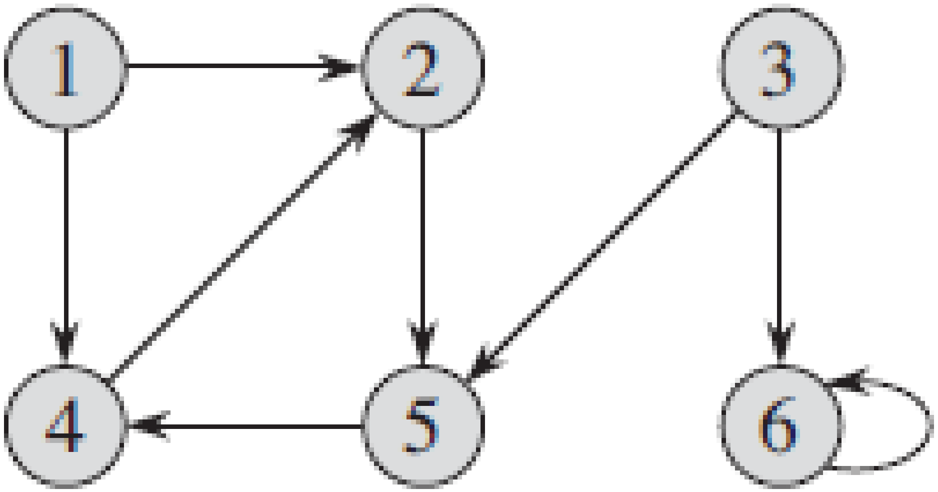
An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

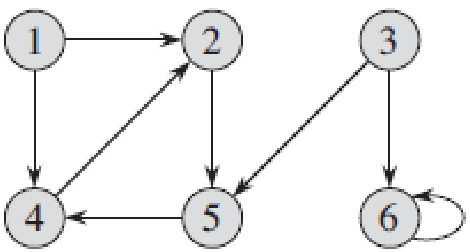
For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Adjacency-list-Matrix -Example

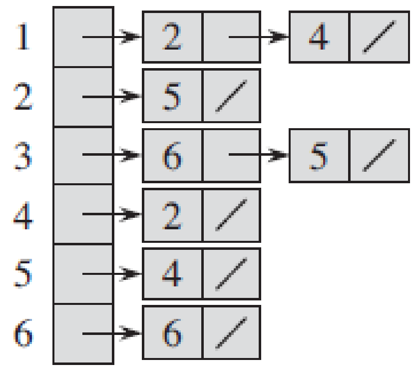


(a)

ADJACENCY-LIST-MATRIX -EXAMPLE



(a)

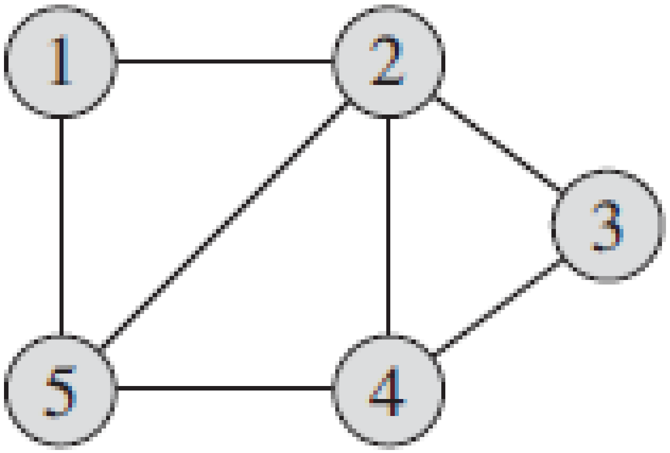


(b)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

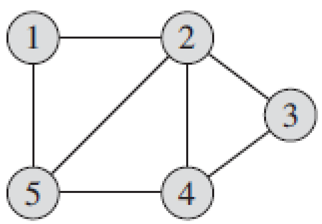
(c)

Adjacency-list-Matrix -Example

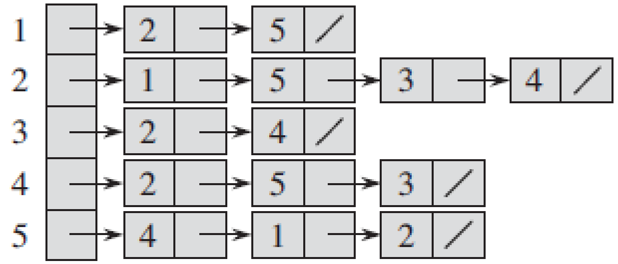


(a)

ADJACENCY-LIST-MATRIX -EXAMPLE



(a)



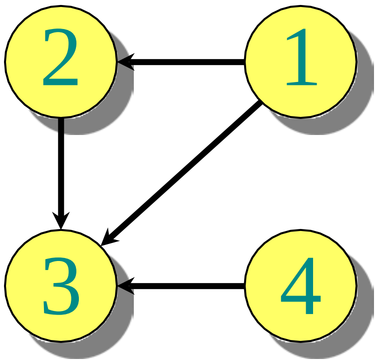
(b)

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

(c)

Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



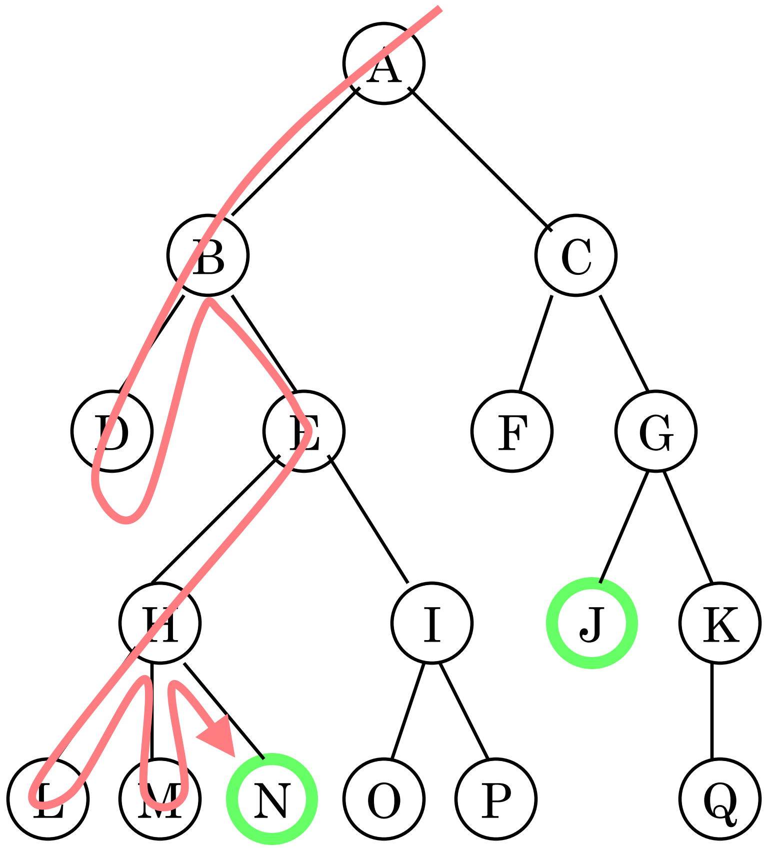
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Handshaking Lemma: $\sum_{v \in V} degree(v) = 2|E|$ for undirected graphs $\Rightarrow$ adjacency lists use $\Theta(V + E)$ storage — a ***sparse*** representation.

Graph Traversals

Depth-first searching

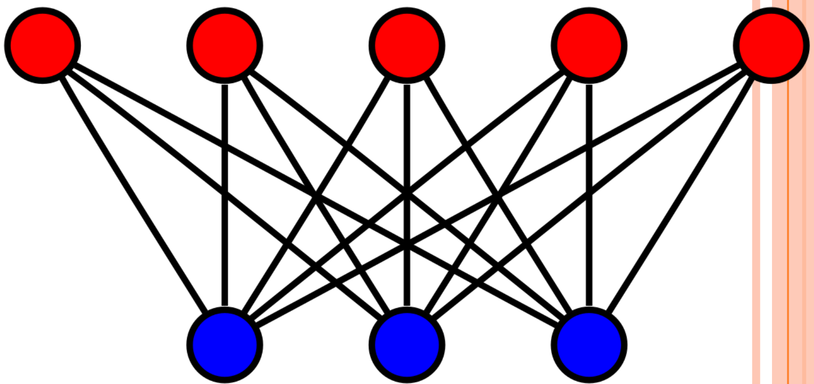


- A **depth-first** search **(DFS)** explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

Depth First Search

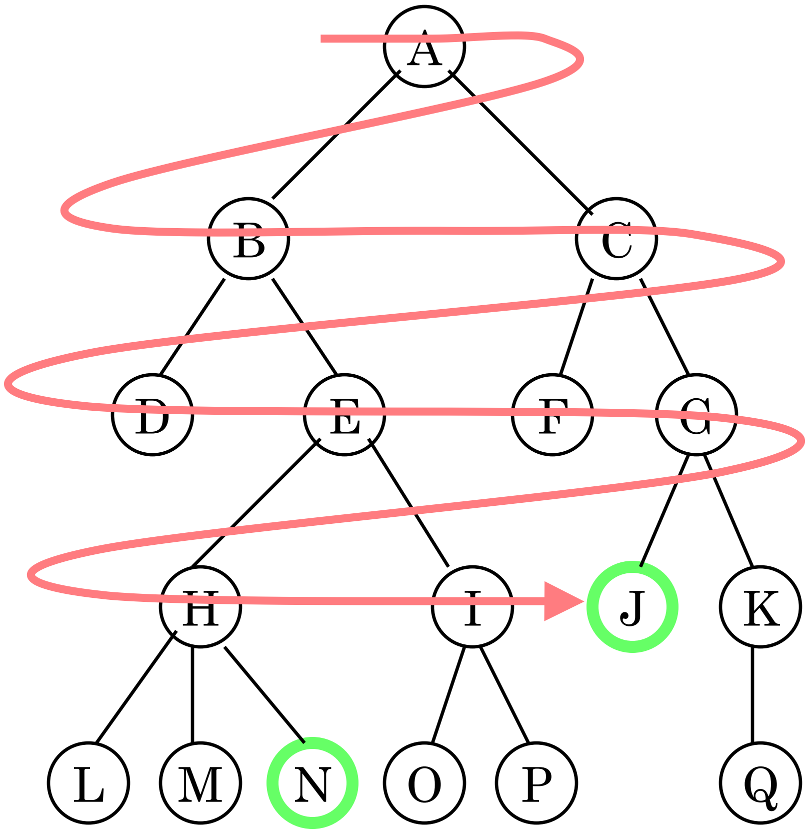
Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

- **Detecting cycle in a graph**
- **Path Finding**
- **Topological Sorting**
- **To test if a graph is bipartite**
- **Finding Strongly Connected Components of a graph**
- **Solving puzzles with only one solution**



Bipartite Graph

Breadth-first searching



- A **breadth-first** search **(BFS)** explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

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 that is reachable from s.**
- It computes the distance (smallest number of edges) from s to each reachable vertex.
- It also produces a "breadth-first tree" with root s that contains all reachable vertices.
- For any vertex reachable from s, the simple path in the breadth-first tree from s to corresponds to a "shortest path" from s to in G, that is, a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.

Breadth-first searching

- **Shortest Path and Minimum Spanning Tree for unweighted graph**
- **Peer to Peer Networks**
- **Crawlers in Search Engines**
- **Social Networking Websites**
- **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
- **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- **In Garbage Collection**
- **Cycle detection in undirected graph:**

Breadth-first searching

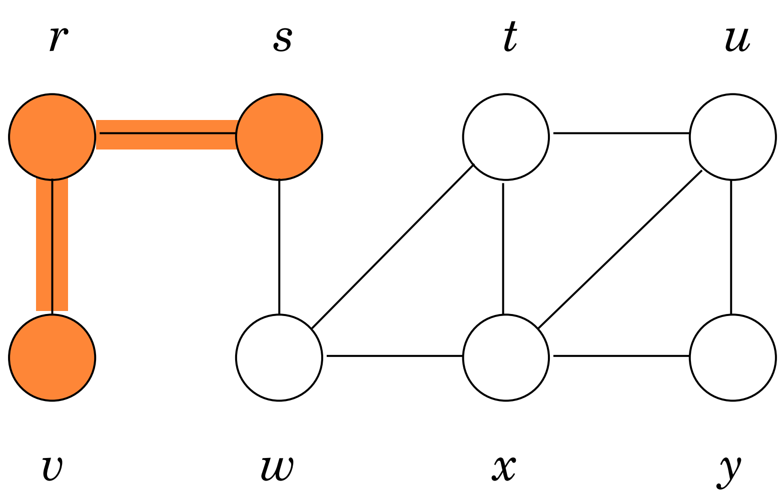
- **To test if a graph is Bipartite**
- **Path Finding**
- **Finding all nodes within one connected component.**

Breadth-first search

- Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- That is, the algorithm **discovers** all vertices at distance **k from s** **before discovering** any vertices at distance **k + 1**.

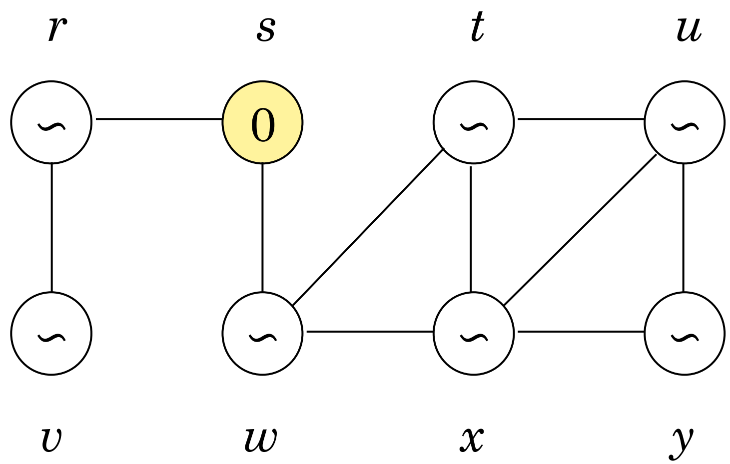
Breadth-first search

- ***Distance***
  - Distance from u to v
    - The number of edges in the shortest path from u to v.
    - The distance from **s to v is 2**.



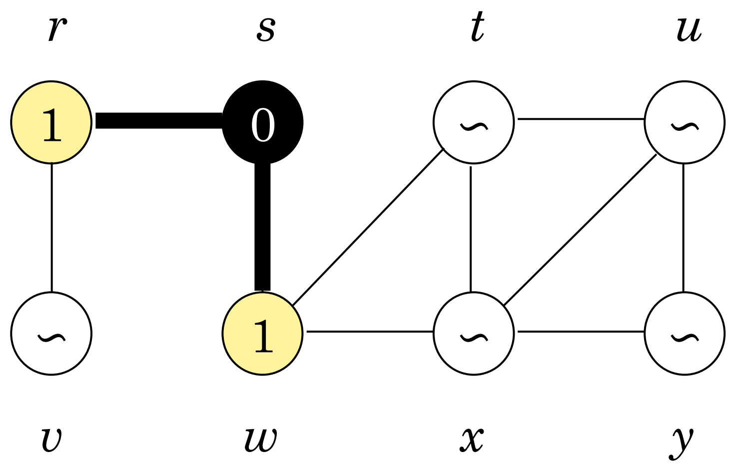
Breadth-first search

- **Breadth-first search**
  - Given a graph $G = (V, E)$ and a ***source*** vertex s, it explores the edges of G to "discover" every reachable vertex from s.
  - **It discovers vertices in the increasing order of distance from the source**. It first discovers all vertices at distance 1, then 2, and etc.



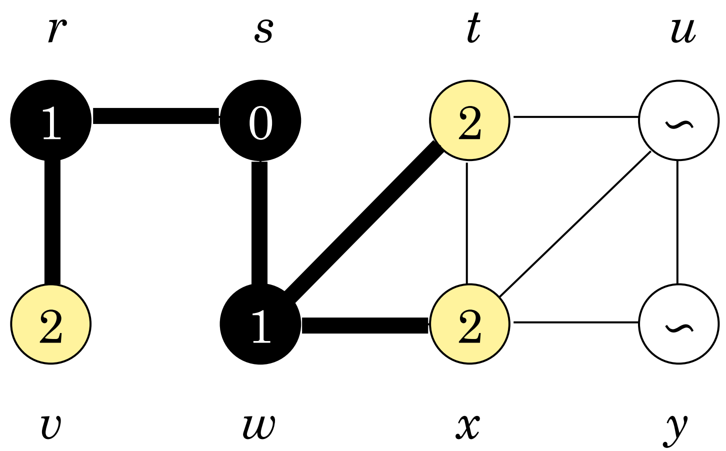
BREADTH-FIRST SEARCH

- **Breadth-first search**
  - Given a graph $G = (V, E)$ and a ***source*** vertex s, it explores the edges of G to "discover" every reachable vertex from s.
  - It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.



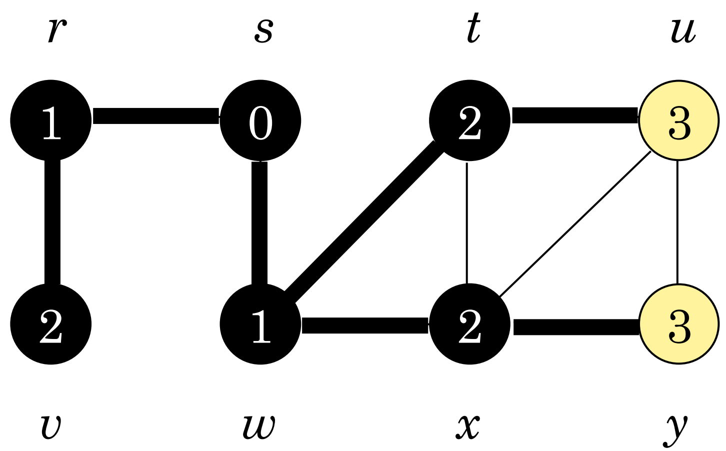
BREADTH-FIRST SEARCH

- **Breadth-first search**
  - Given a graph $G = (V, E)$ and a ***source*** vertex s, it explores the edges of G to "discover" every reachable vertex from s.
  - It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.



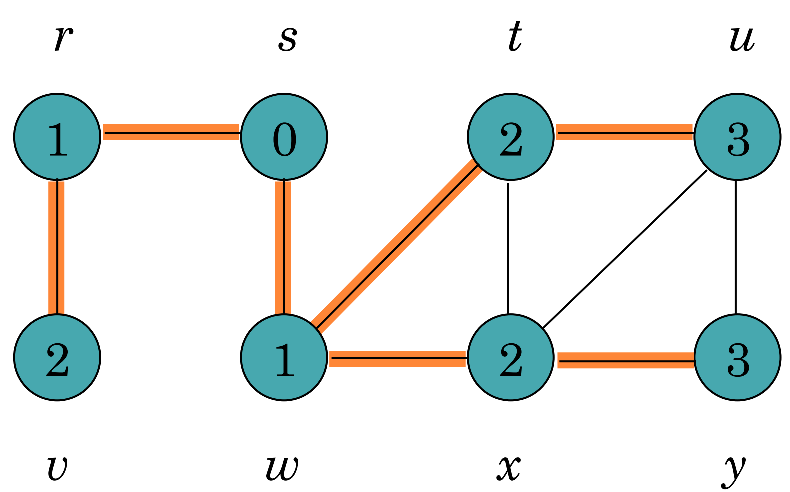
Breadth-first search

- **Breadth-first search**
  - Given a graph $G = (V, E)$ and a ***source*** vertex s, it explores the edges of G to "discover" every reachable vertex from s.
  - It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.



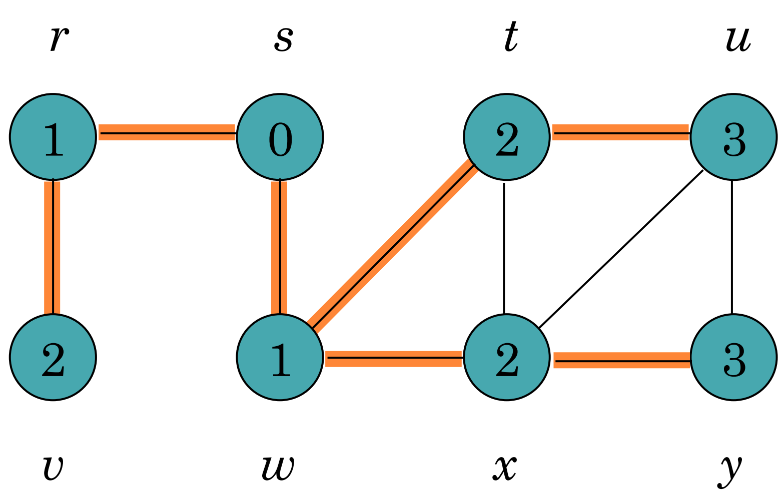
Breadth-first search

- **Breadth-first search**
  - It also computes
    - the distance of vertices from the source: $d[u] = 3$
    - the predecessor of vertices: $\pi[u] = t$



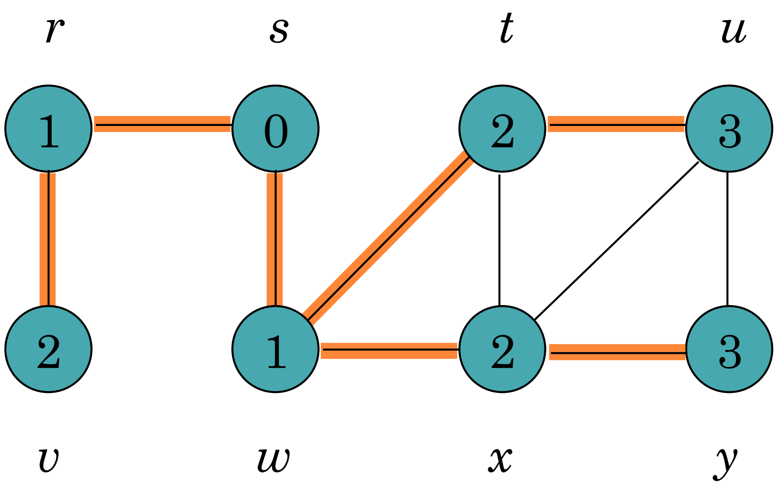
Breadth-first search

- The ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$,
  - $V_\pi = \{v \in V: \pi[v] \neq \text{NIL}\} \cup \{\, s\}$
  - $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$.
- V_π consists of the vertices reachable from s and, for all $v \in V_\pi$ *and* the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G



BREADTH-FIRST SEARCH

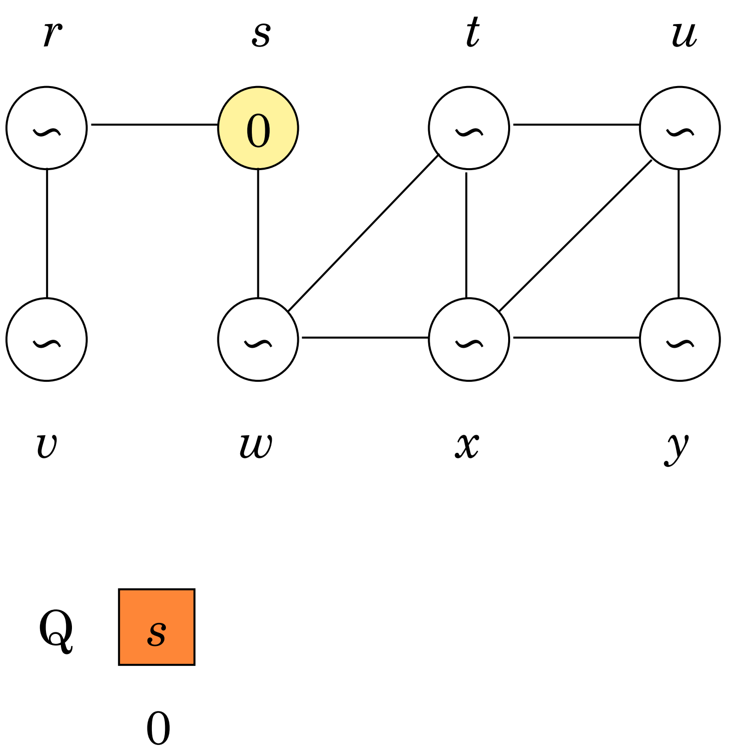
- The predecessor subgraph G_π is a ***breadth-first tree.***
  - since it is connected and $|E_\pi| = |V_\pi| - 1$.
  - The edges in E_π are called ***tree edges***.

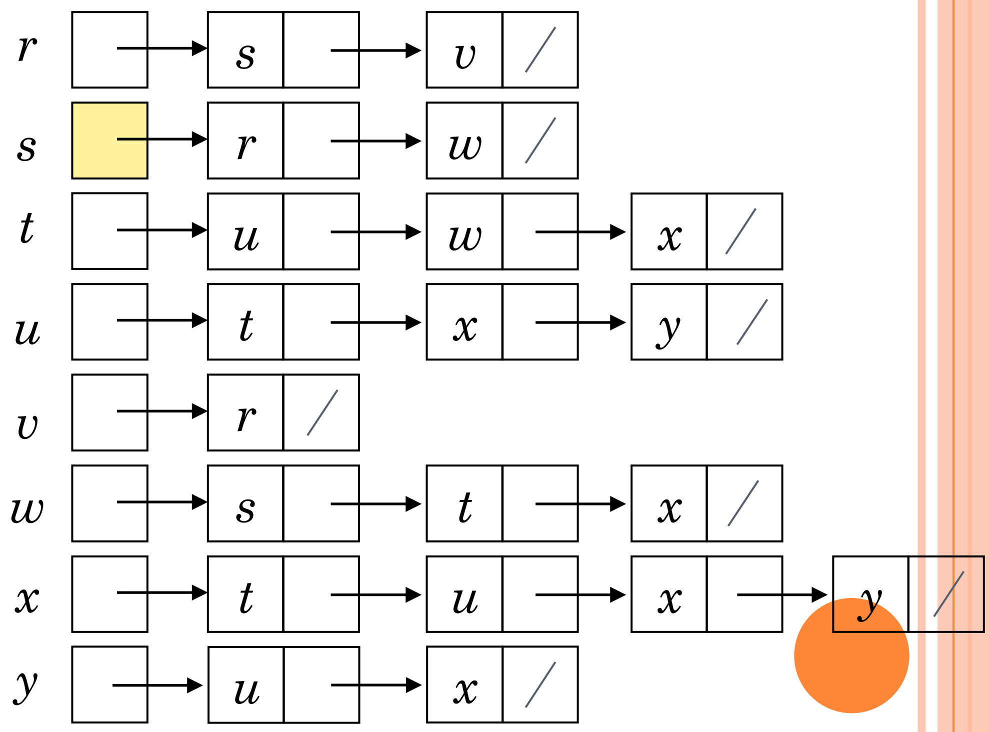


Breadth-first search

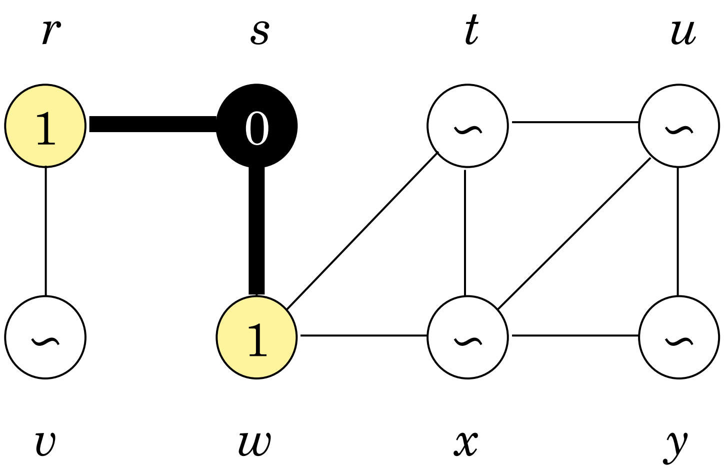
```
BFS(G, s)
for each vertex u  V [G] - {s}
      do color[u] ← WHITE
          d[u] ← ∞
          π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← Ø
ENQUEUE(Q, s)
while Q ≠ Ø
      do u ← DEQUEUE(Q)
          for each v ∈ Adj[u]
               do if color[v] = WHITE
                    then color[v]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

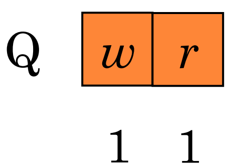
Breadth-first search

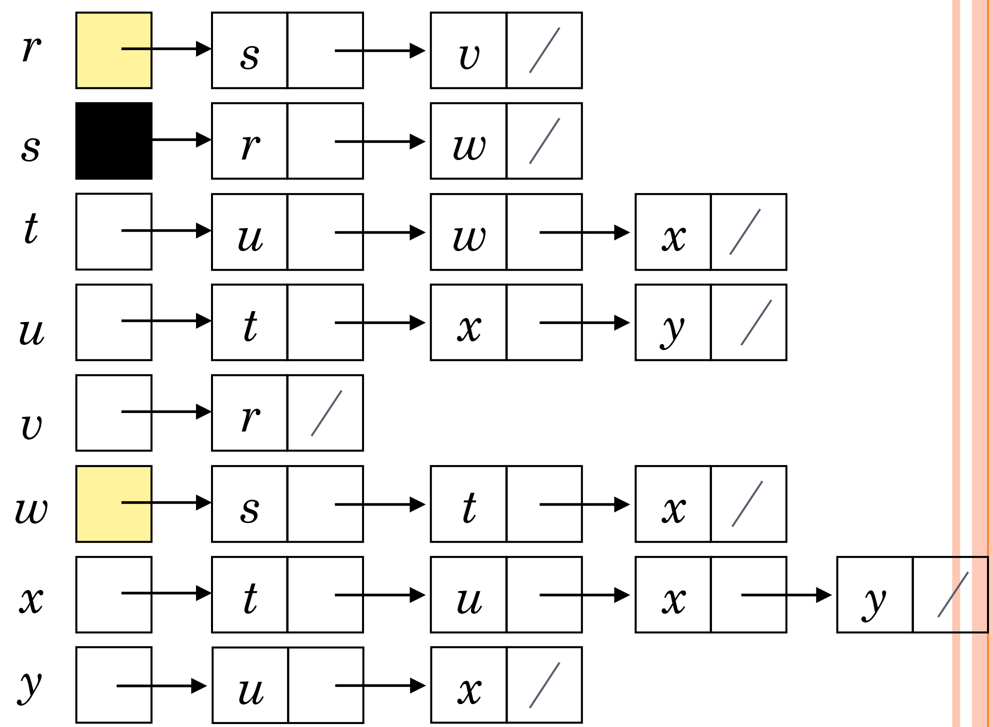




Breadth-first search

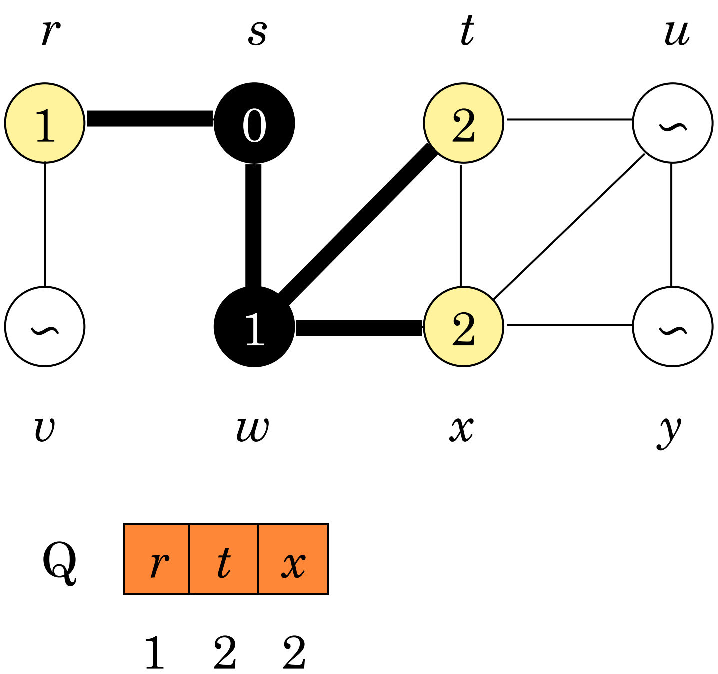


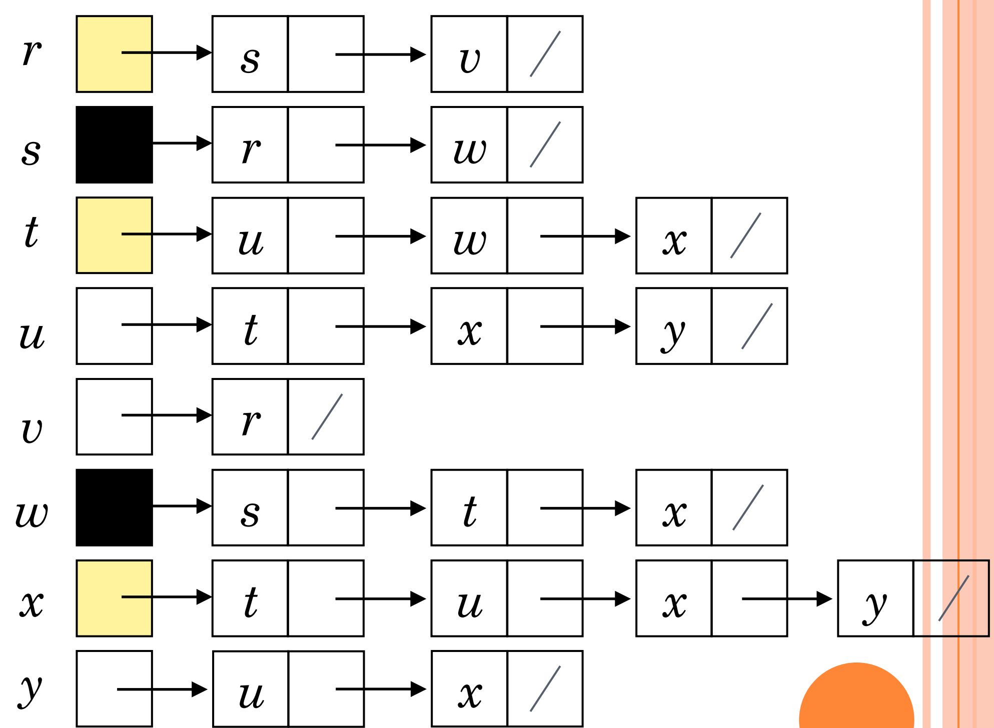




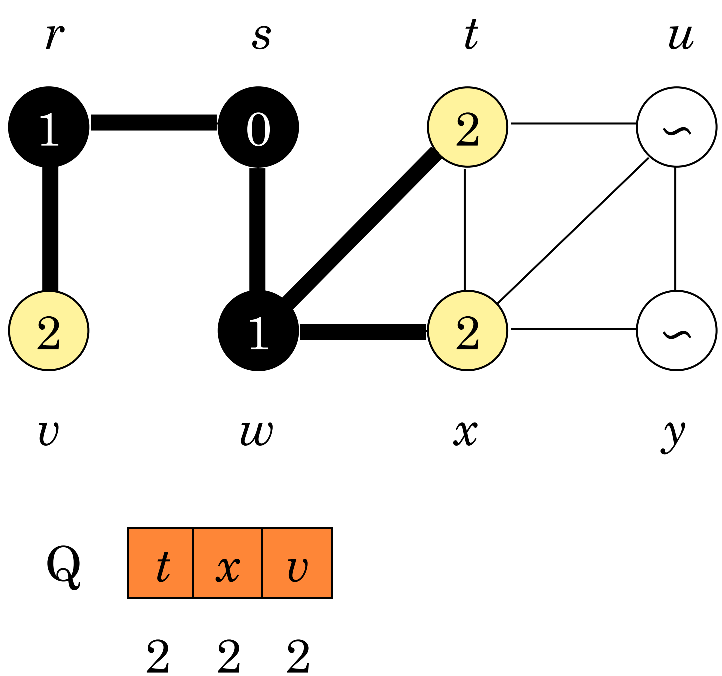
- white: not discovered (not entered the Q)
- gray: discovered (in the Q)
- black: finished (out of the Q)

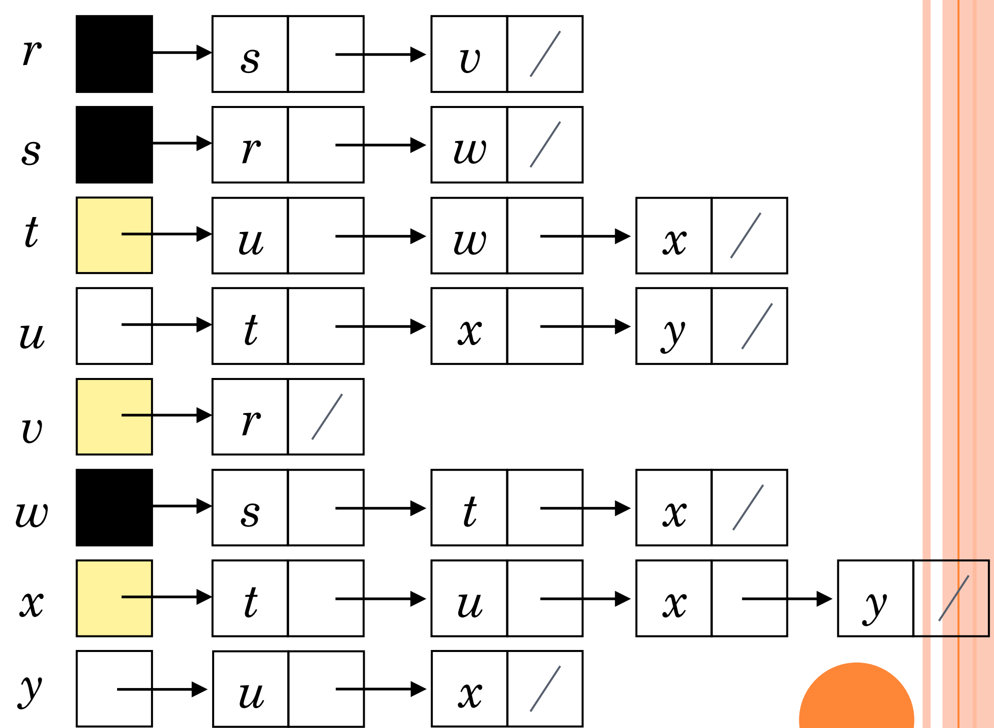
BREADTH-FIRST SEARCH



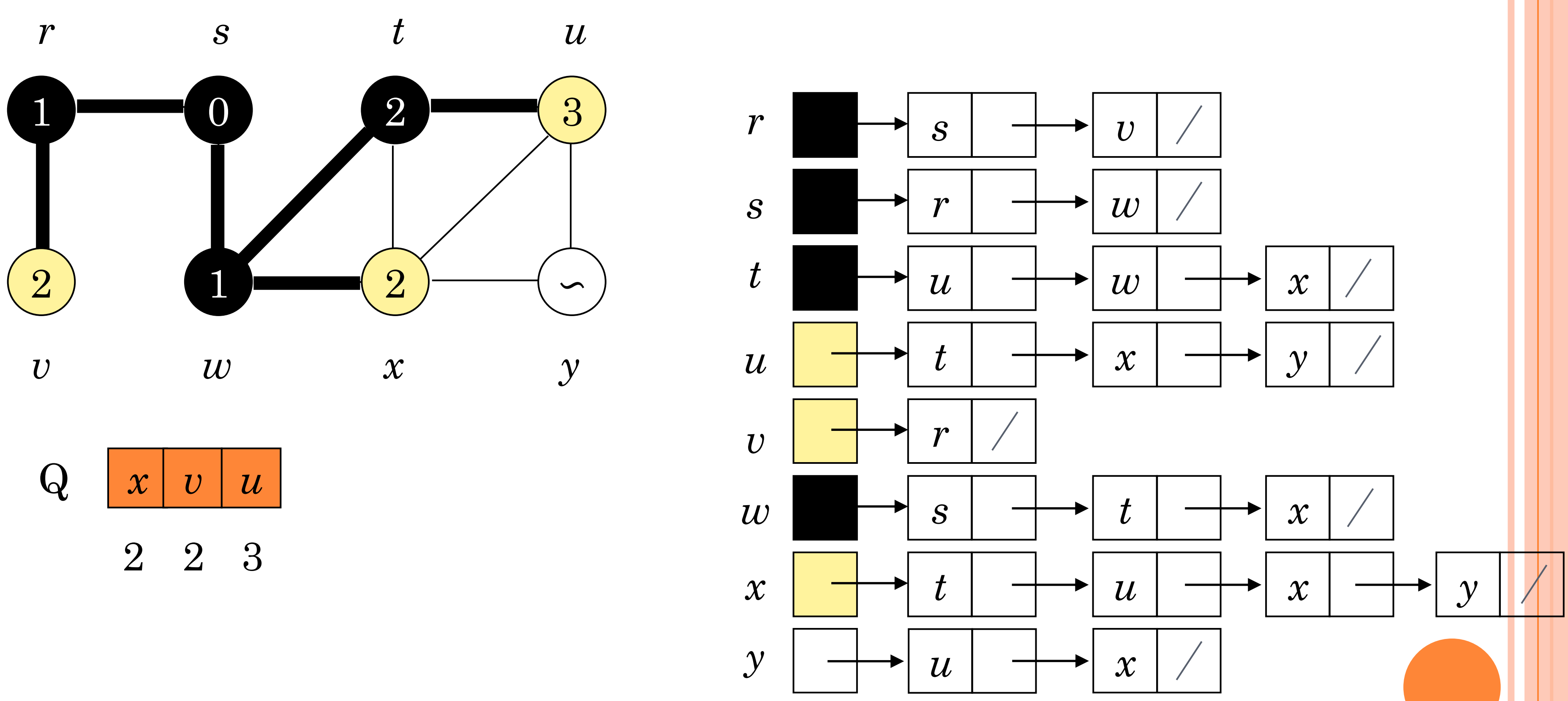


Breadth-first search

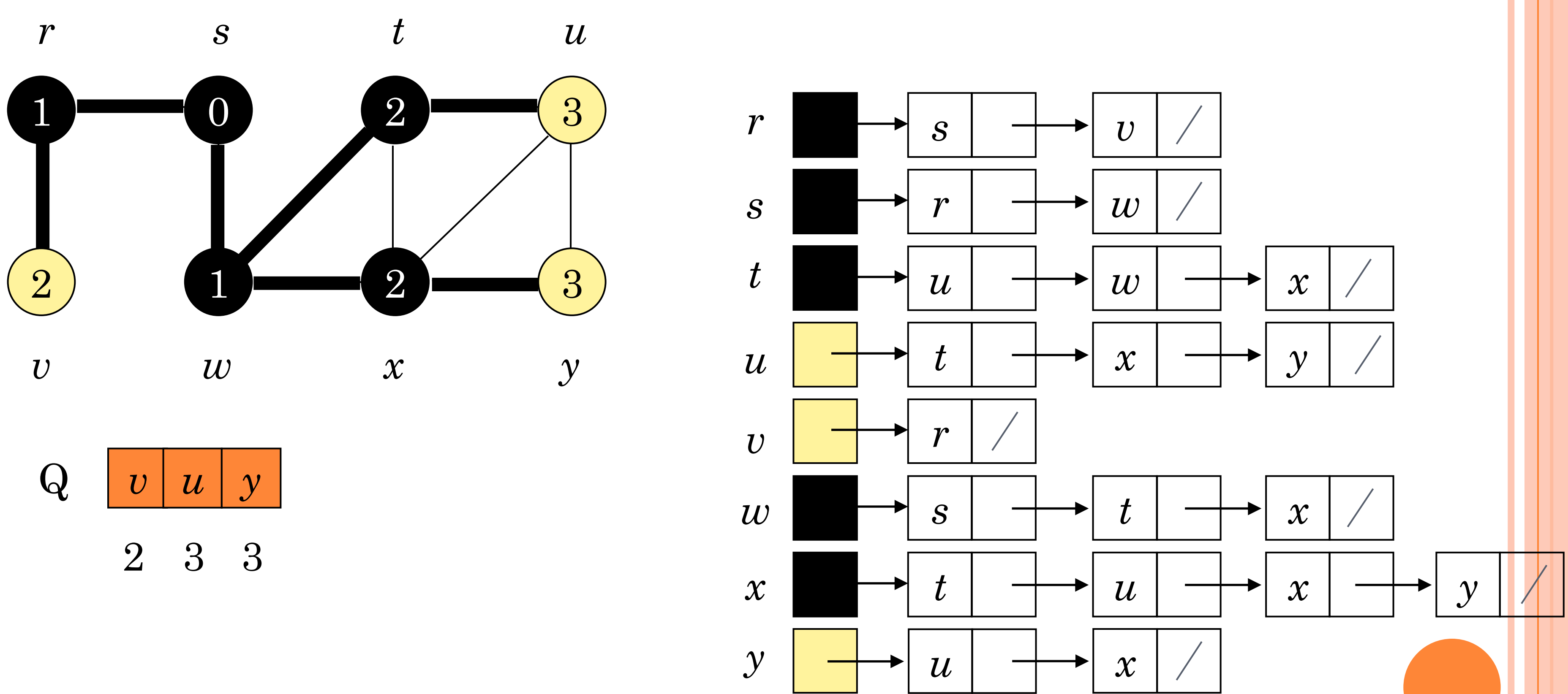




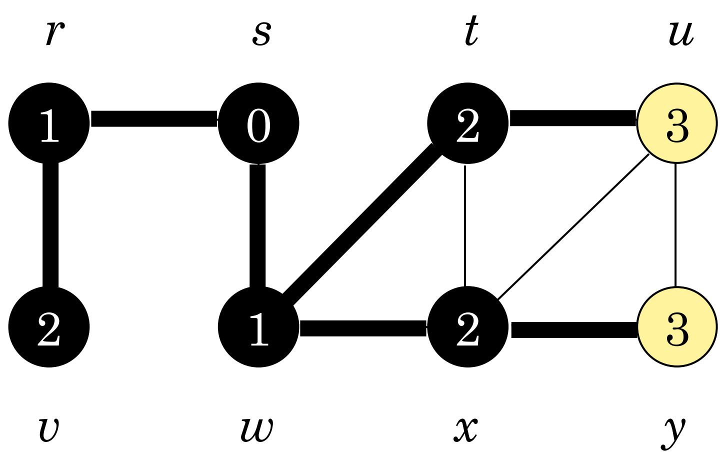
Breadth-first search



Breadth-first search

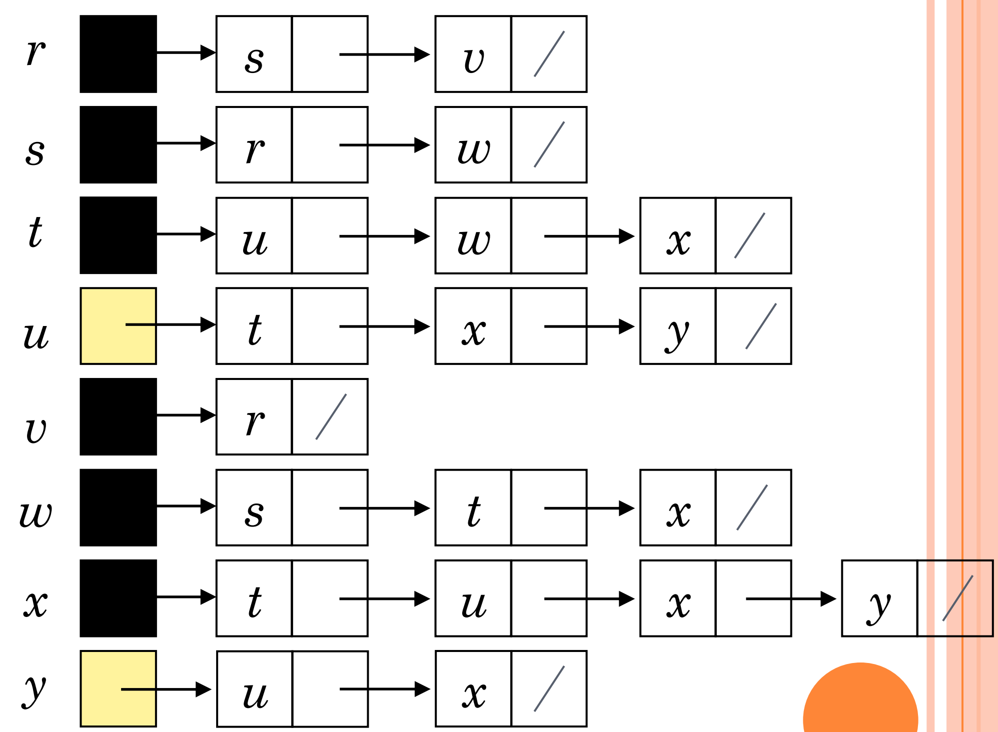


Breadth-first search

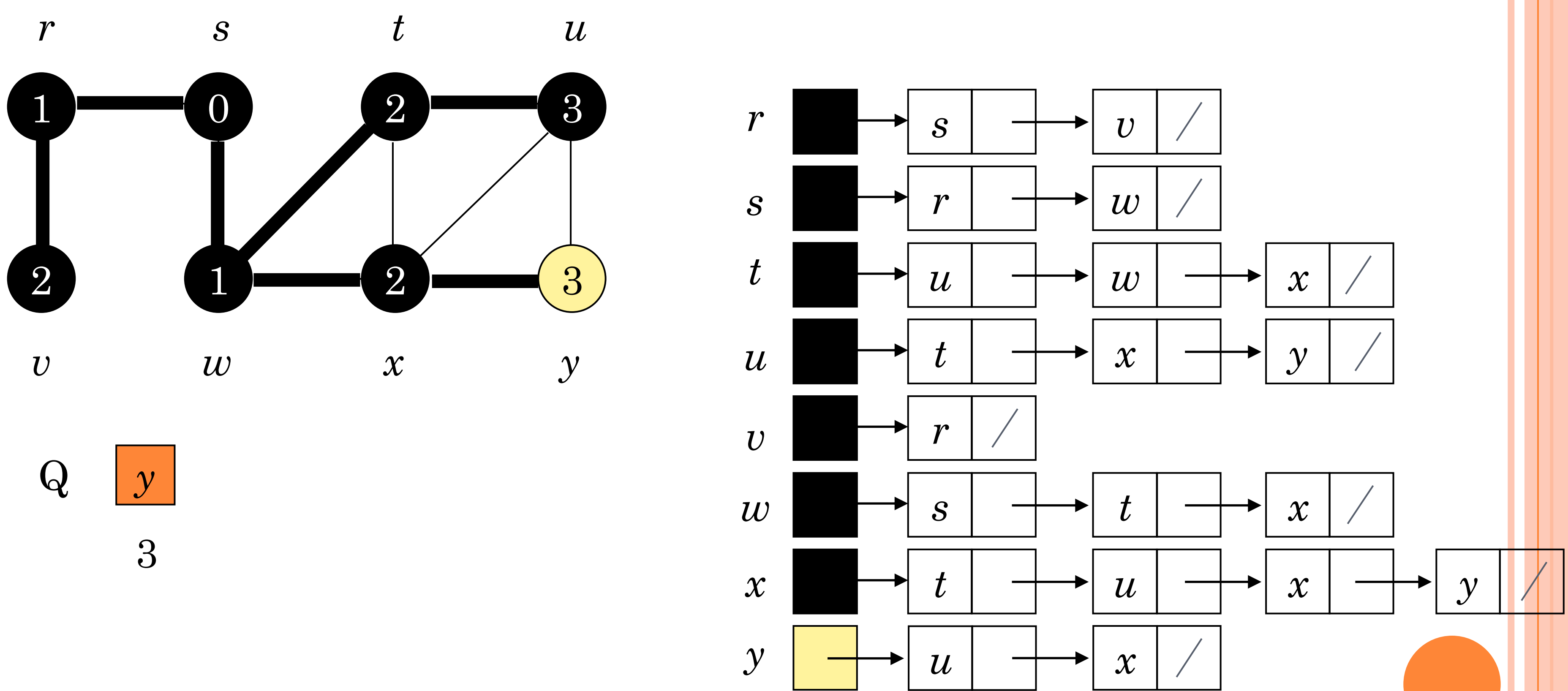


Q | u | y |

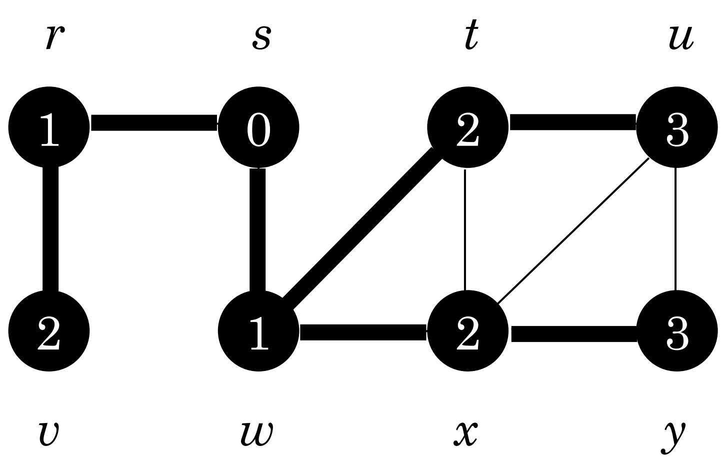
3 3



Breadth-first search

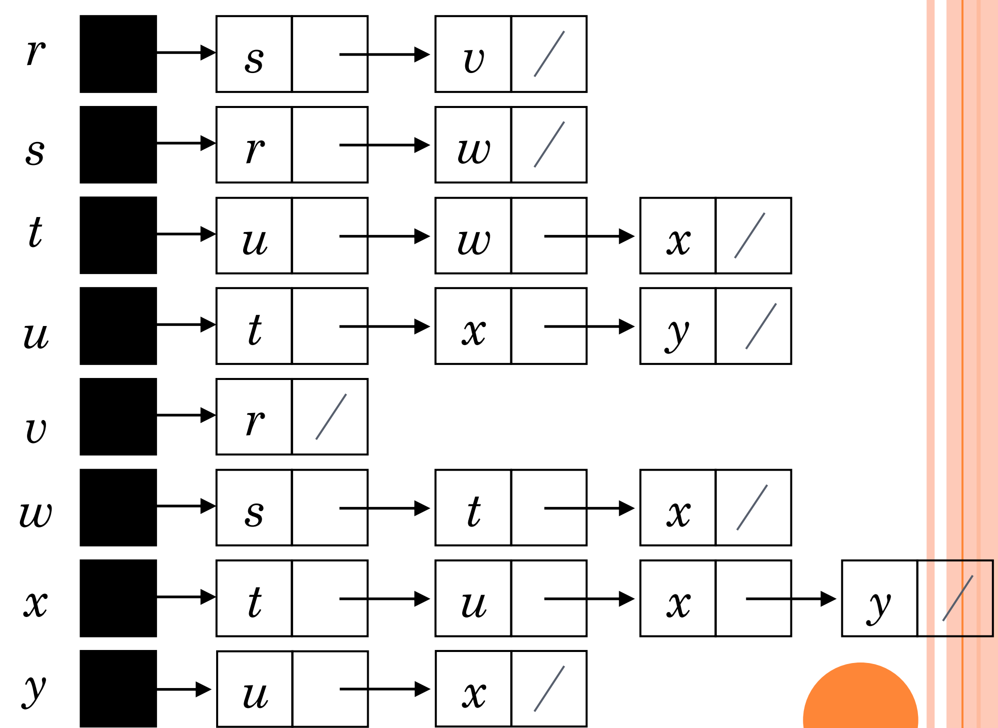


Breadth-first search



Q ∅

3



Breadth-first search

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

O (V)

O (E)

O (V +E)

Analysis

- **Running time**
  - **Initialization**: $\Theta(V)$
  - **Exploring the graph**: $O(E)$
    - An enque operation for an edge exploration.
    - #deque operations = #enque operations
    - An edge is explored at most once.
  - **Overall:** $O(V + E)$

Analysis- Initialization

- 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.

- The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is **O(V).**

Analysis- Exploring the graph

- Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once.
- Since the sum of the lengths of all the adjacency lists is Θ (E), 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).**
- Thus, breadth-first search runs in **time linear** in the size of the adjacency-list representation of G.

Content

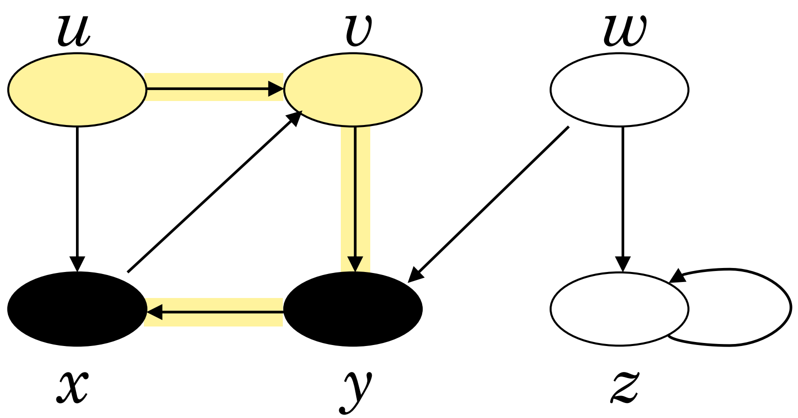
- *Breadth-first search*
- **Depth-first search**

Depth First Search Algorithm

- Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.
- Once all of 's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which 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.

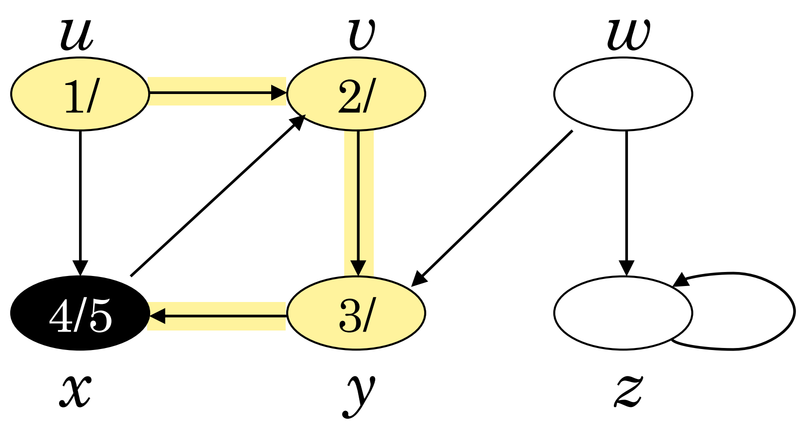
Depth-first search

- ***Colors of vertices***
  - Each vertex is initially ***white***. (not discovered)
  - The vertex is ***grayed*** when it is ***discovered.***
  - The vertex is ***blackened*** when it is ***finished***, that is, when its adjacency list has been examined completely.

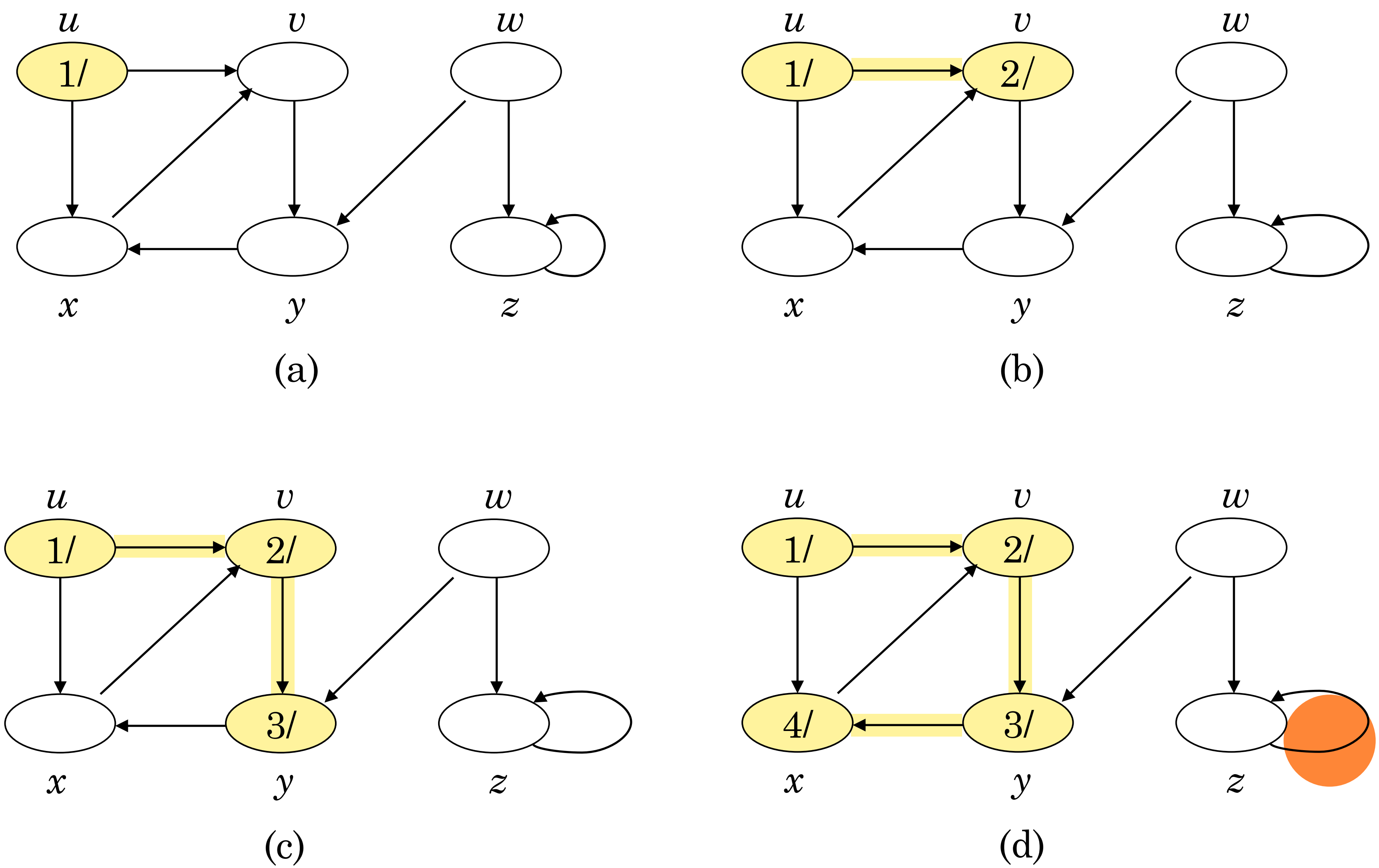


Depth-first search

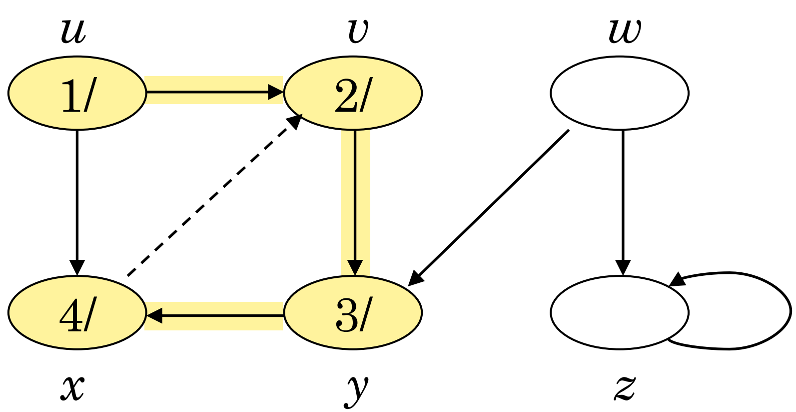
- ***Timestamps***
  - Each vertex v has two timestamps.
    - $d[v]$: ***discovery time*** (when v is grayed)
    - $f\,[v]$: ***finishing time*** (when v is blacken)



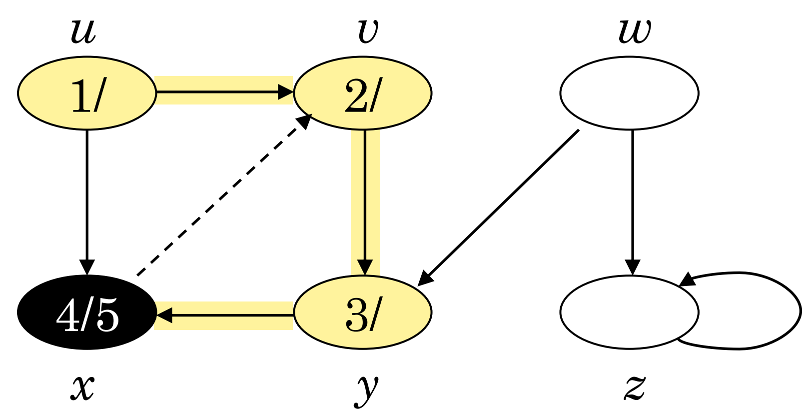
Depth-first search



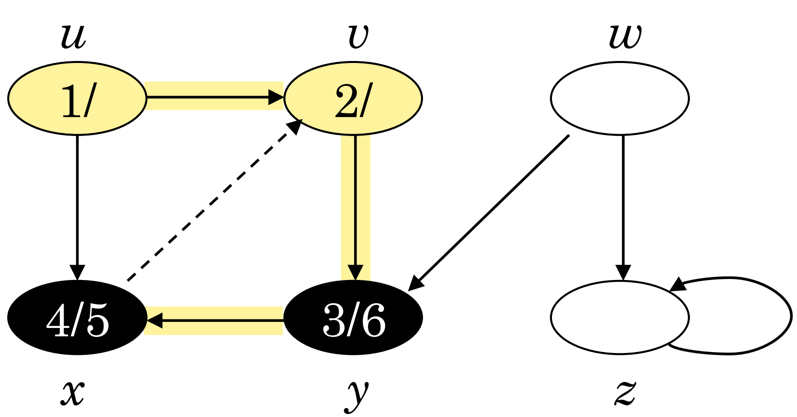
Depth-first search



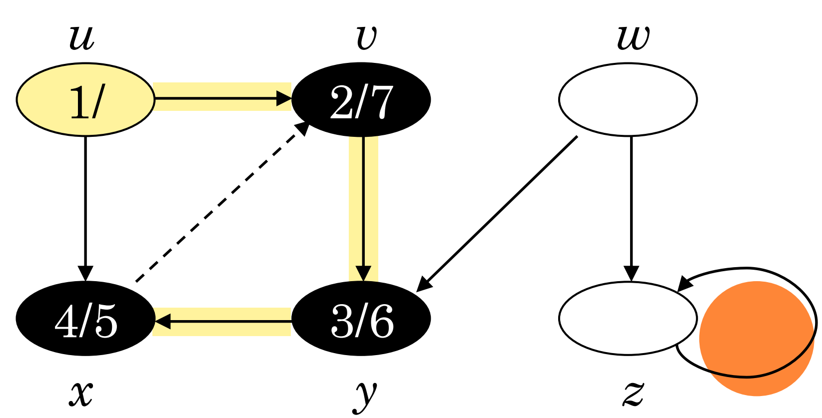
(e)



(f)

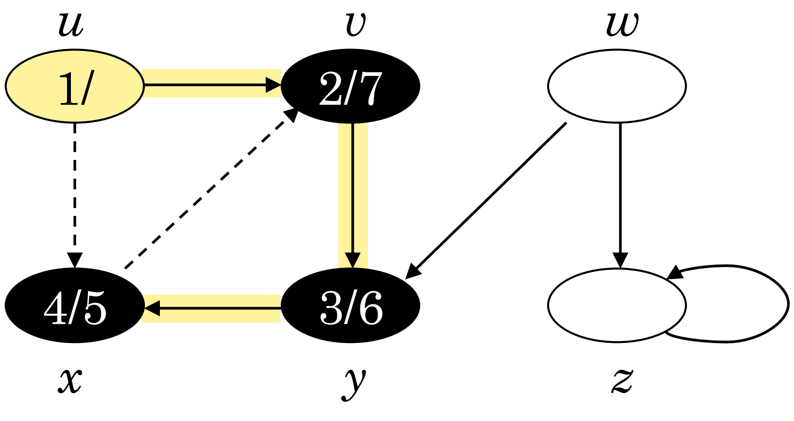


(g)

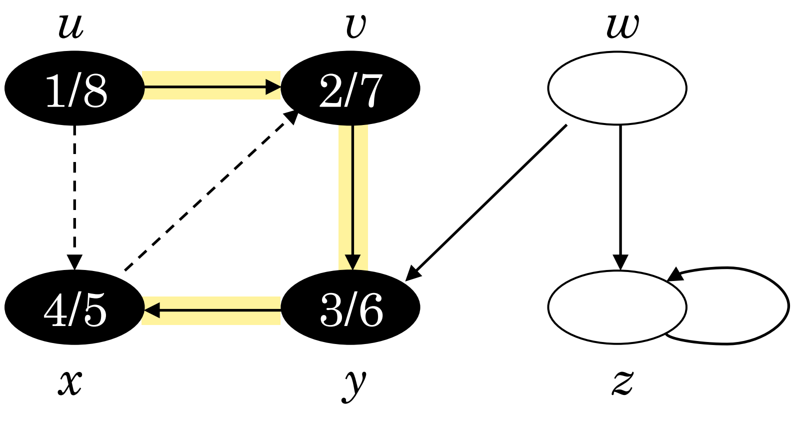


(h)

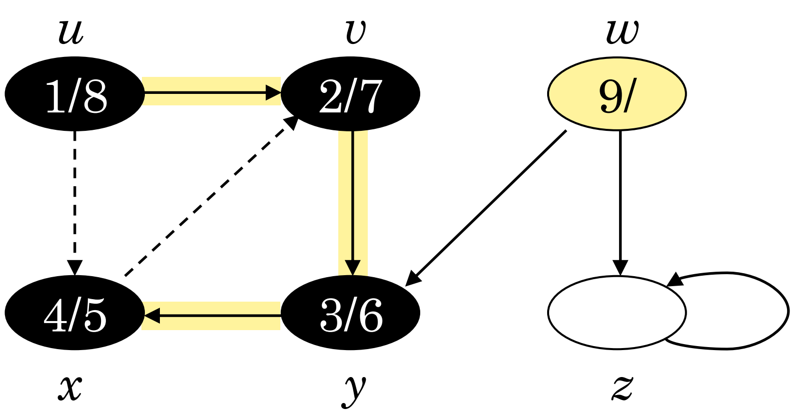
Depth-first search



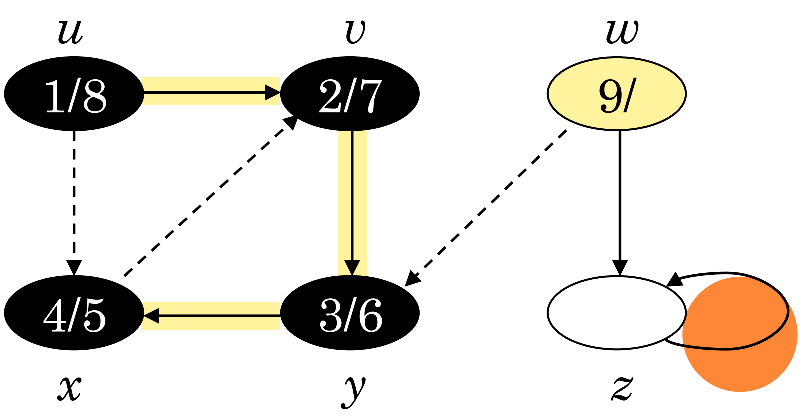
(i)



(j)

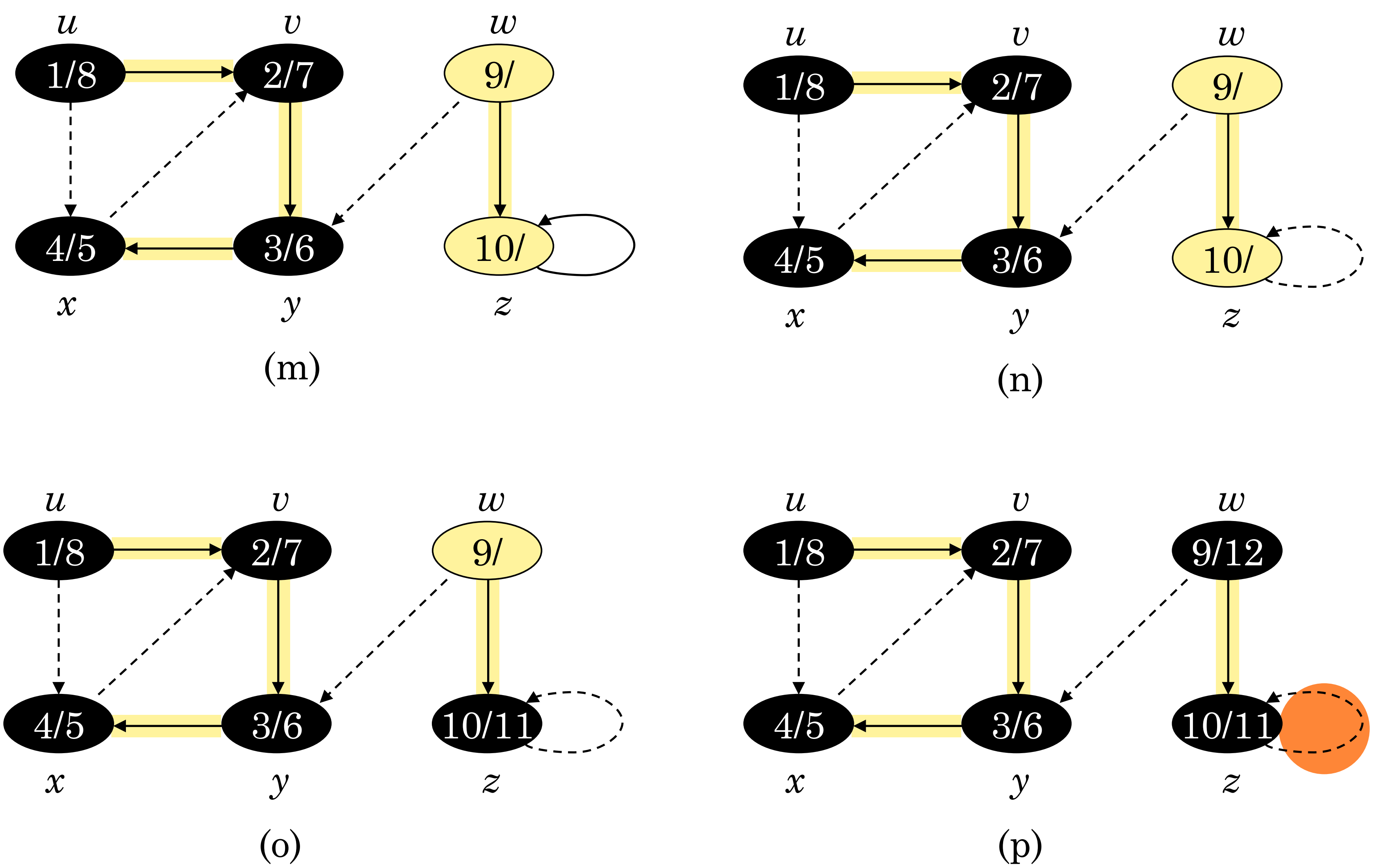


(k)



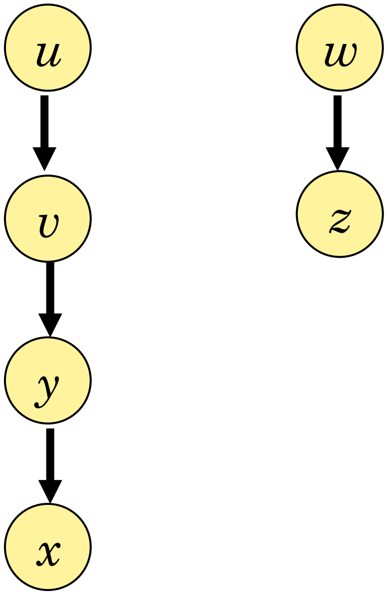
(l)

Depth-first search



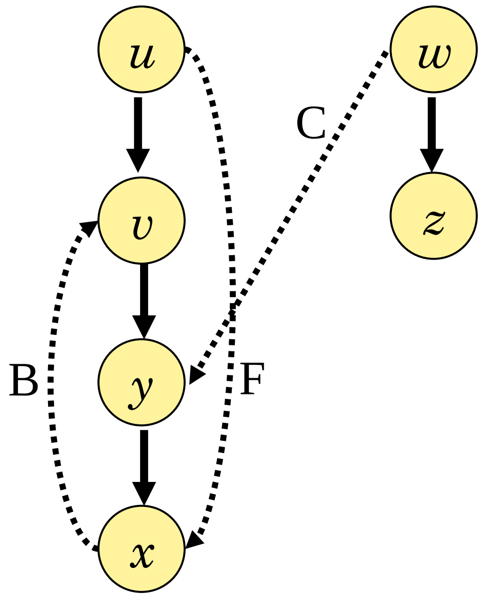
DEPTH-FIRST SEARCH

- The ***predecessor subgraph*** is a ***depth-first forest***.



Depth-first search

- **Classification of edges**
  - ***Tree edges***
  - ***Back edges***
  - ***Forward edges***
  - ***Cross edges***



Depth-first search

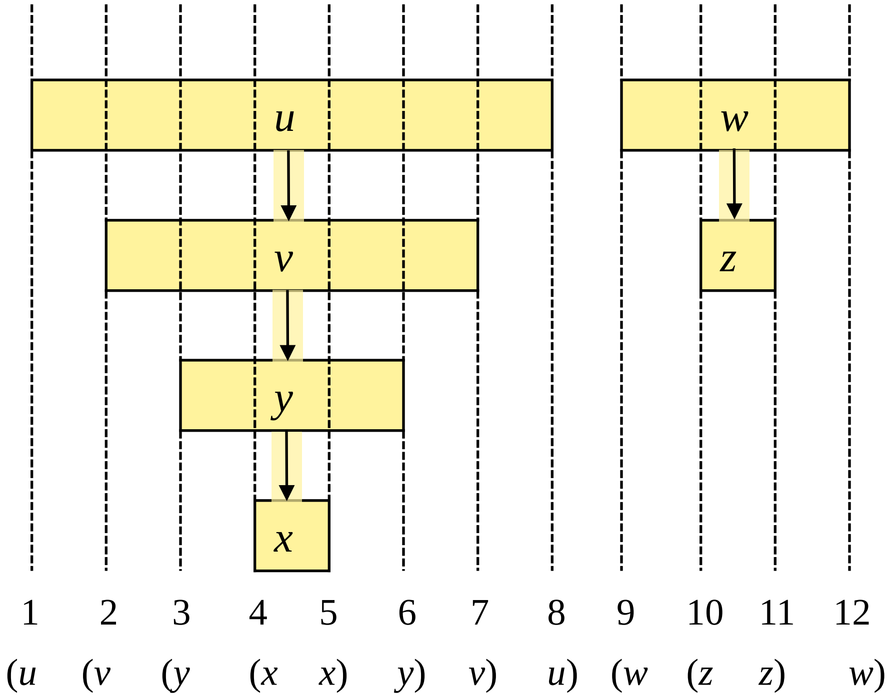
- ***Tree edges*:** Edges in the depth-first forest.
- ***Back edges***: Those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered to be back edges.
- ***Forward edges***: Those edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- ***Cross edges***: 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.

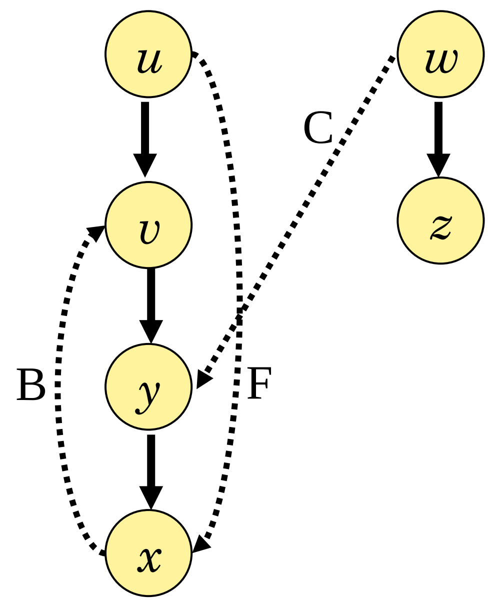
DEPTH-FIRST SEARCH

- Classification by the DFS algorithm
  - Each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored:
    - WHITE indicates a tree edge,
    - GRAY indicates a back edge, and
    - BLACK indicates a forward or cross edge.

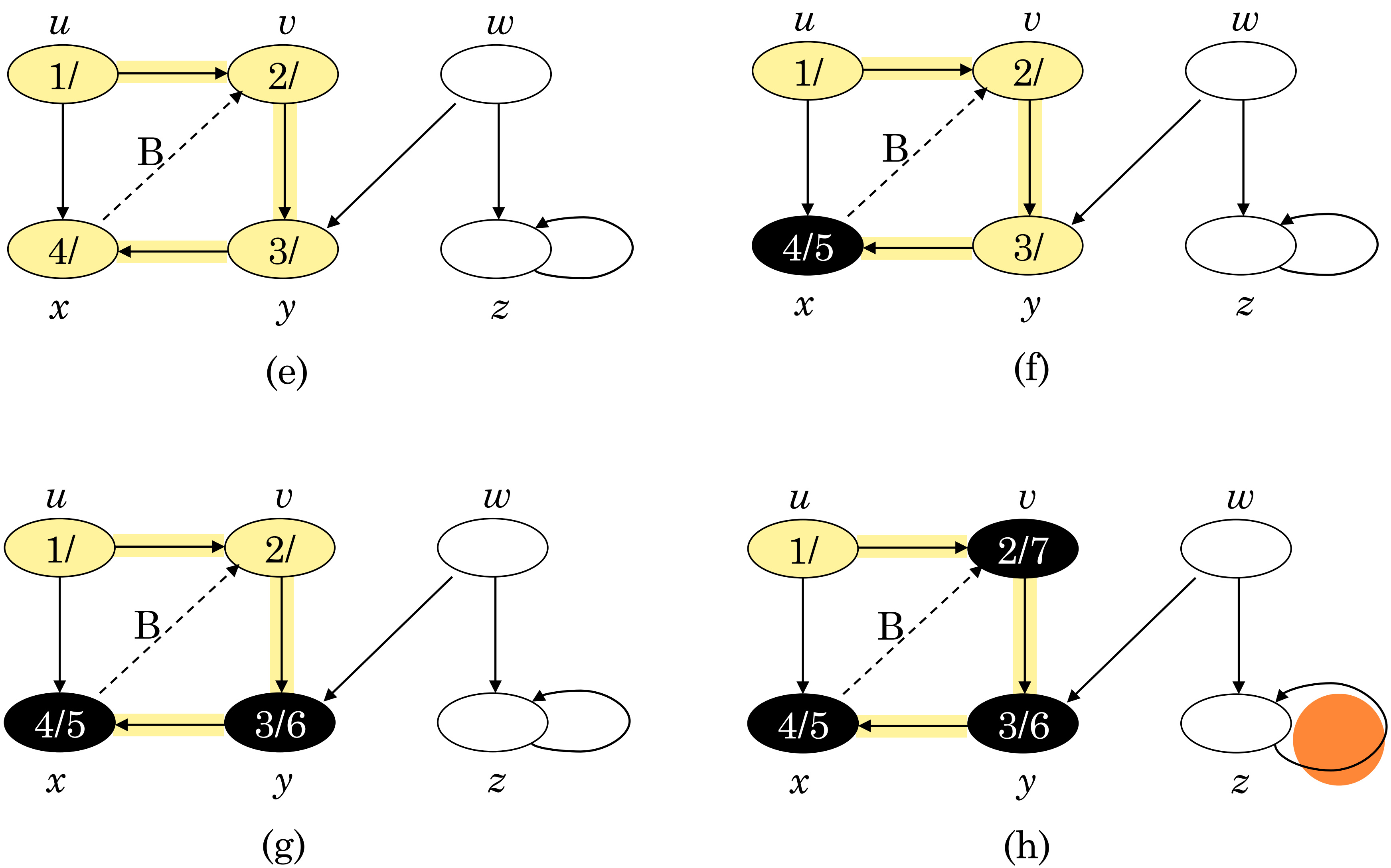
DEPTH-FIRST SEARCH (*PARENTHESIS STRUCTURE*)

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 finishing makes a well-formed expression in the sense that the parentheses are properly nested.

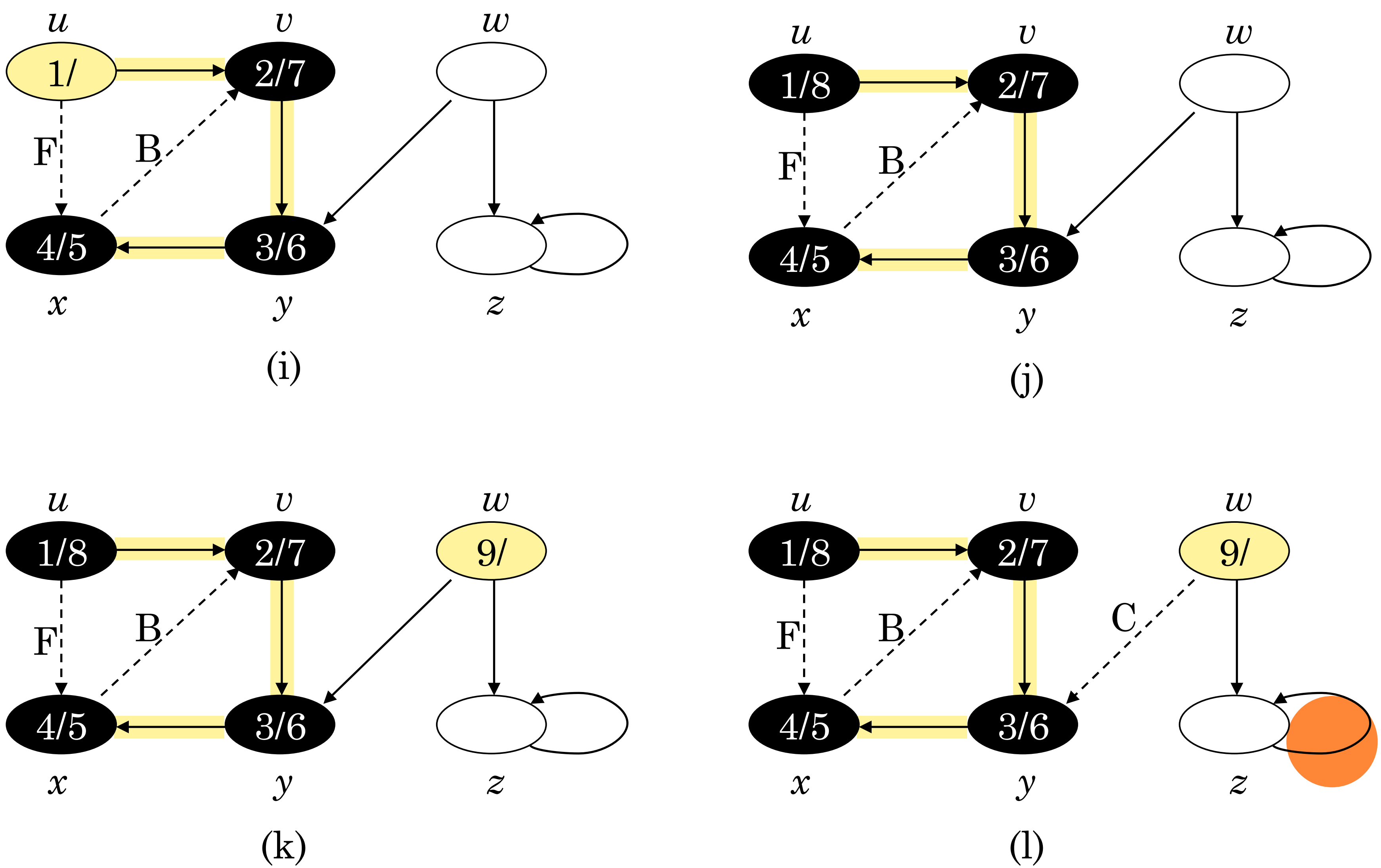




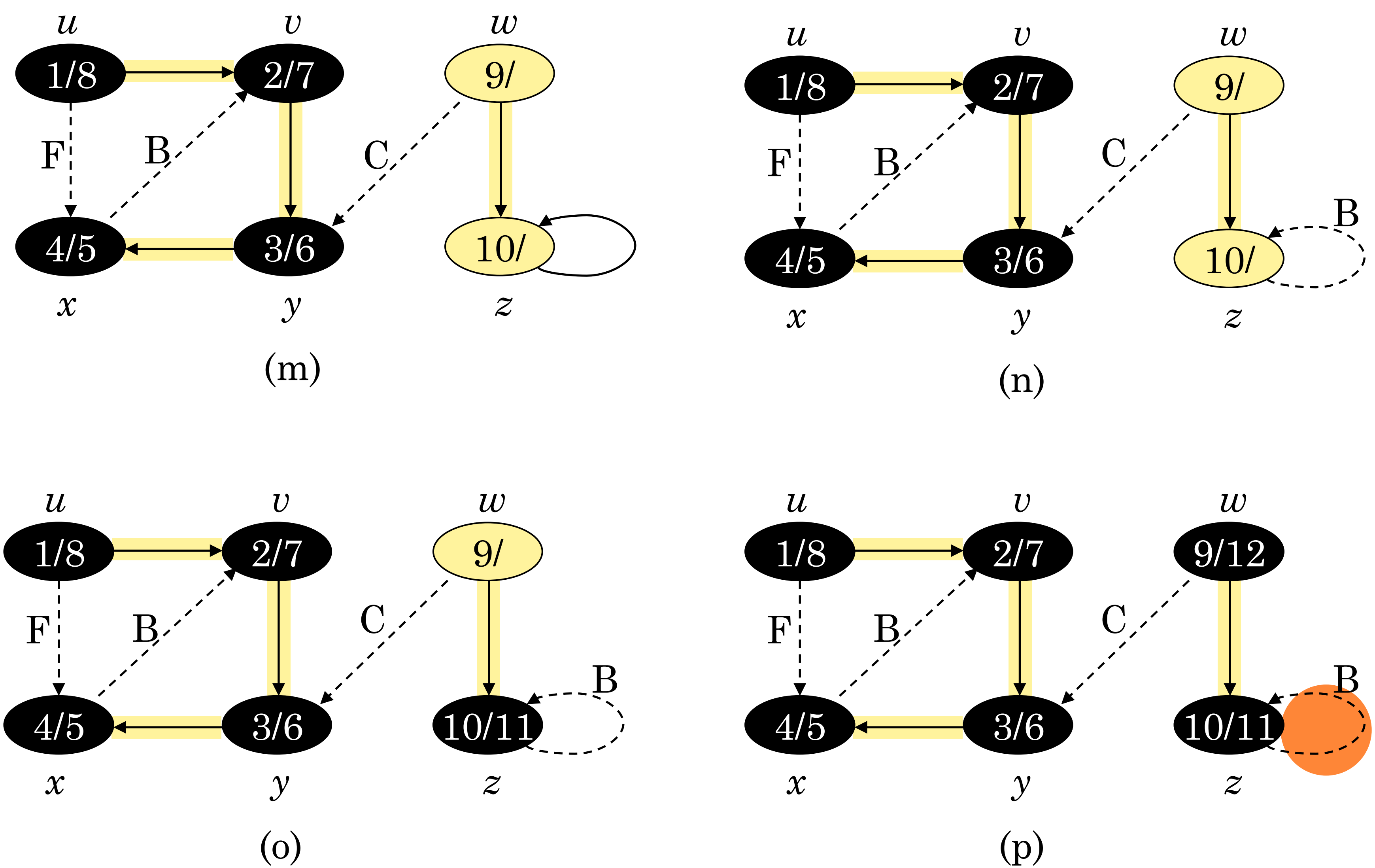
Depth-first search



Depth-first search



Depth-first search



Depth-first search

```
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, u)
```

Θ (V)

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

Θ (E)

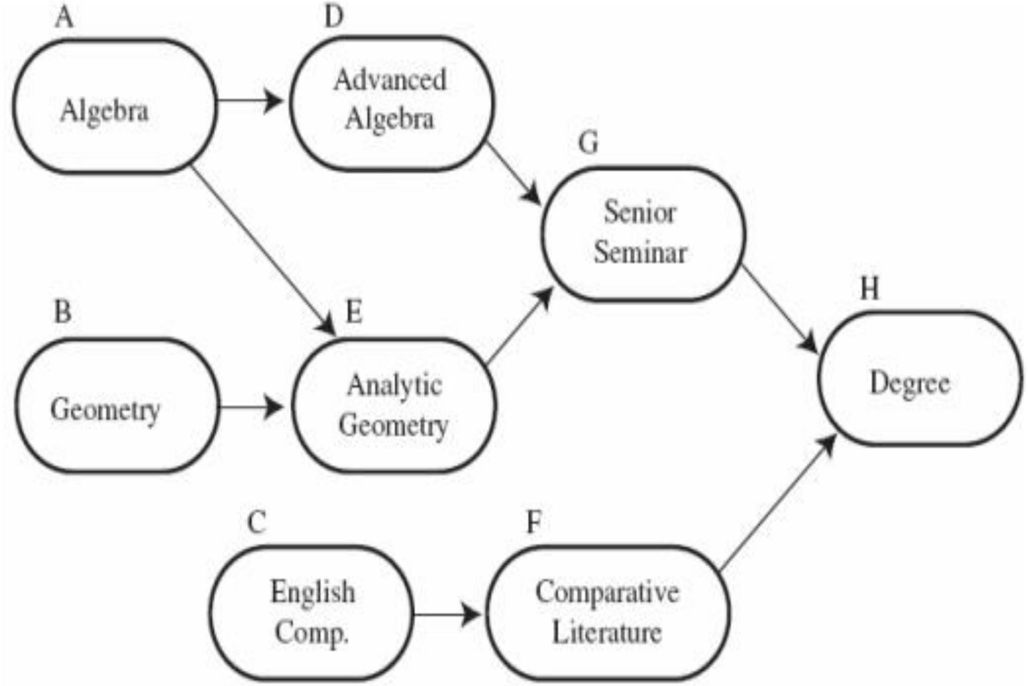
Θ (V+E)

ANALYSIS

- The loops on lines 1–3 and lines 5–7 of DFS take time **ϴ (V),** exclusive of the time to execute the calls to DFS-VISIT.
- The procedure DFS-VISIT is called exactly once for each vertex ϵ V , since the vertex u on which DFS-VISIT is 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 **ϴ (E).**
- The running time of DFS is therefore **ϴ (V + E)**

Applications

Topological Sorting

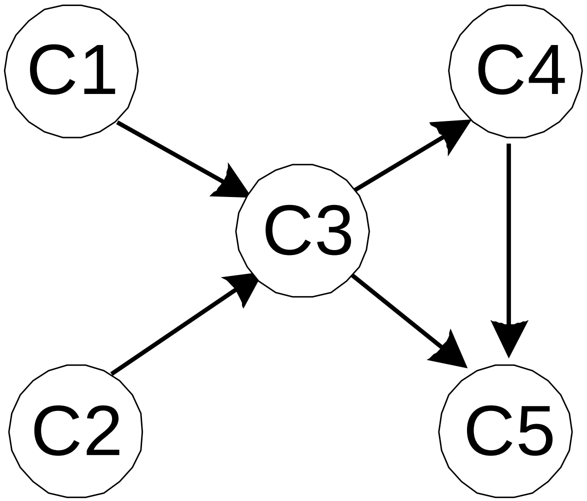


Course prerequisites

Topologically sorted order: C F B A E D G H

Topological Sorting

- Problem: find a total order consistent with a partial order
- Example:



Five courses has the prerequisite relation shown in the left. Find the right order to take all of them sequentially

Note: problem is solvable iff graph is dag

Topological Sorting Algorithms

DFS-based algorithm:

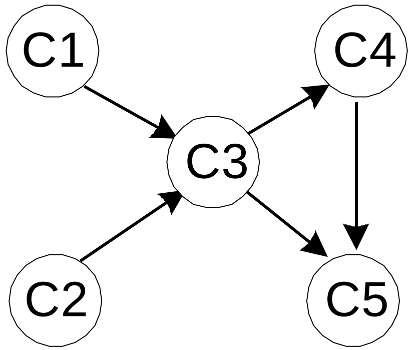
- DFS traversal: note the order with which the vertices are popped off stack (dead end)
- Reverse order solves topological sorting
- Back edges encountered?→ NOT a DAG!

Source removal algorithm

- Repeatedly identify and remove a *source* vertex, ie, a vertex that has no incoming edges

- Both $\Theta(V+E)$ using adjacency linked lists

AN EXAMPLE (TOPOLOGICAL SORT)



(a)

$C5_1$

$C4_2$

$C3_3$

$C1_4$ $C2_5$

(b)

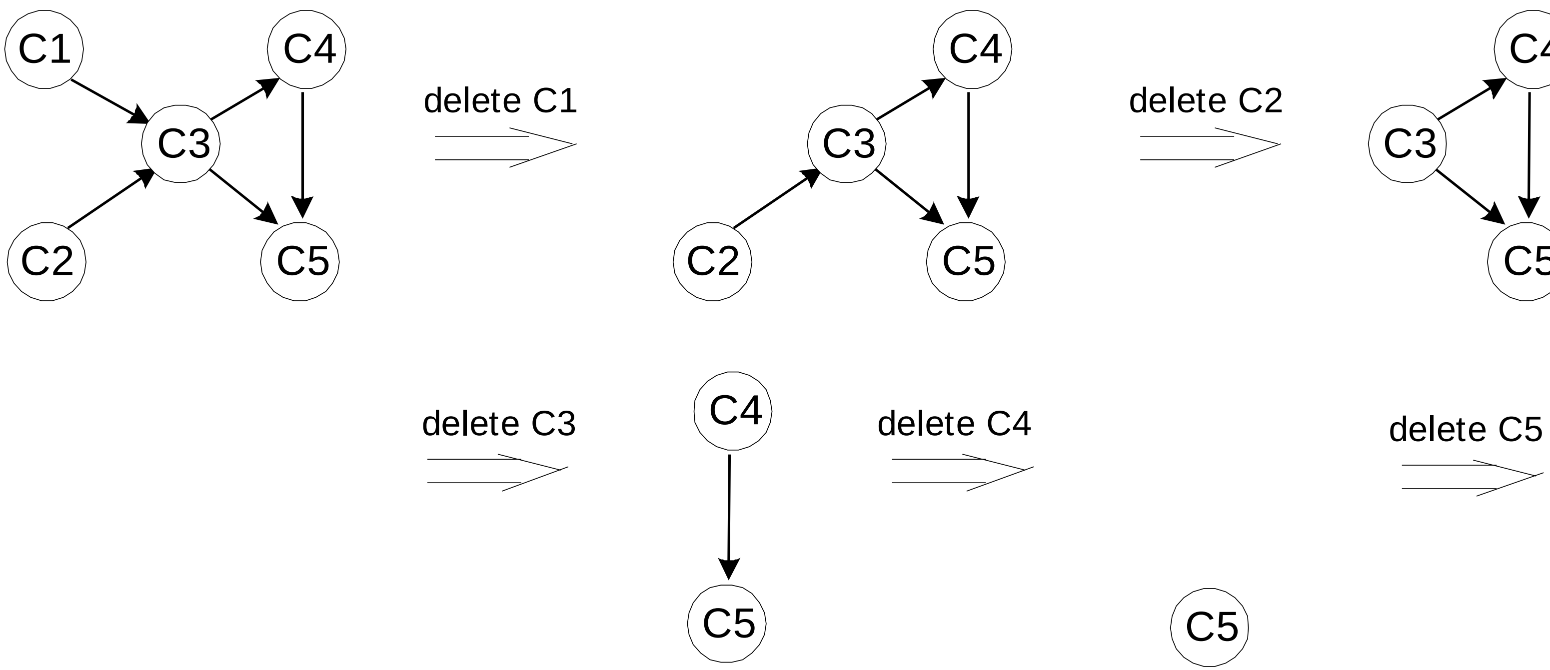
The popping-off order:

C5, C4, C3, C1, C2

The topologically sorted list:

C2 C1 → C3 → C4 → C5

(c)



Topological Sort

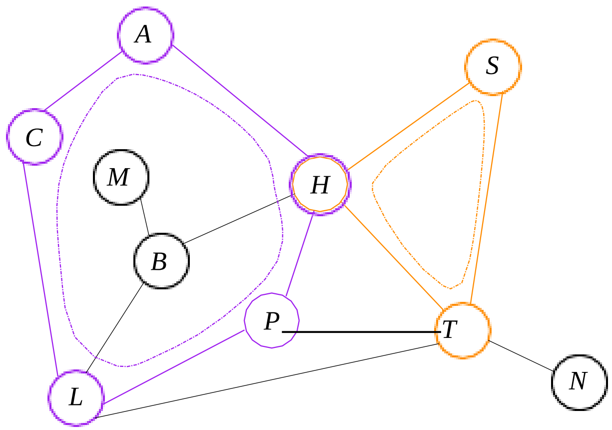
TOPOLOGICAL-SORT(G)

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

Cycle

A path $(v_0, v_1, v_2, \ldots, v_k)$ forms a cycle if $v_0 = v_k$ and all edges on the path are distinct

- A cycle is simple if $v_1, v_2, \ldots, v_k$ are distinct
- A graph with no cycles is acyclic



- (T, S, H, T) is a simple cycle
- (A, C, L, P, H, A) is a simple cycle

An Application of DFS: Cycle Finding

Question

Given an undirected graph G, how to determine whether or not G contains a cycle?

Lemma

G is acyclic if and only if a DFS of G yields no back edges.

Proof.

$\Rightarrow$: Suppose that there is a back edge (u, v). Then, vertex v is an ancestor (excluding predecessor) of u in the DFS trees. There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.
$\Leftarrow$: If there is no back edge, then since an edge in an undirected graph is either a tree edge or a back edge, there are only tree edges, implying that the graph is a forest, and hence is acyclic.

□

Cycle Finding

Cycle(G)

```
foreach u in V do  color[u] = WHITE;  pred[u] = NULL;
end
foreach u in V do
    if color[u] = WHITE then
        Visit(u);
    end
end
output "No Cycle";
```

Visit(u)

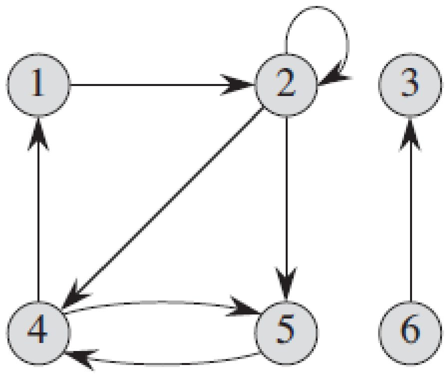
```
color[u] = GRAY;
foreach v in Adj(u) do
    //consider (u, v )
    if color[v] = WHITE then
        // v  unvisited
        pred[v] = u;
        Visit(v); //visit v
    else if v != pred[u] then
        //back edge detected
        output "Cycle found!";
        exit; //terminate
    end
end
color[u] = BLACK;
```

Running time: $O(V)$

- only traverse tree edges, until the first back edge is found
- at most $V - 1$ tree edges

Strongly Connected Component

- A directed graph is ***strongly connected*** if every two vertices are reachable from each other.
- The ***strongly connected components*** of a directed graph are the equivalence classes of vertices under the “are mutually reachable” relation.
- A directed graph is strongly connected if it has only one strongly connected component.
- The graph in Figure B.2(a) has three strongly connected components:
- {1; 2; 4; 5 }, {3}, and {6}
- All pairs of vertices in {1; 2; 4; 5} are mutually reachable.
- The vertices {3; 6} do not form a strongly connected component, since vertex 6 cannot be reached from vertex 3.



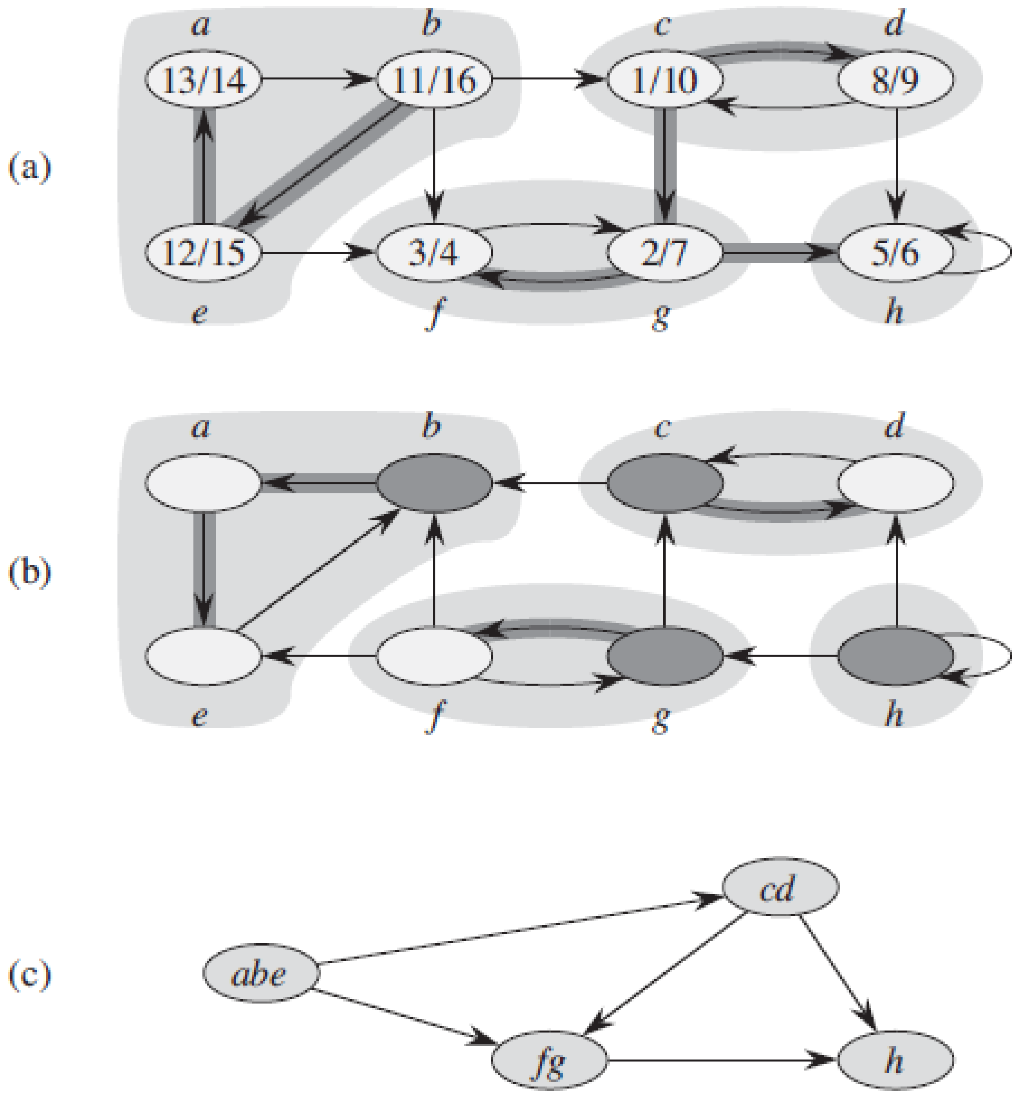
(a)

Strongly Connected Component

The following linear-time (i.e., ,.Θ (V+E) time algorithm computes the strongly connected components of a directed graph G = { V , E } using two depth-first searches, one on G and one on G^T

```
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
```

Strongly Connected Component



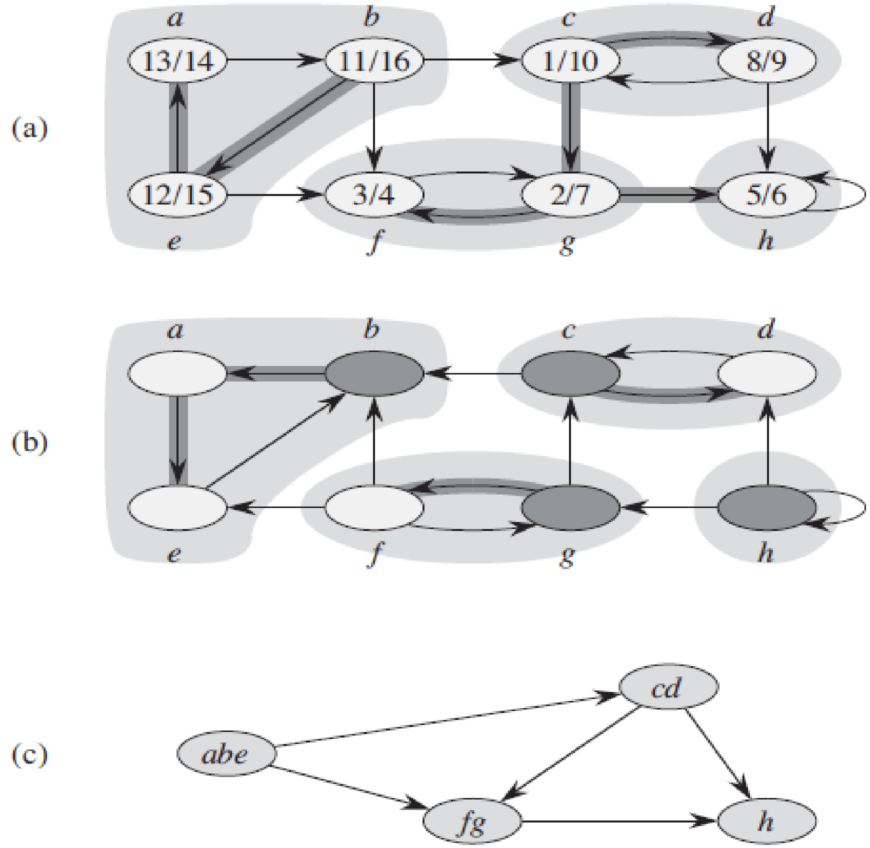


Figure 22.9 **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^{T}. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

Reference

- **Introduction to Algorithms**
  - Chapter # 22
  - Thomas H. Cormen