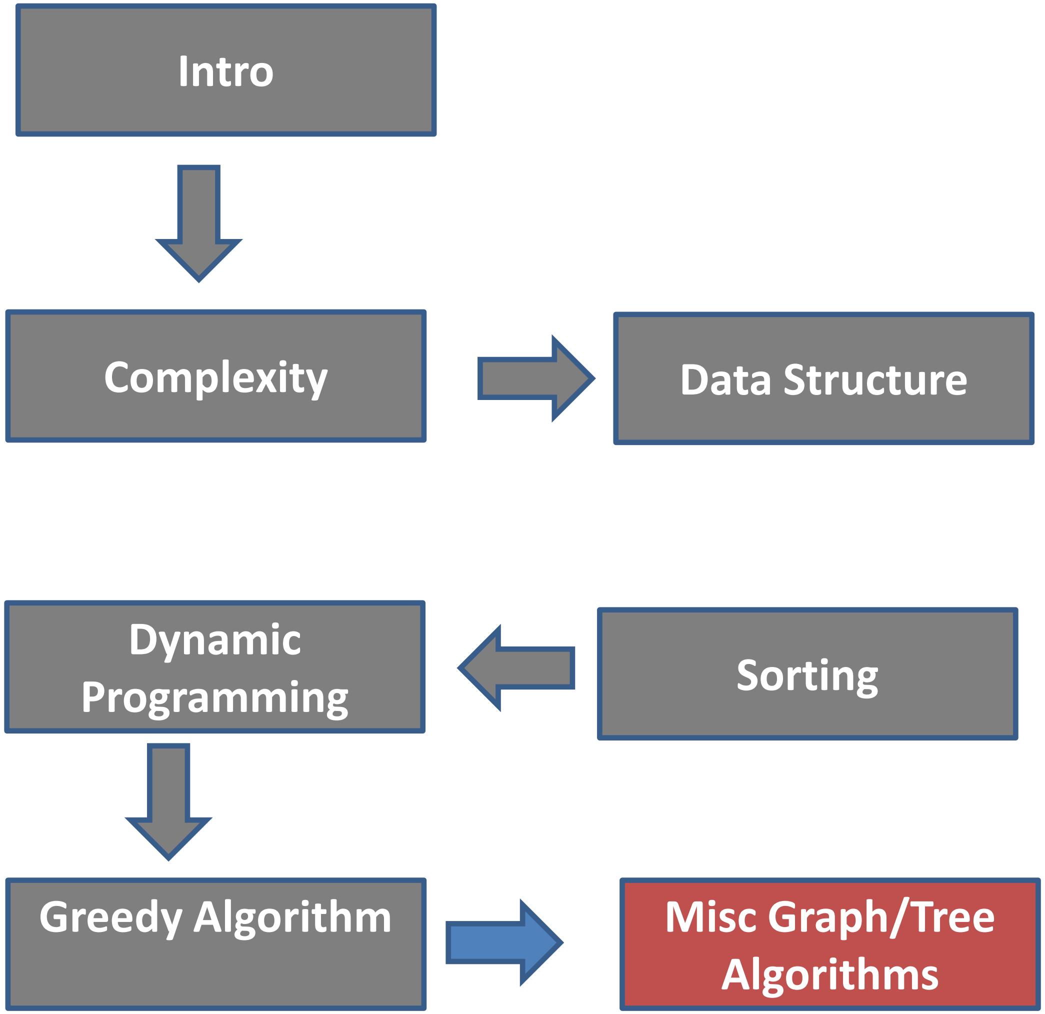
An Introduction to Algorithms

By

Hossein Rahmani

h_rahmani@iust.ac.ir

http://webpages.iust.ac.ir/h_rahmani/


Intro
Complexity
Data Structure
Trees
Hash Functions
Sorting
Dynamic Programming
Greedy Algorithm
Misc Graph/Tree Algorithms

Graphs

- *Graph $G = (V, E)$*
  - V = set of vertices
  - E = set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbf{R}$.
  - Dense: $|E| \approx |V|^2$.
  - Sparse: $|E| << |V|^2$.
- $|E| = O(|V/^2)$

Graphs

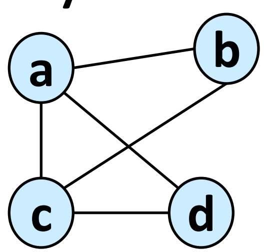
- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  – Symmetric if G is undirected.
  – Not necessarily so if G is directed.
- If G is connected:
  – There is a path between every pair of vertices.
  – $|E| \geq |V| - 1$.
  – Furthermore, if $|E| = |V| - 1$, then G is a tree.

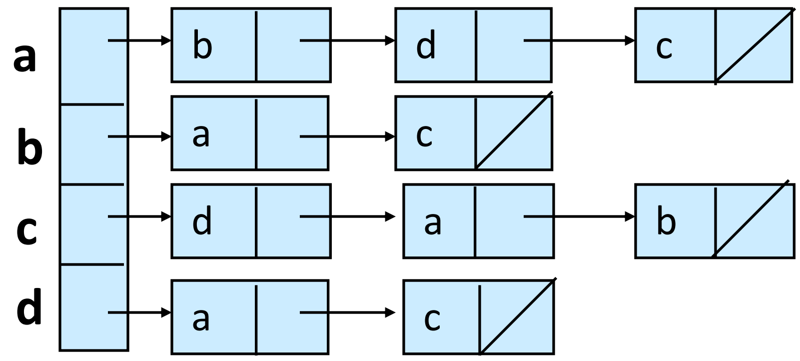
Graph Samples

- Geography:
  - Cities and roads
  - Airports and flights (diameter $\approx$ 20 !!)
- Publications:
  - The co-authorship graph
    - E.g. the Erdos distance
  - The reference graph
- Phone calls: who calls whom
- Almost <u>everything</u> can be modeled as a graph !

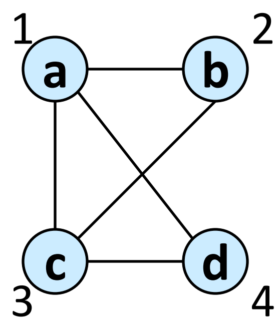
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





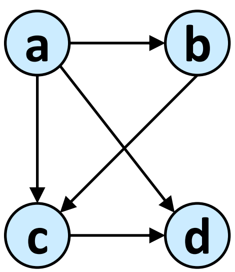
  - Adjacency Matrix.

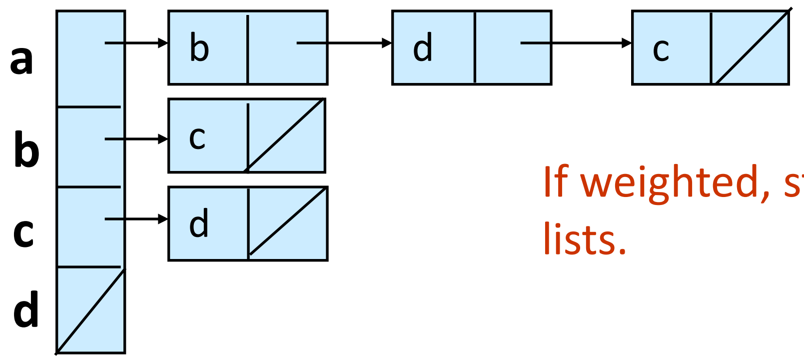


	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

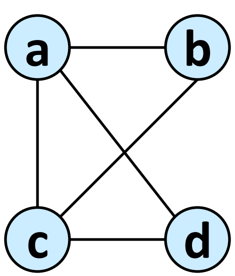
Adjacency Lists

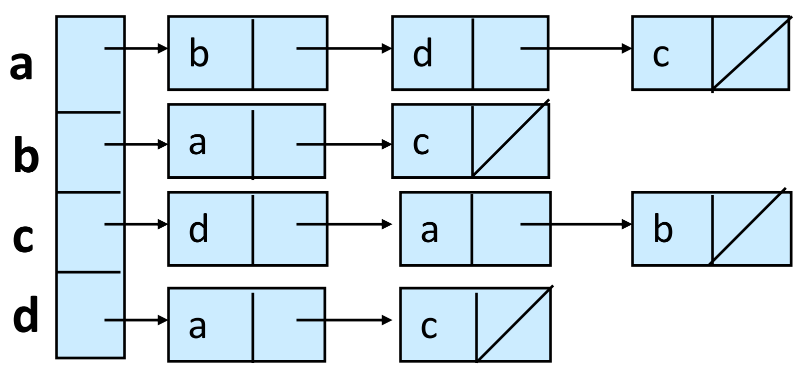
- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u.





If weighted, store weights also in adjacency lists.





Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{out-degree}(v) = |E|$$

    No. of edges leaving *v*

  - Total storage: $\Theta(|V| + |E|)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{degree}(v) = 2|E|$$

    No. of edges incident on *v*. Edge (*u*,*v*) is incident on vertices *u* and *v*.

  - Total storage: $\Theta(|V| + |E|)$

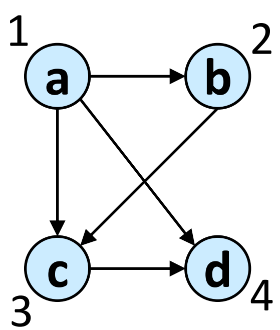
Pros and Cons: adj list

- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.
- Cons
  - Determining if an edge $(u, v) \in G$ is not efficient.
    - Have to search in u's adjacency list. $\Theta(\text{degree}(u))$ time.
    - $\Theta(V)$ in the worst case.

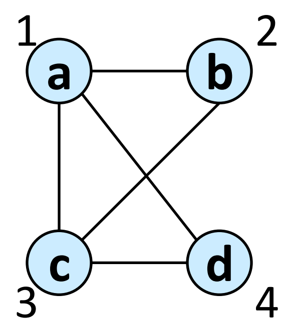
Adjacency Matrix

- $|V| \times |V|$ matrix A.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- A is then given by:

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



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



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

$A = A^{\mathrm{T}}$ for undirected graphs.

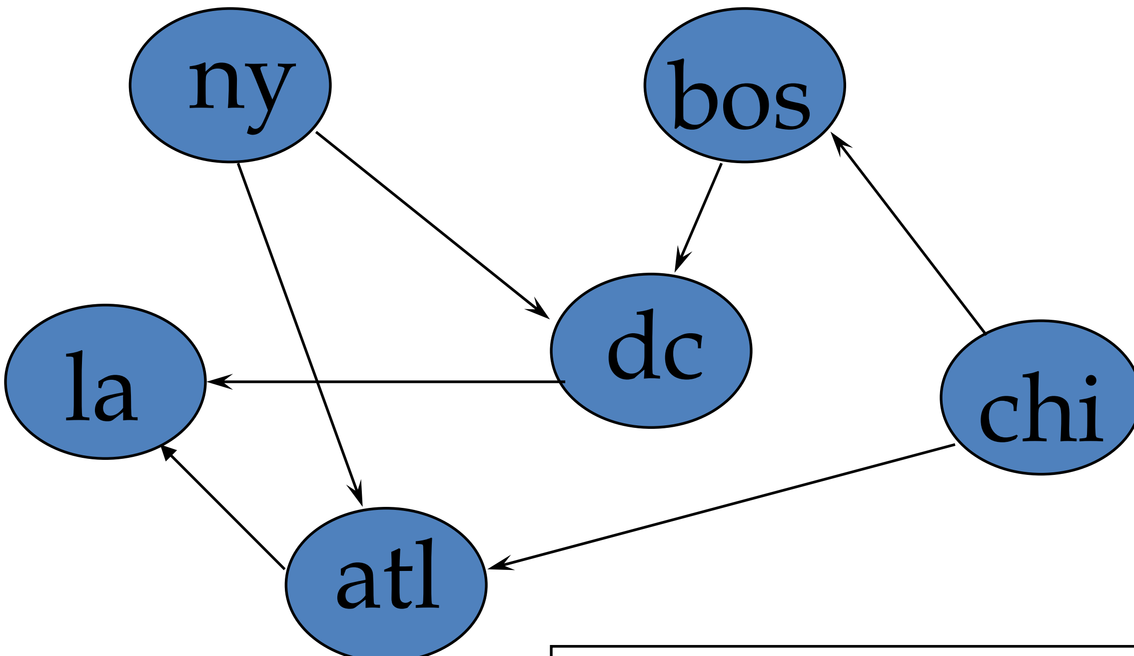
Space and Time

- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph.

Some graph operations

	adjacency matrix	adjacency lists
insertEdge	O(1)	O(e)
isEdge	O(1)	O(e)
#successors?	O(V)	O(e)
#predecessors?	O(V)	O(E)

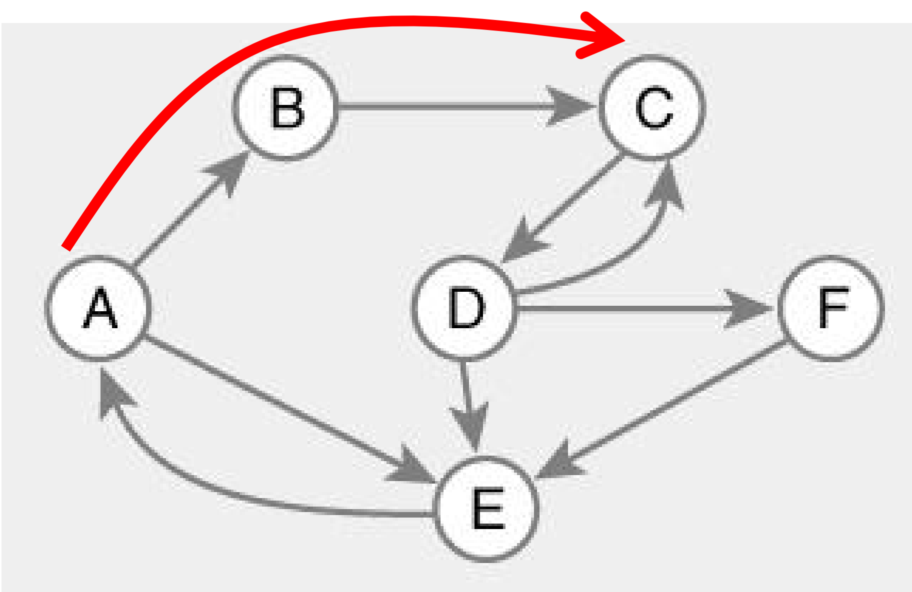
traversing a graph



Where to start?
Will all vertices be visited?
How to prevent multiple visits?

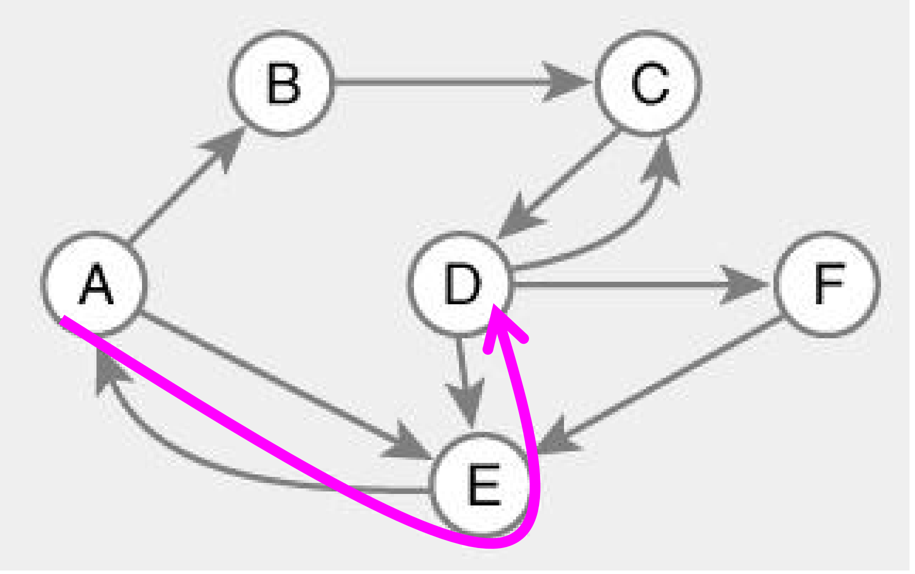
Graph Definitions

- Path
  - Sequence of nodes n_1, n_2, … n_k
  - Edge exists between each pair of nodes n_i , n_{i+1}
  - Example
    - A, B, C is a path



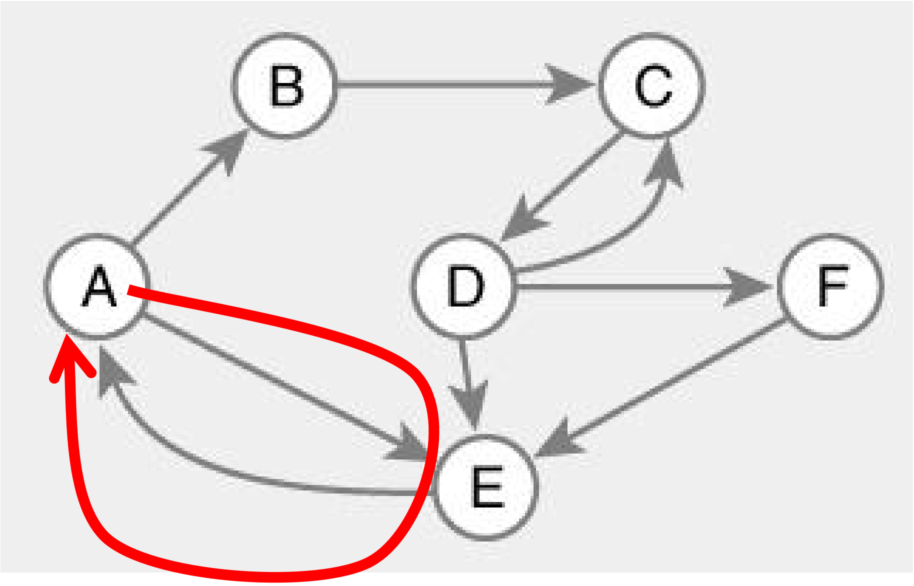
Graph Definitions

- Path
  - Sequence of nodes n_1, n_2, … n_k
  - Edge exists between ea
  - Example
    - A, B, C is a path
    - A, E, D is not a path



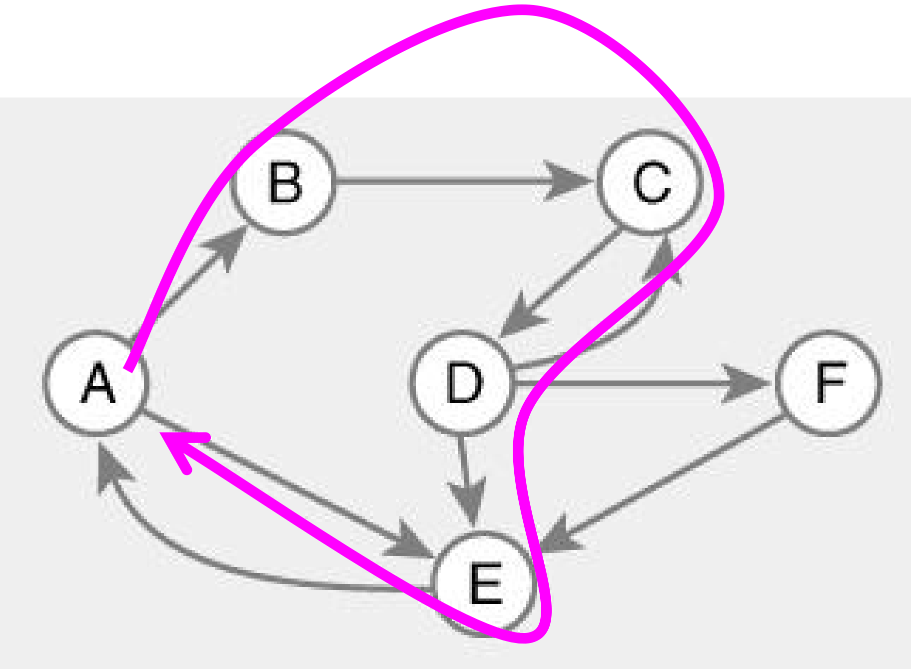
Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A



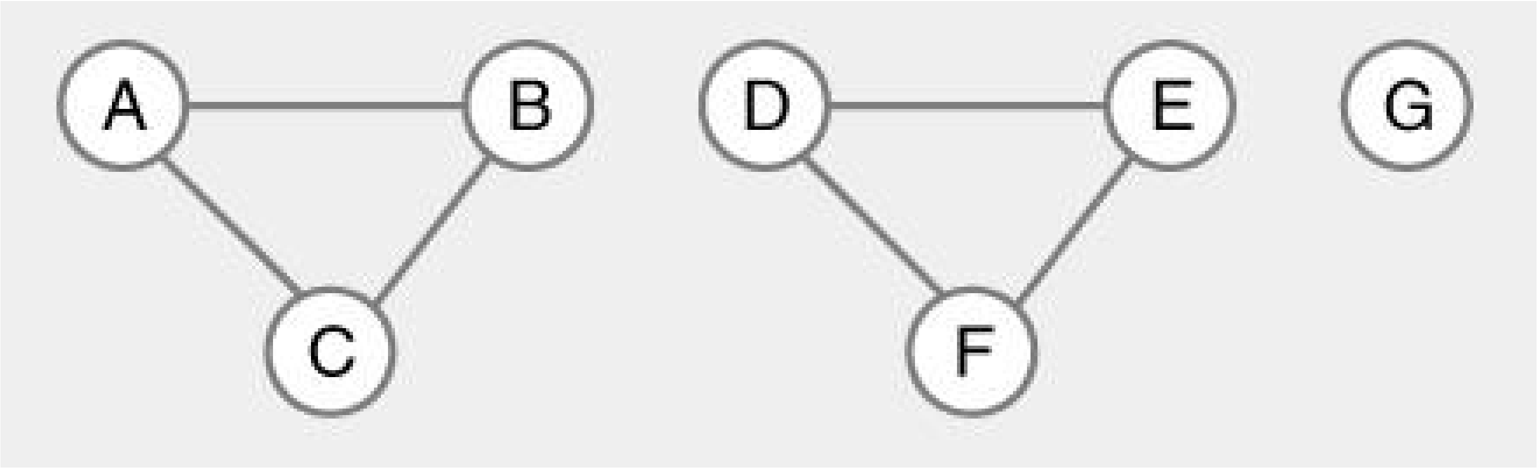
Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A
    - A, B, C, D, E, A
- Simple path
  - No cycles in path
- Acyclic graph
  - No cycles in graph



Graph Definitions

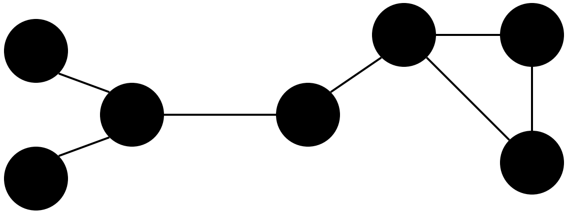
- Reachable
  - Path exists between nodes
- Connected graph
  - Every node is reachable from some node in graph



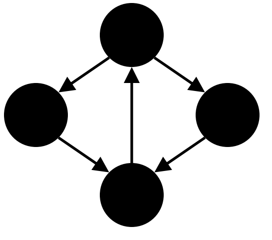
Unconnected graphs

Connectivity

Undirected graphs are *connected* if there is a path between any two vertices

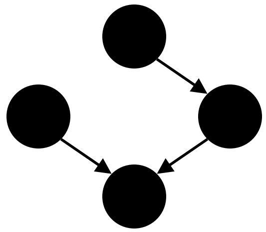


Directed graphs are *strongly connected* if there is a path from any one vertex to any other

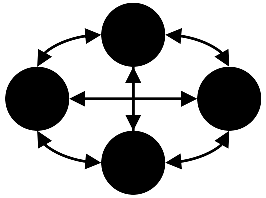


Connectivity

Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



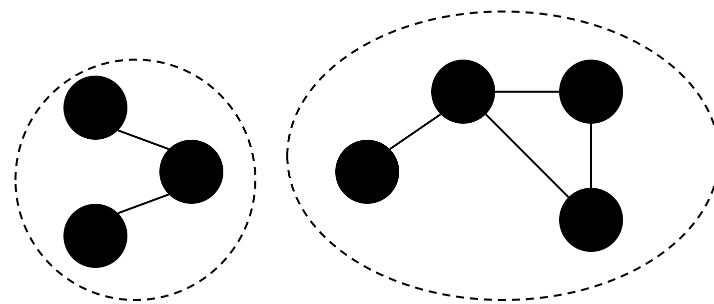
A *complete* graph has an edge between every pair of vertices



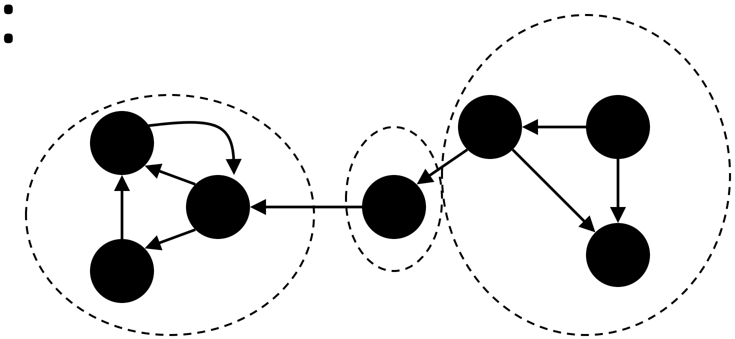
Connectivity

A *(strongly) connected component* is a subgraph which is (strongly) connected

CC in an undirected graph:



SCC in a directed graph:



Distance and Diameter

- The distance between two nodes, d(u,v), is the length of the shortest paths, or ∞ if there is no path

- The diameter of a graph is the largest distance between any two nodes

- Graph is strongly connected iff diameter < ∞

Bipartiteness

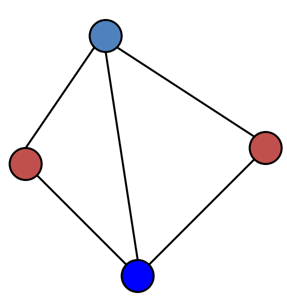
Graph G = (V,E) is **bipartite** iff it can be partitioned into two sets of nodes A and B such that each edge has one end in A and the other end in B

Alternatively:

- Graph G = (V,E) is bipartite iff all its cycles have even length
- Graph G = (V,E) is bipartite iff nodes can be coloured using two colours

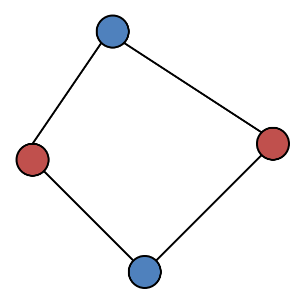
Question: given a graph G, how to test if the graph is bipartite?

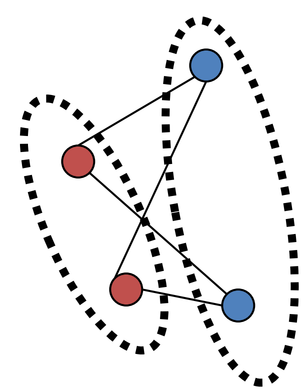
Note: graphs without cycles (trees) are bipartite



non bipartite

bipartite:



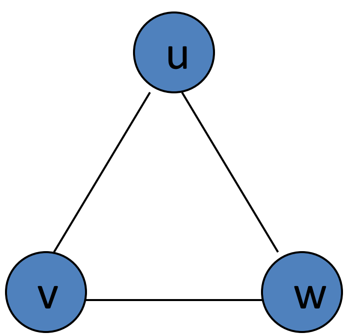


Subgraphs

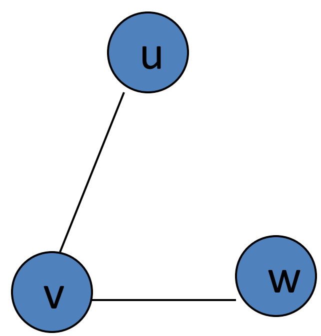
- A subgraph of a graph G = (V, E) is a graph H =(V’, E’) where V’ is a subset of V and E’ is a subset of E

  Application example: solving sub-problems within a graph

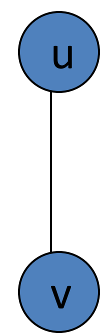
  Representation example: V = {u, v, w}, E = ({u, v}, {v, w}, {w, u}}, H_1 , H_2



G



H_1



H_2

Graph - Isomorphism

- G1 = (V1, E2) and G2 = (V2, E2) are isomorphic if:
- There is a one-to-one and onto function f from V1 to V2 with the property that
  - a and b are adjacent in G1 if and only if f (a) and f (b) are adjacent in G2, for all a and b in V1.
- Function f is called isomorphism

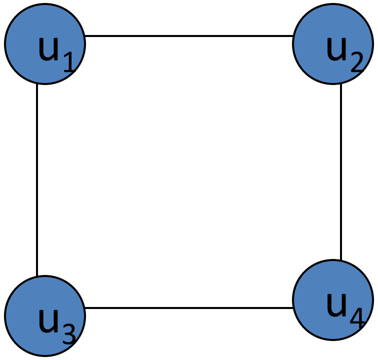
Application Example:

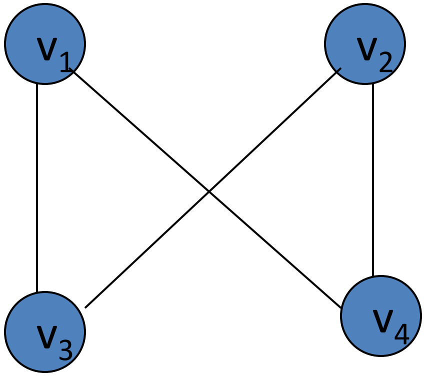
In chemistry, to find if two compounds have the same structure

Graph - Isomorphism

Representation example: G1 = (V1, E1) , G2 = (V2, E2)

$f(u_1) = v_1$, $f(u_2) = v_4$, $f(u_3) = v_3$, $f(u_4) = v_2$,





Graph-searching Algorithms

- Searching a graph:
  - Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

Breadth-first Search

- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v] =$ distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - Builds breadth-first tree <u>with root *s*</u> that contains all <u>reachable</u> vertices.

Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.

BFS(G,s)

```
for each vertex u in V[G] – {s}
        do color[u] ← white
           d[u] ← ∝
           π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        do u ← dequeue(Q)
              for each v in 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
```

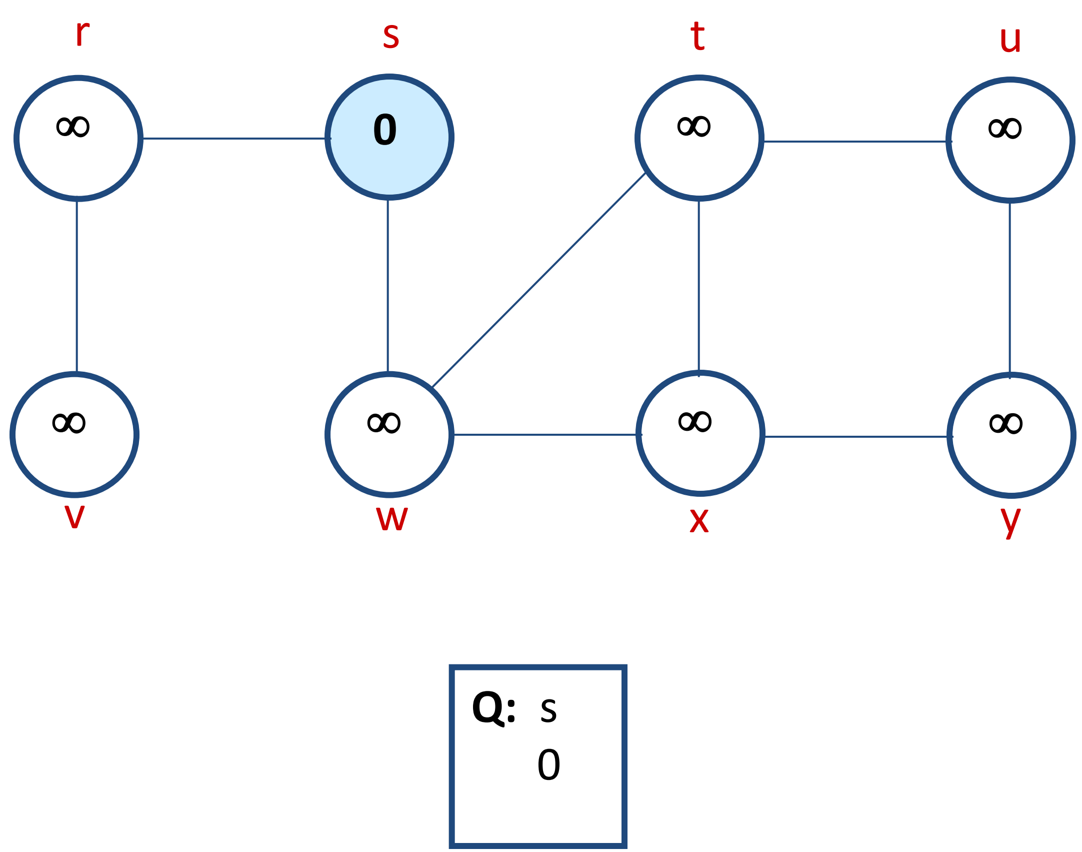
initialization

access source *s*

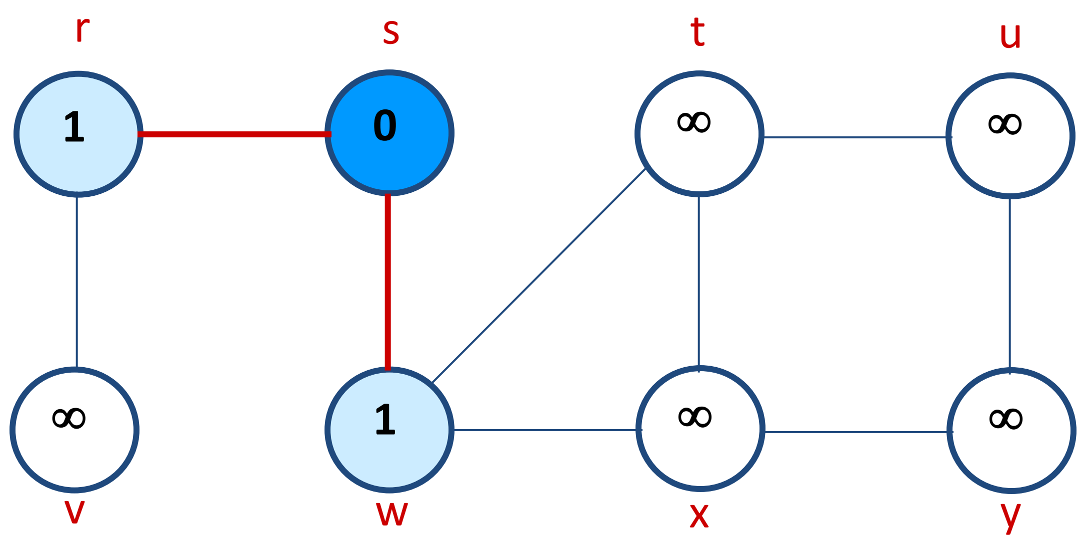
white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

Example (BFS)

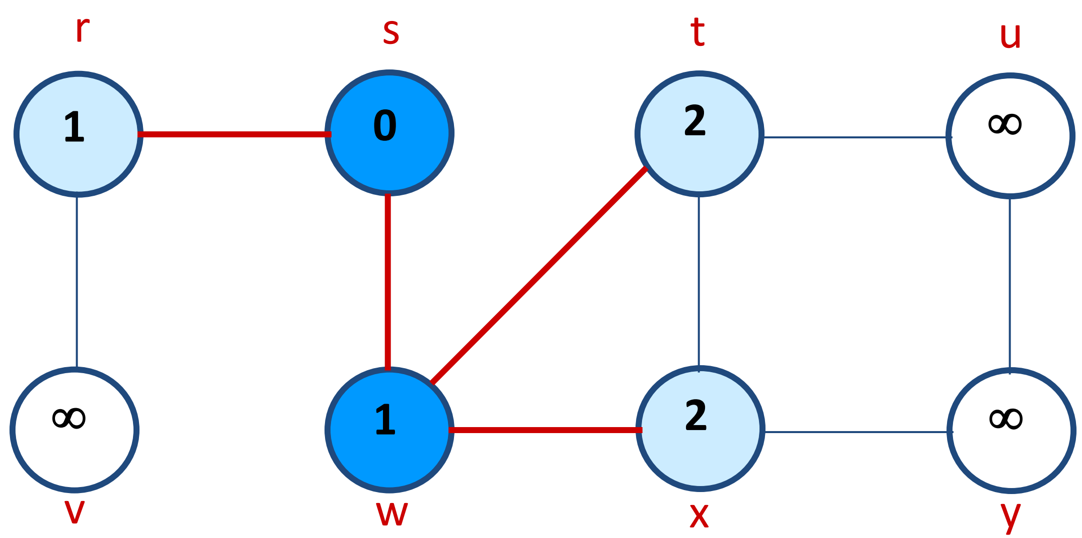


Example (BFS)



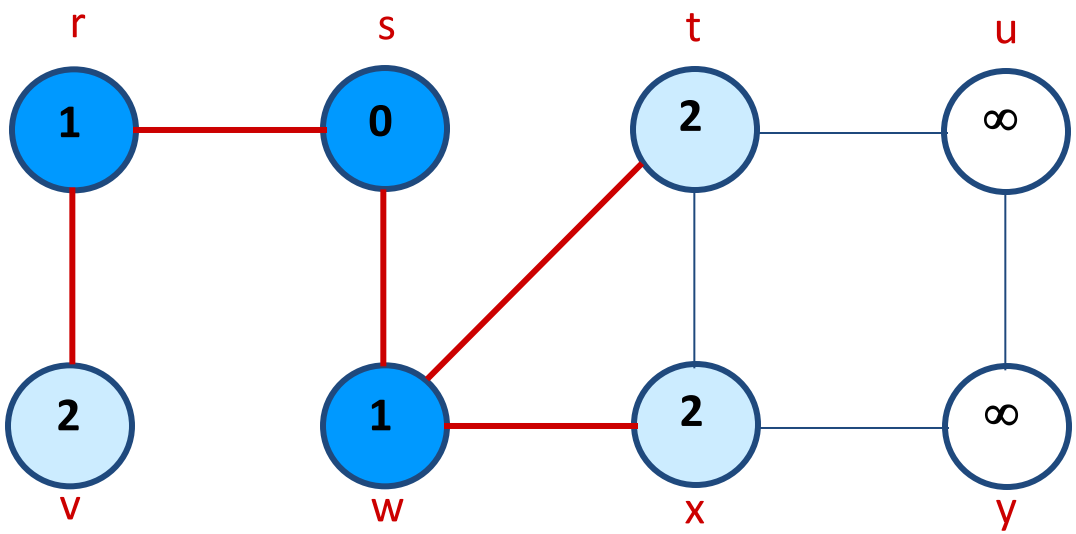
Q: w r
1 1

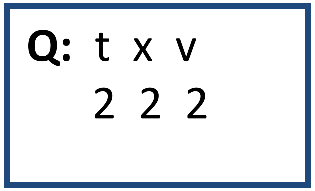
Example (BFS)



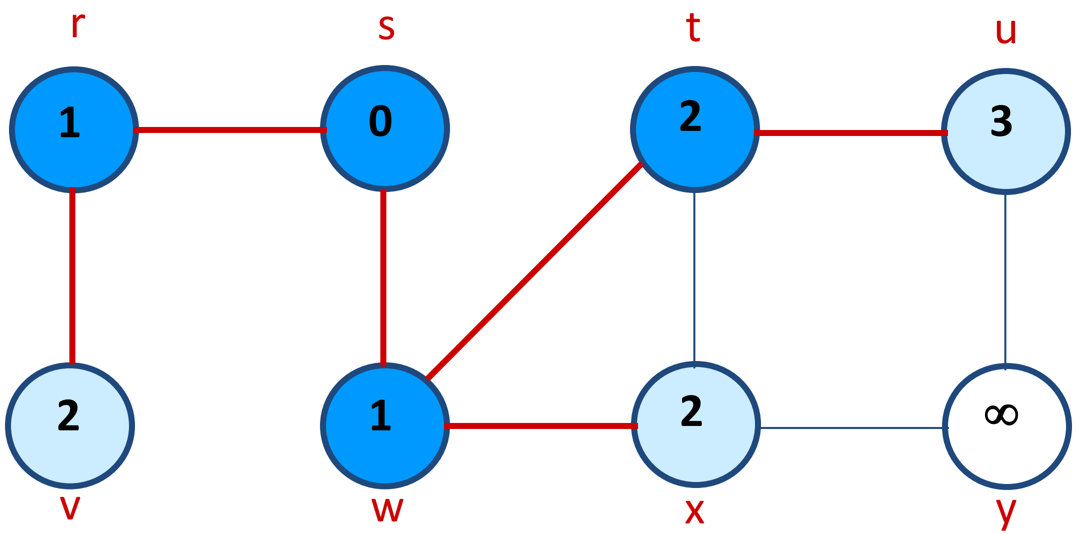
Q: r t x
1 2 2

Example (BFS)



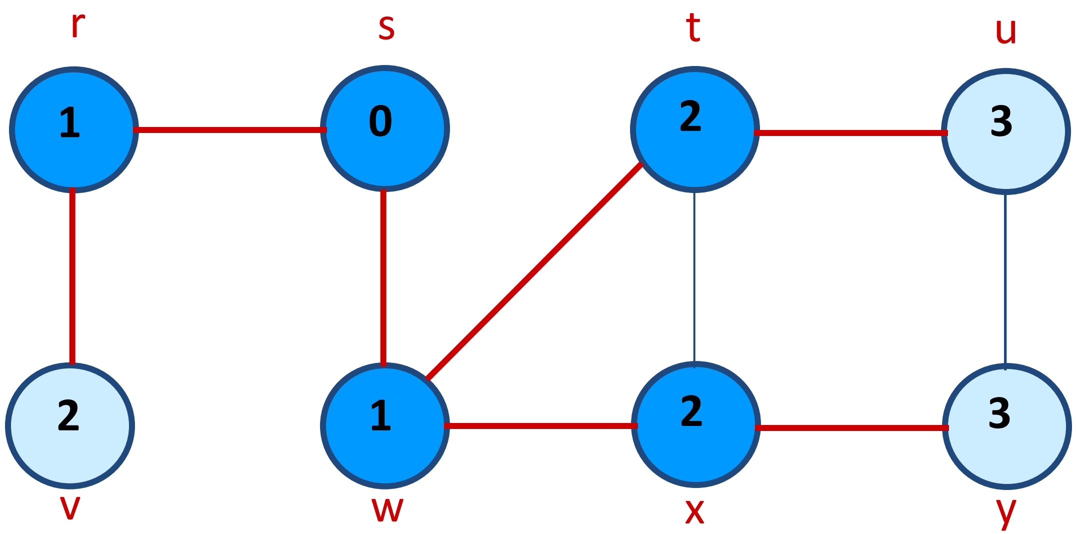


Example (BFS)



Q: x v u
2 2 3

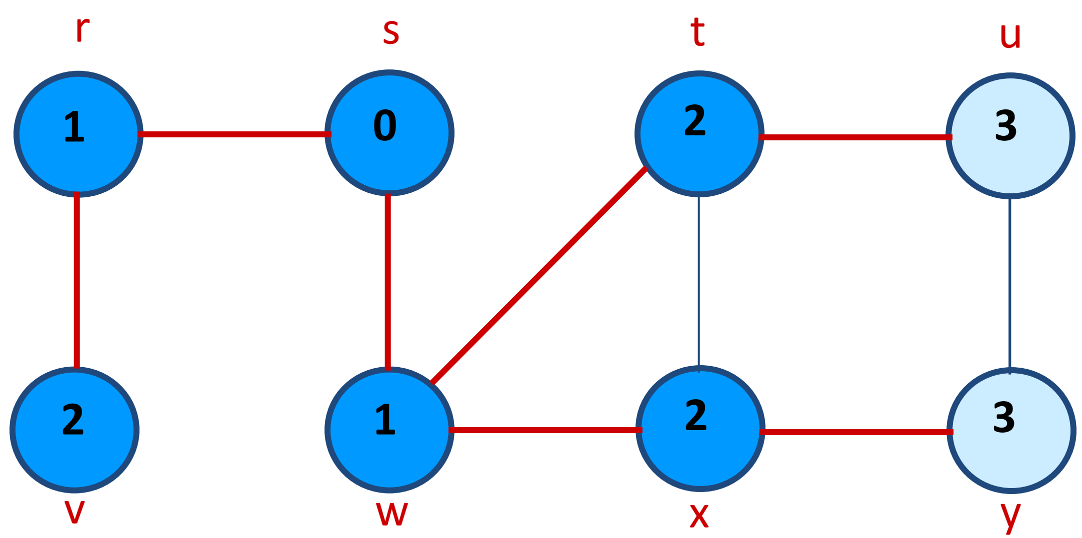
Example (BFS)



Q: v u y

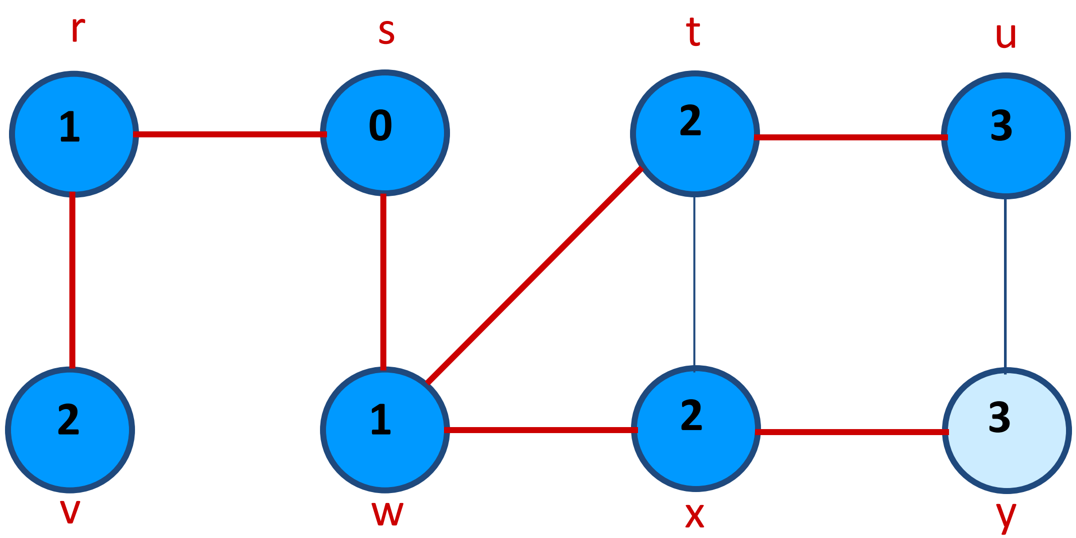
2 3 3

Example (BFS)



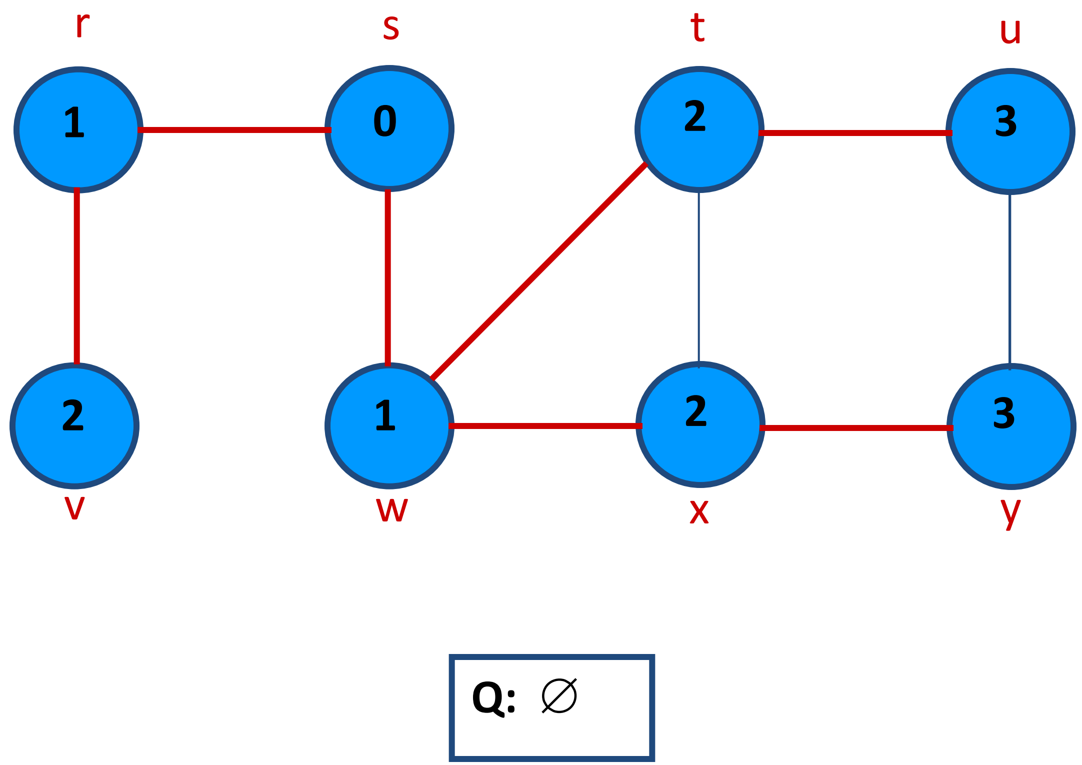
Q: u y
3 3

Example (BFS)

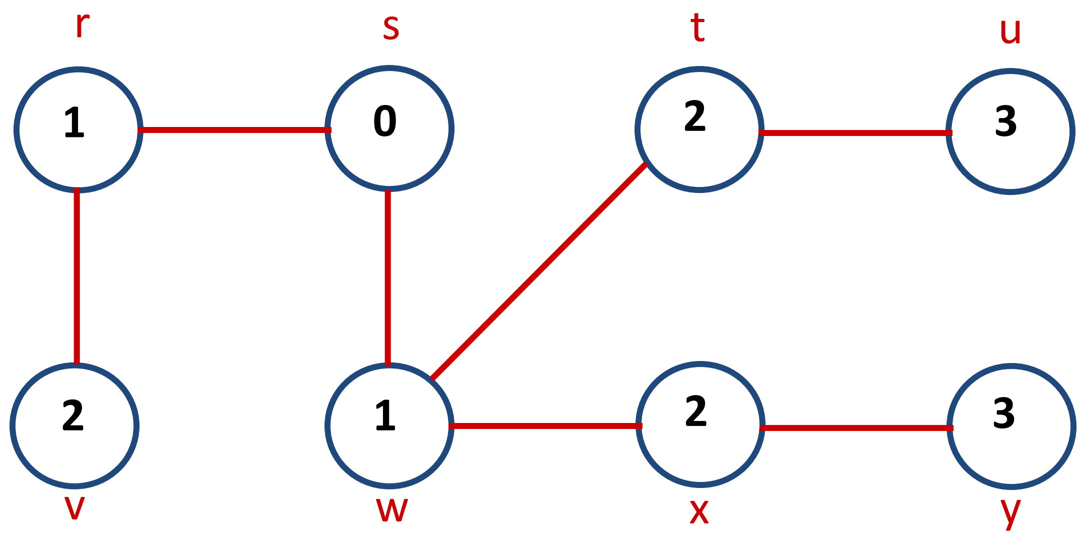


Q: y
3

Example (BFS)



Example (BFS)



BF Tree

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered.
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that <u>*v* was discovered</u> during the scan of <u>*u*'s adjacency list</u>.
- Coloring scheme for vertices as BFS. A vertex is
  - "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if it is a leaf node or all vertices adjacent to it have been finished.

Pseudo-code

<u>DFS(G)</u>

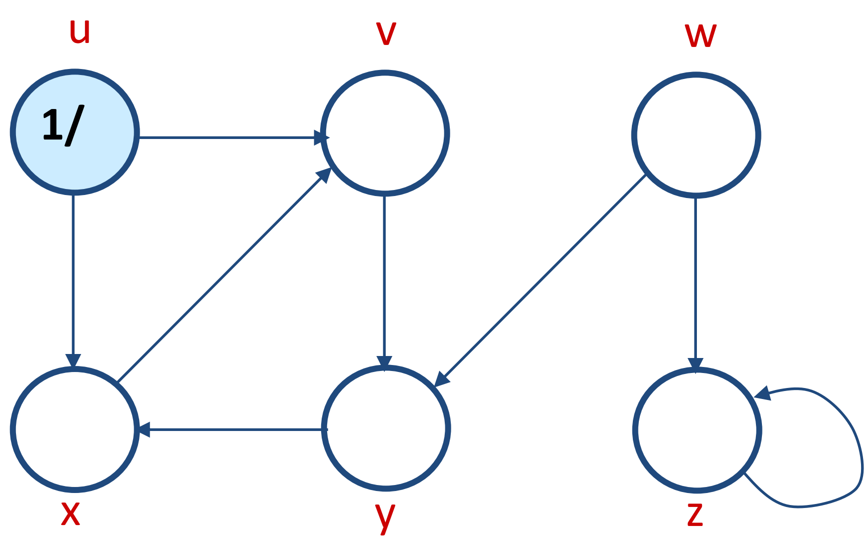
1. **for** each vertex $u \in V[G]$
2. **do** $color[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$
6. **do if** $color[u]$ = white
7. **then** DFS-Visit(u)

Uses a global timestamp ***time***.

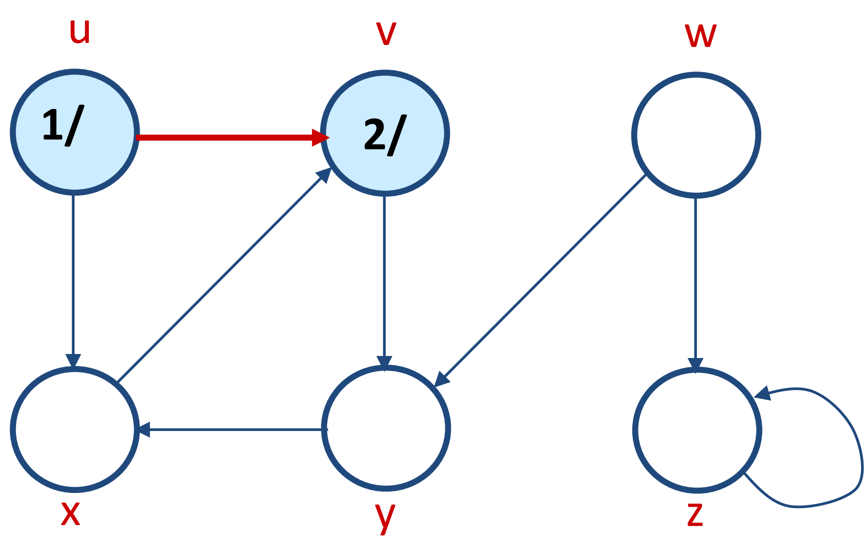
DFS-Visit(u)

1. $color[u] \leftarrow$ GRAY // White vertex u has been discovered
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for** each $v \in Adj[u]$
5. **do if** $color[v]$ = WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK // Blacken u; it is finished.
9. $f[u] \leftarrow time \leftarrow time + 1$

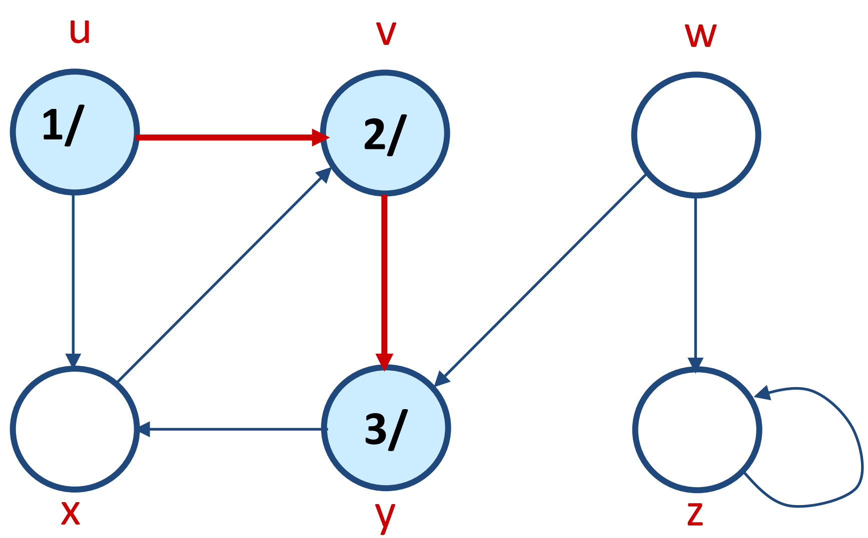
Example (DFS)



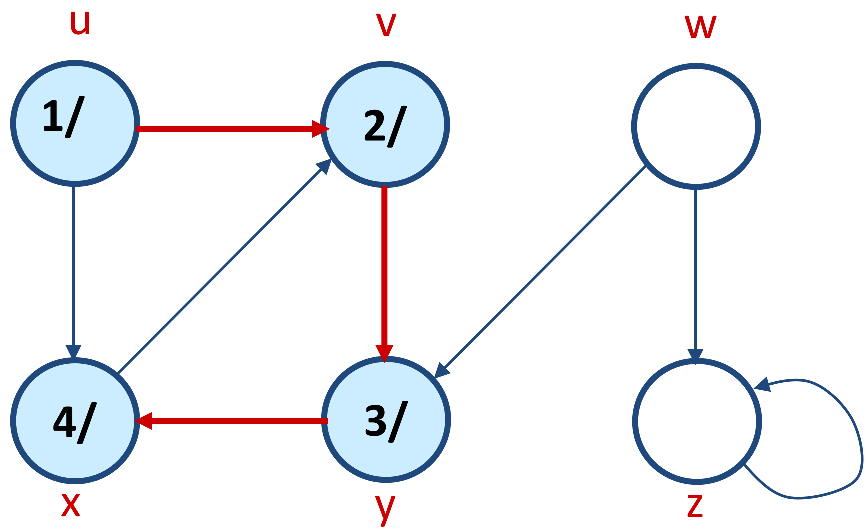
Example (DFS)



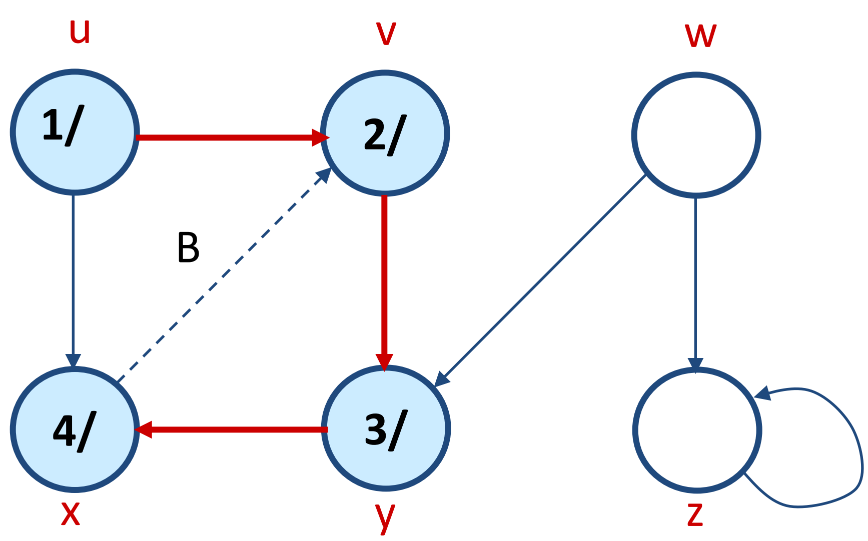
Example (DFS)



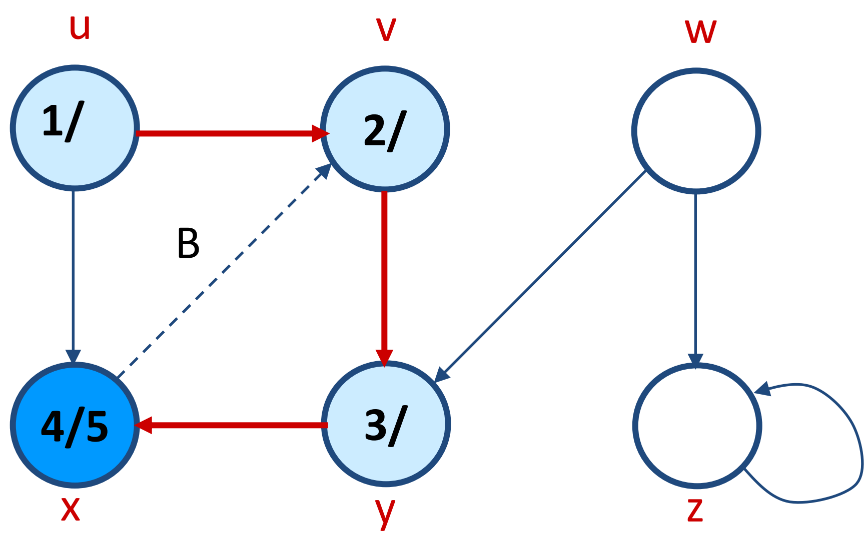
Example (DFS)



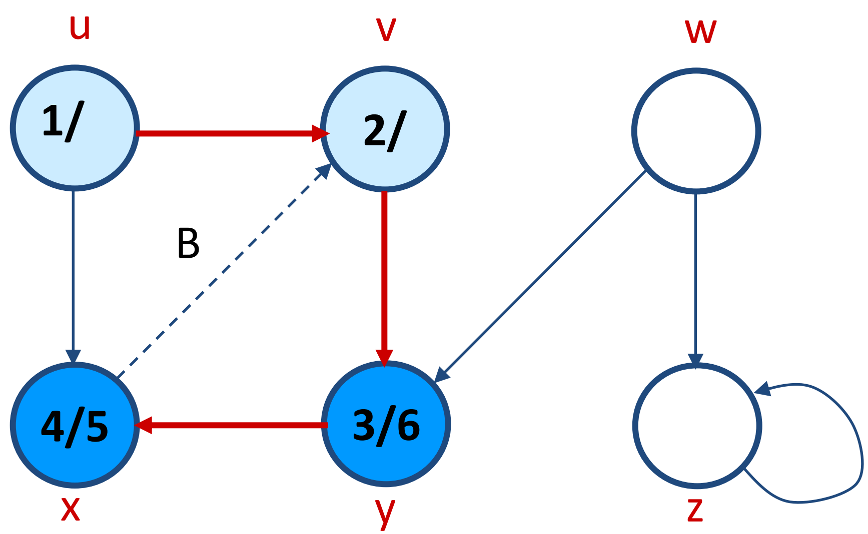
Example (DFS)



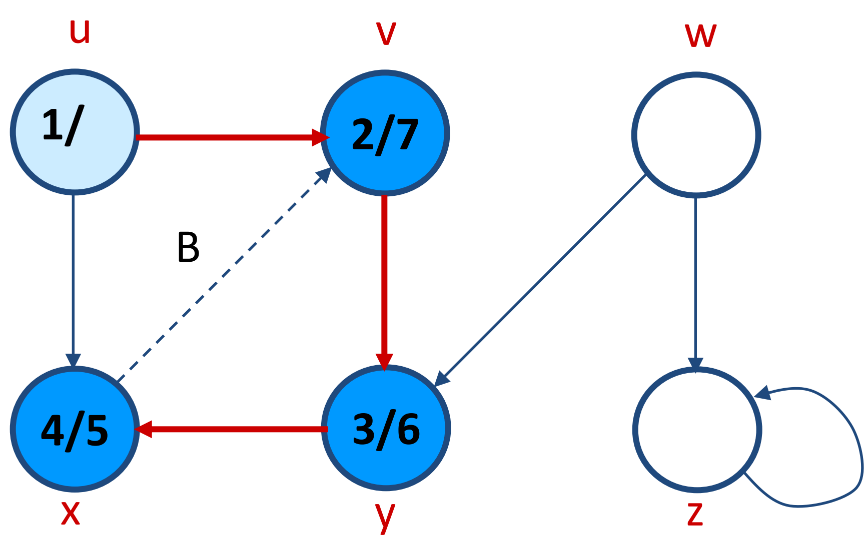
Example (DFS)



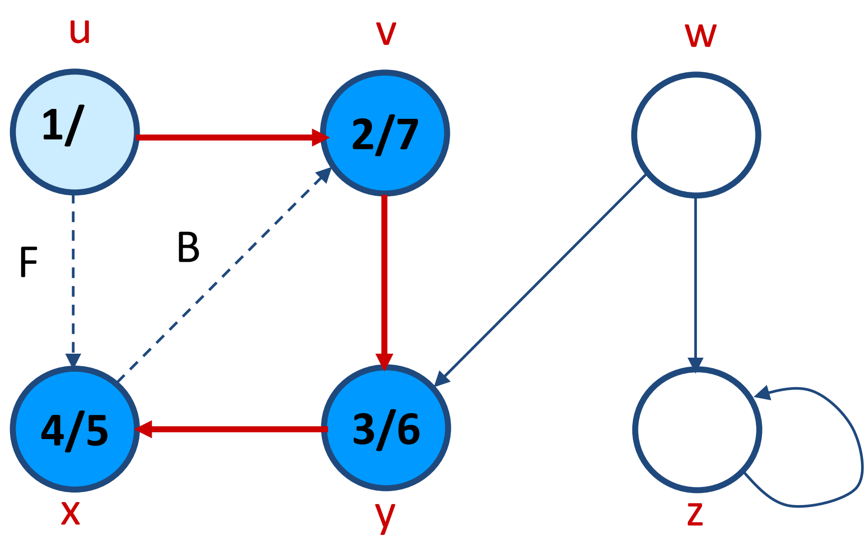
Example (DFS)



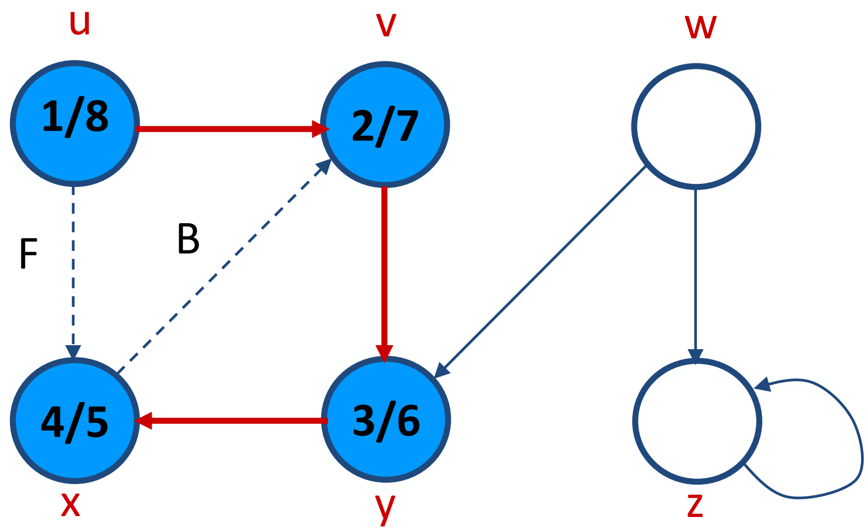
Example (DFS)



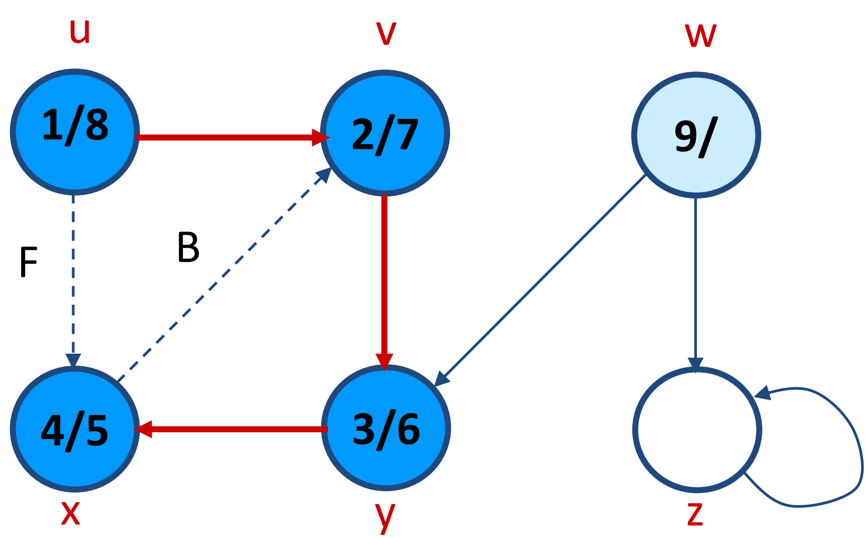
Example (DFS)



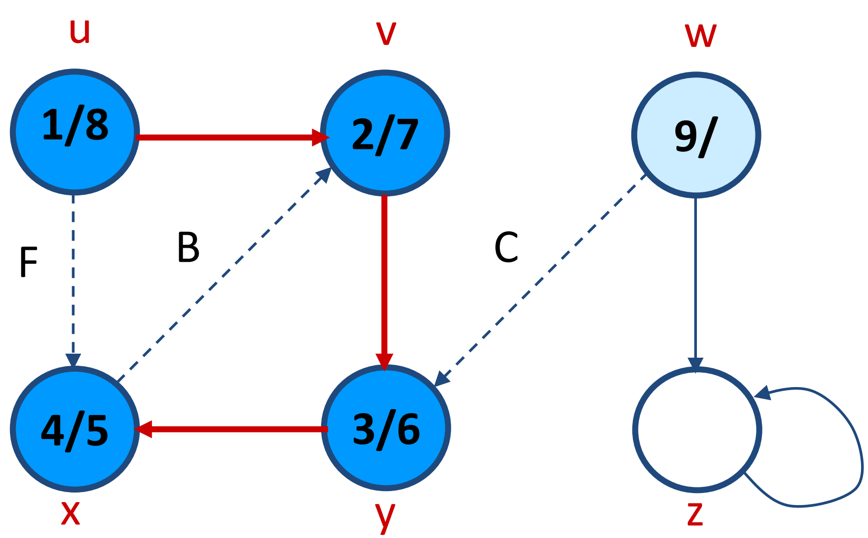
Example (DFS)



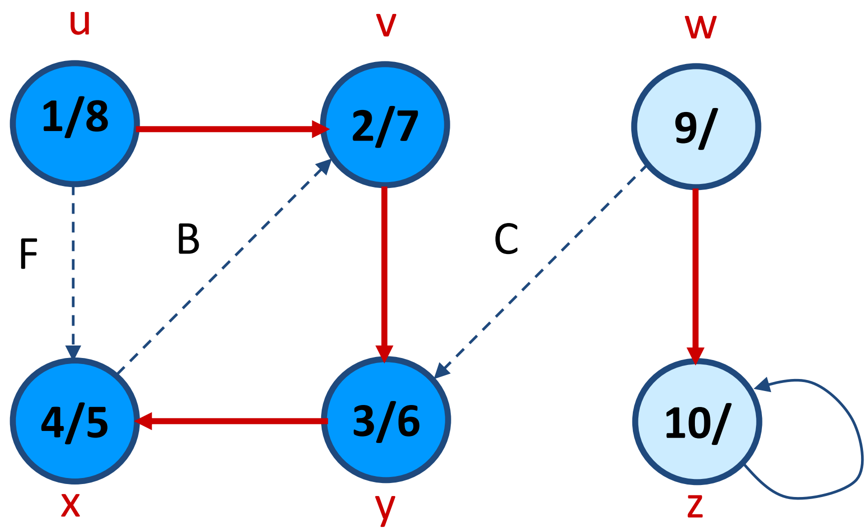
Example (DFS)



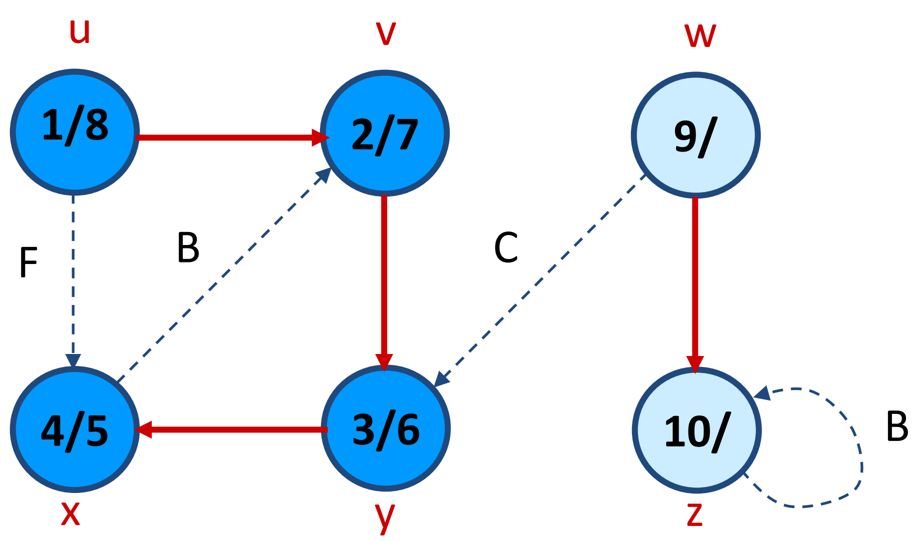
Example (DFS)



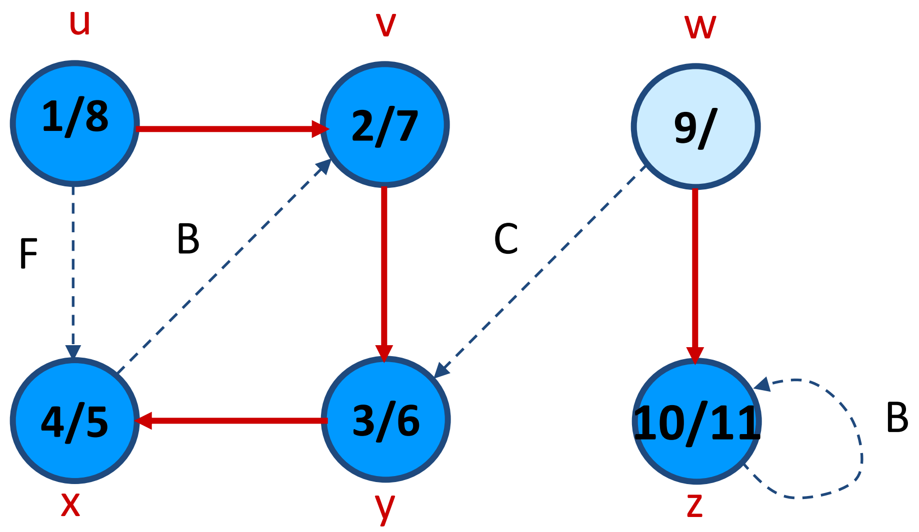
Example (DFS)



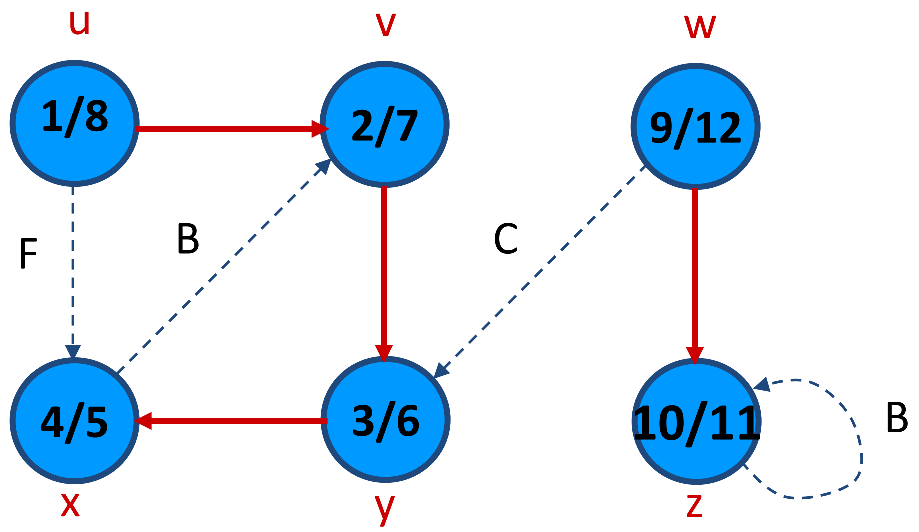
Example (DFS)



Example (DFS)



Example (DFS)!!!

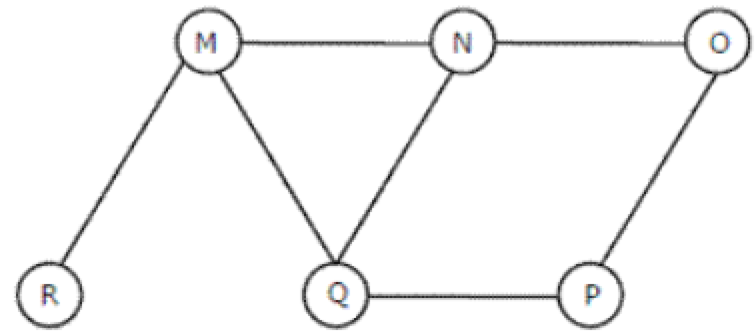


Recursive DFS Algorithm

```
Traverse( )
    for all nodes X
        visited[X]= False
    DFS( 1st node )
DFS( X )
    visited[X] = True
    for each successor Y of X
        if (visited[Y] = False)
            DFS( Y )
```

Quiz 1

- The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is

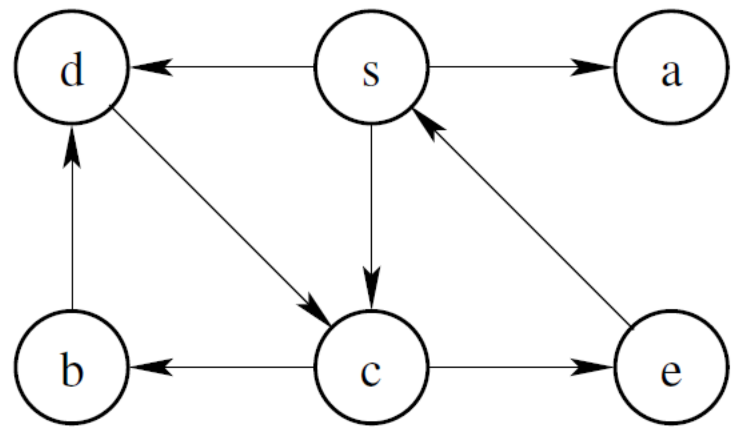


A	MNOPQR
B	NQMPOR
C	QMNPRO
D	QMNPOR

Quiz 2

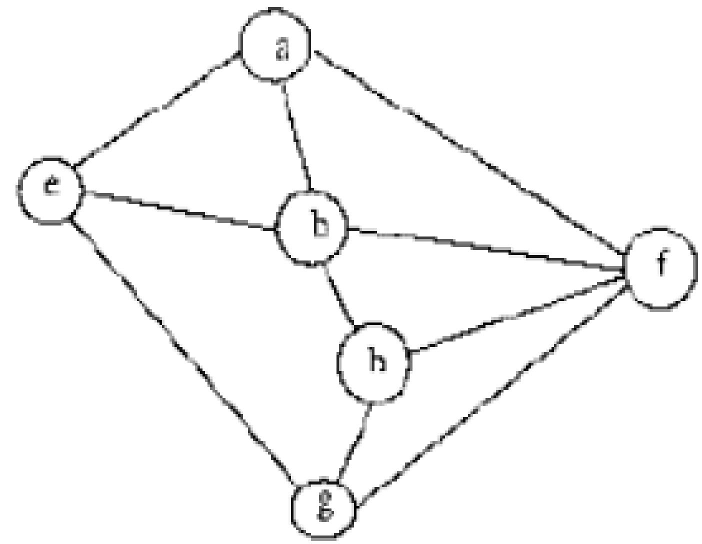
Give the visited node order for each type of graph search, starting with s, given the following adjacency lists and accompanying figure:

$$
\begin{aligned}
adj(s) &= [a, c, d],\\
adj(a) &= [\,],\\
adj(c) &= [e, b],\\
adj(b) &= [d],\\
adj(d) &= [c],\\
adj(e) &= [s].
\end{aligned}
$$



- BFS?
- DFS?

Quiz 3

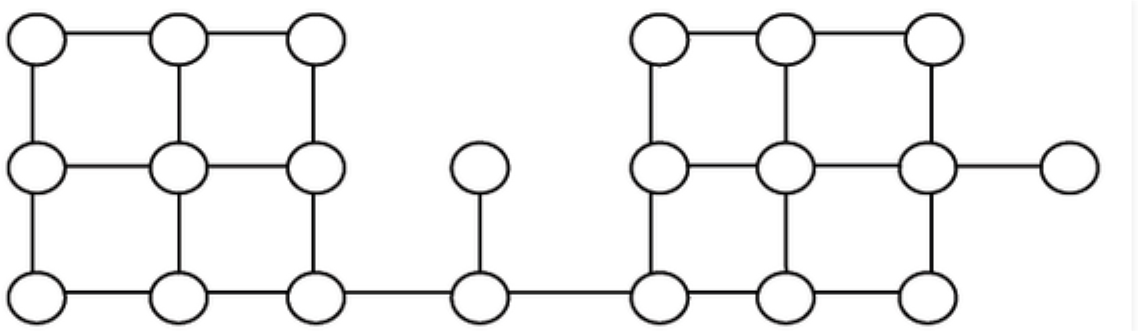


- Consider the following graph
  Among the following sequences
  I) a b e g h f
  II) a b f e h g
  III) a b f h g e
  IV) a f g h b e Which are depth first traversals of the above graph

A	I, II and IV only
B	I and IV only
C	II, III and IV only
D	I, III and IV only

Quiz 4

- Suppose depth first search is executed on the graph below starting at some unknown vertex. Assume that a recursive call to visit a vertex is made only after first checking that the vertex has not been visited earlier. Then the maximum possible recursion depth (including the initial call) is ?



A	17
B	18
C	19
D	20

Quiz 5

- Discuss the Order of BFS and DFS algorithms, with respect to V and E.