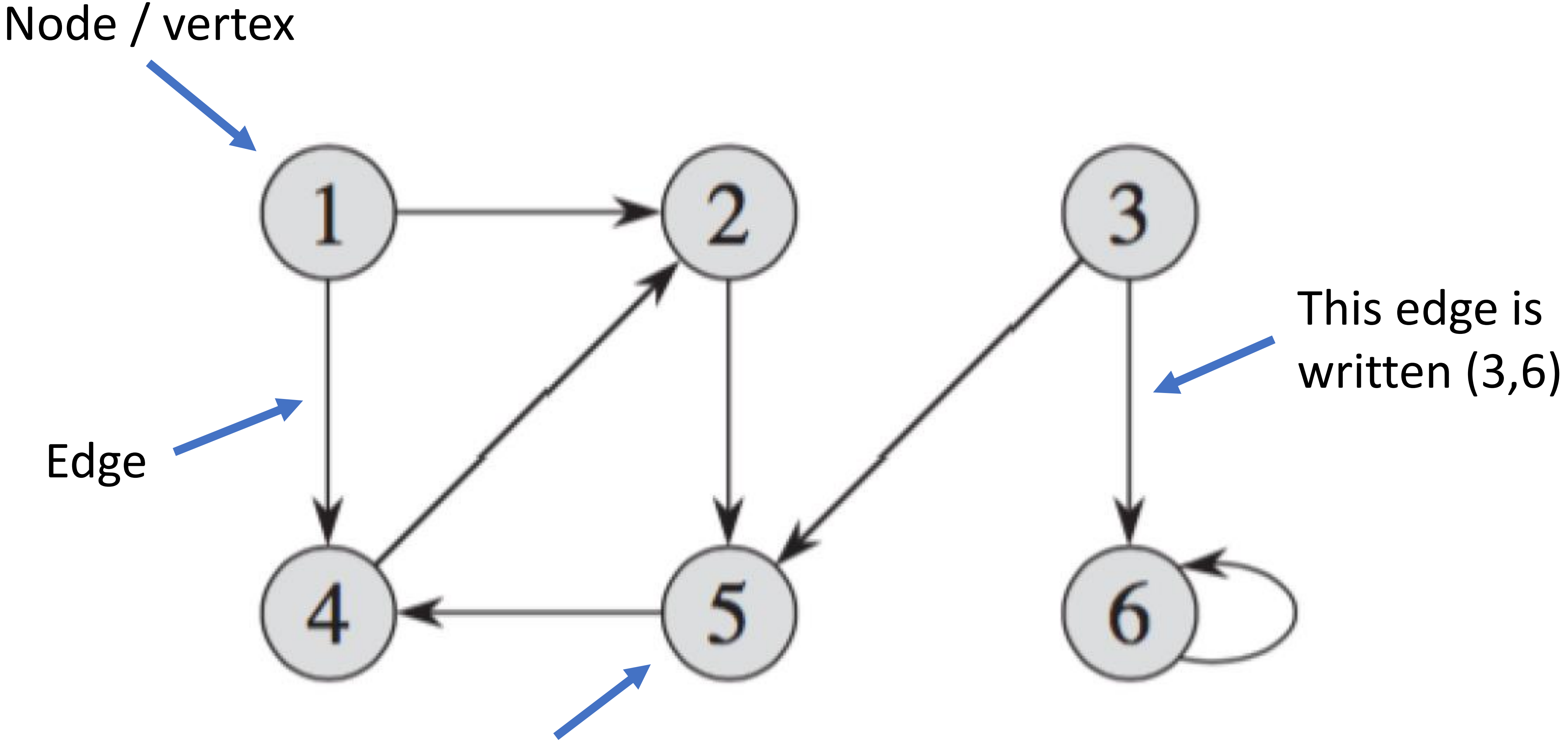
Introduction to graphs

Graph terminology



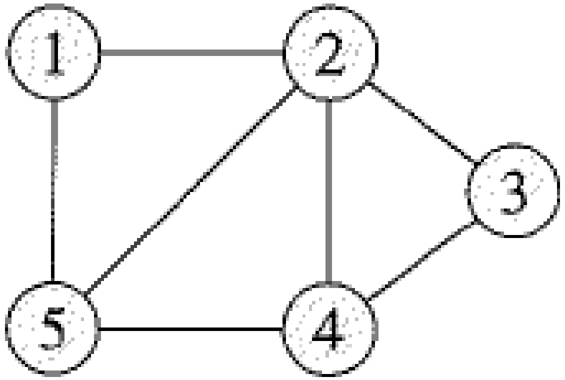
Node 5 is a **neighbor** or a **child** of node 2 because there's an edge from 2 to 5.

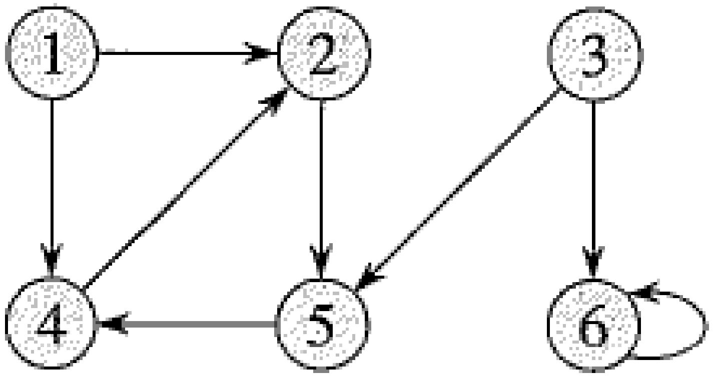
Order of Computation?

	A	B	C
1	10	20	= A1+B1
2	50	30	= A2+B2
3	= (A1+A2)/C3	= (B1+B2)/C3	= C1+C2

a) C1 C2 A3 B3 C3
b) A3 B3 C2 C1 C3
c) C2 C1 C3 B3 A3
d) Don’t know

Graphs





Undirected graphs

Directed graphs

$G = (V, E)$ graph with vertices V og edges E

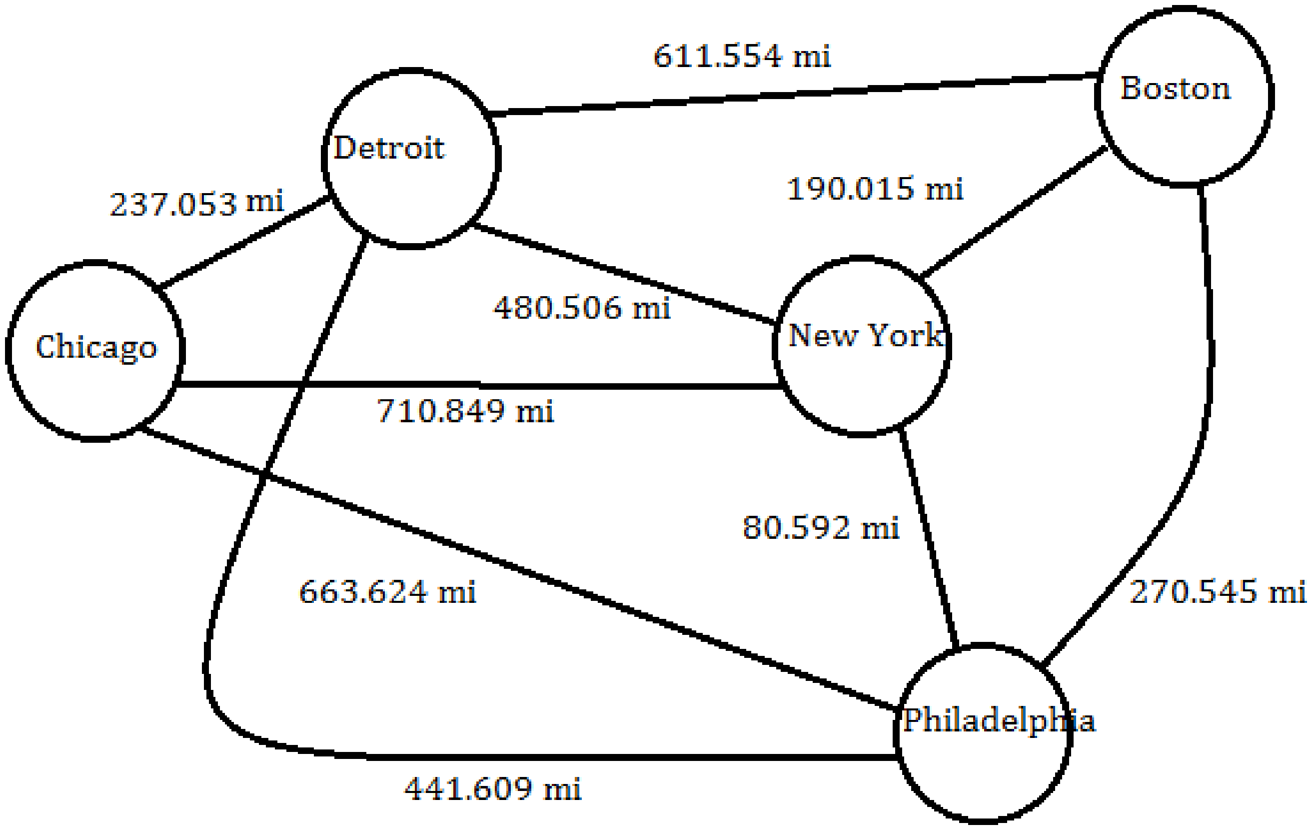
E : **$\{u, v\}$** edge between u and v in a undirected graph and

(u, v) directed edge from u to v.

$n = |V|$ = number of vertices

$m = |E|$ = number of edges (connections between vertices)

Weighted graphs

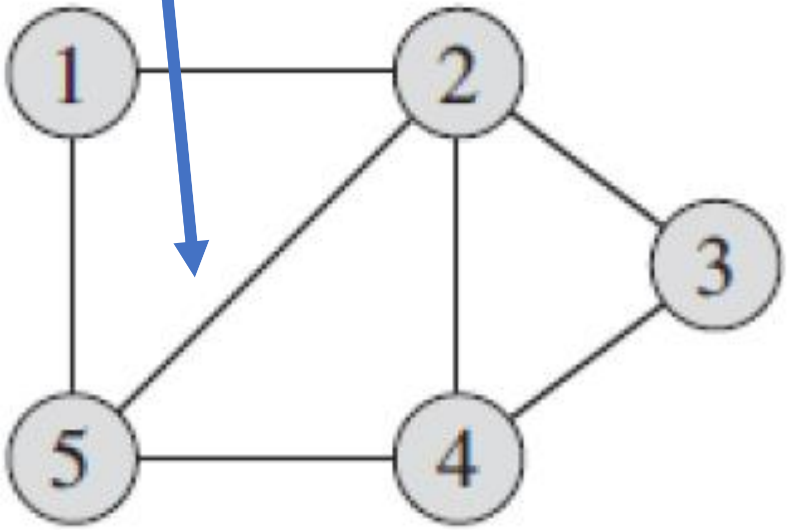


In some contexts we may want to assign a **weight** to the edges of a graph

The adjacency matrix

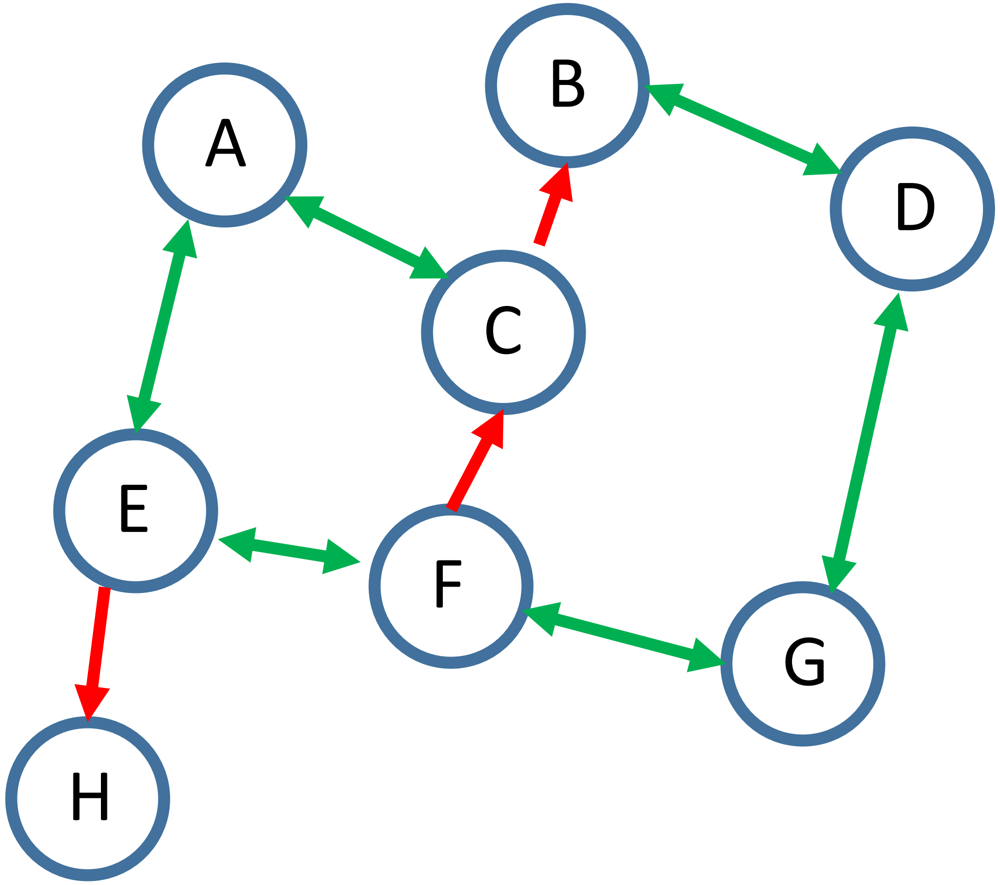
There is a connection from node 2 to node 5

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



There is no connection from node 5 to node 3

Adjacency matrix for directed graph

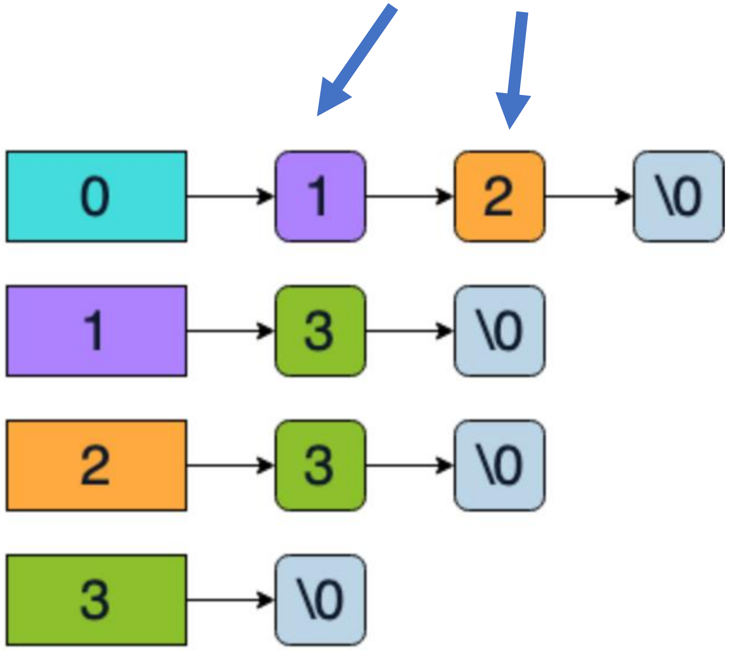


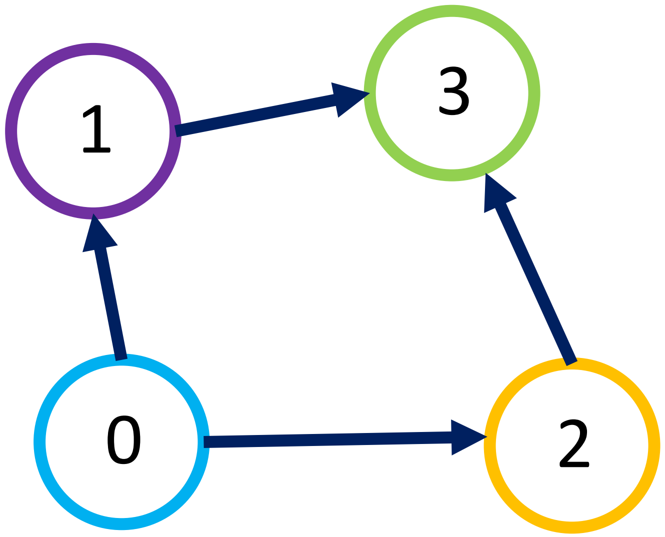
How do you think the adjacency matrix for this graph would look?

Adjacency list

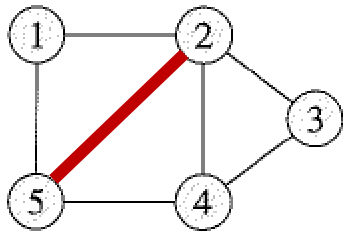
An array of linked lists. Each list contains all the neighbors of a vertex.

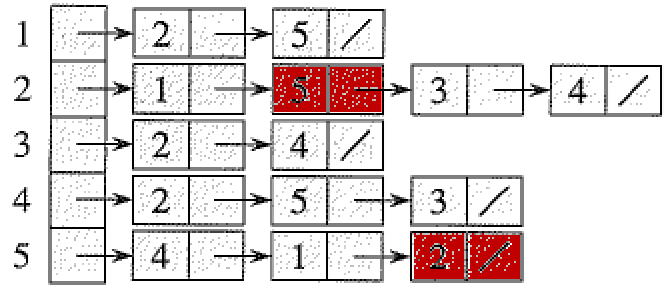
This shows that 0 is connected to 1 and 2





Representing Graphs



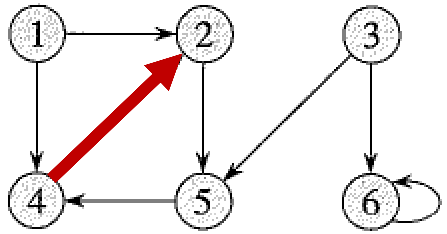


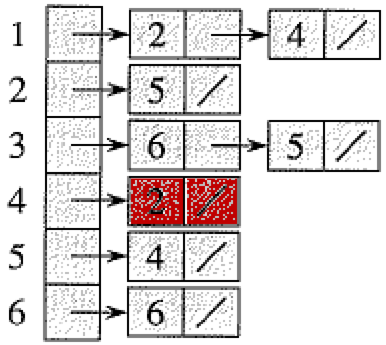
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Undirected graphs

Space O(n+m)

Space O(n^2)





	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

Directed graphs

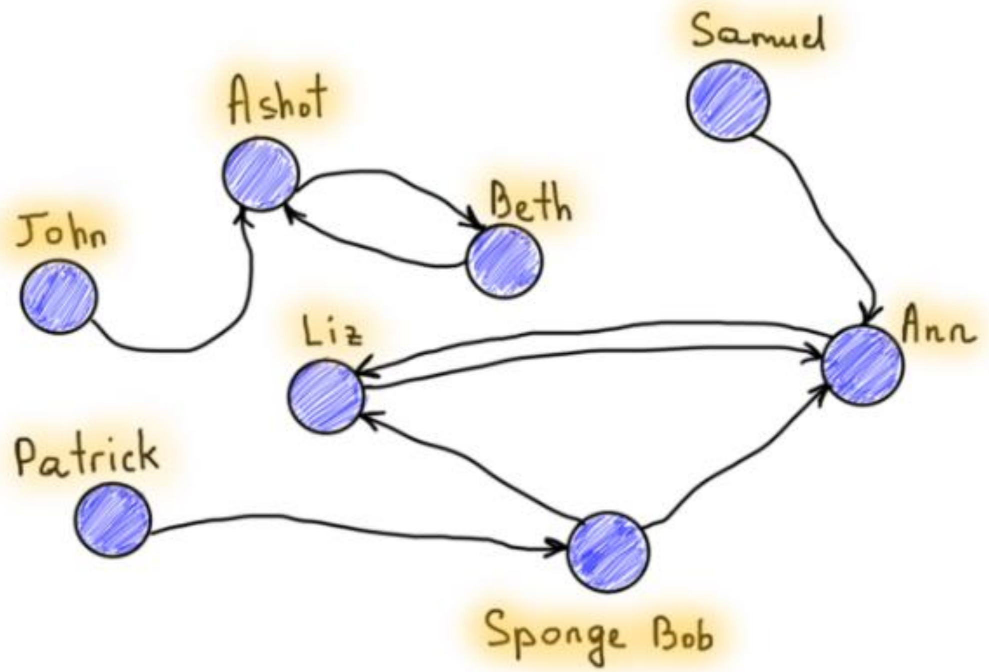
Efficiency of adjacency matrix vs adjacency list

Consider the directed graph representing who-follows-who on twitter.

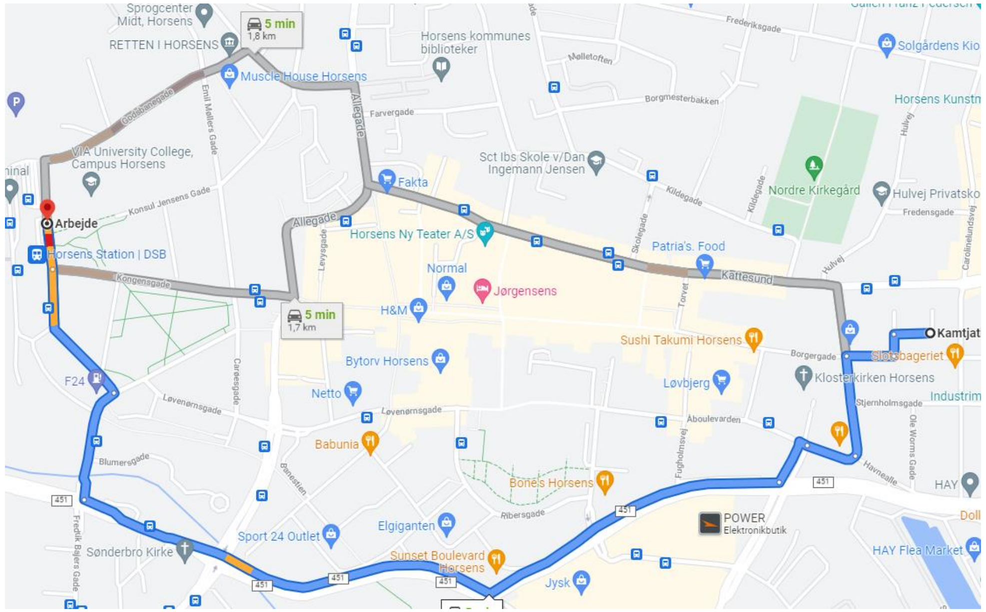
For each of the tasks below, do you think the matrix or list representation is best?

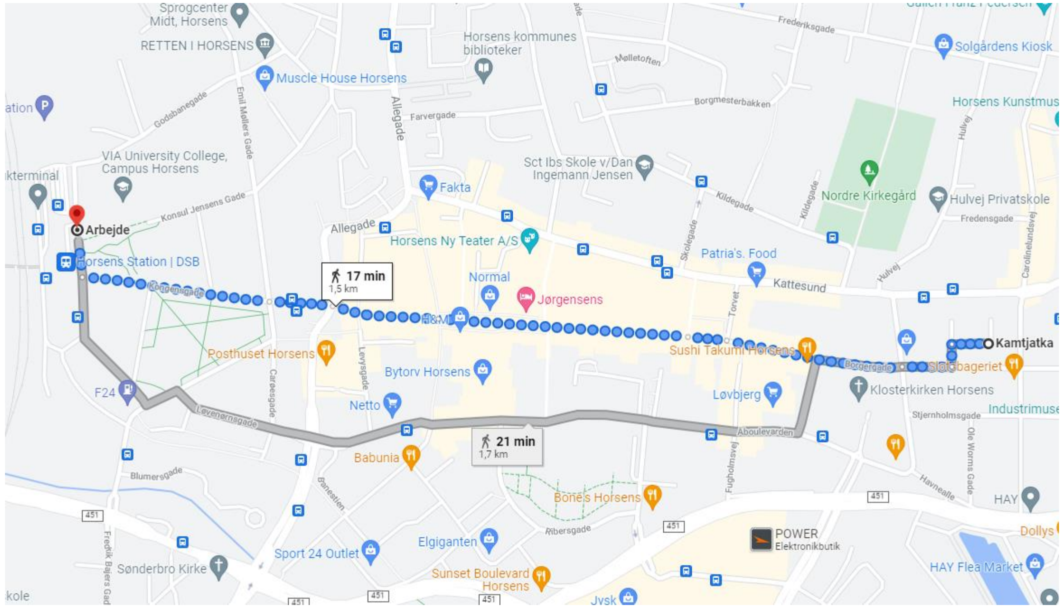
1) How many followers does Ann have?

2) Does Sponge Bob follow Beth?



Example 1

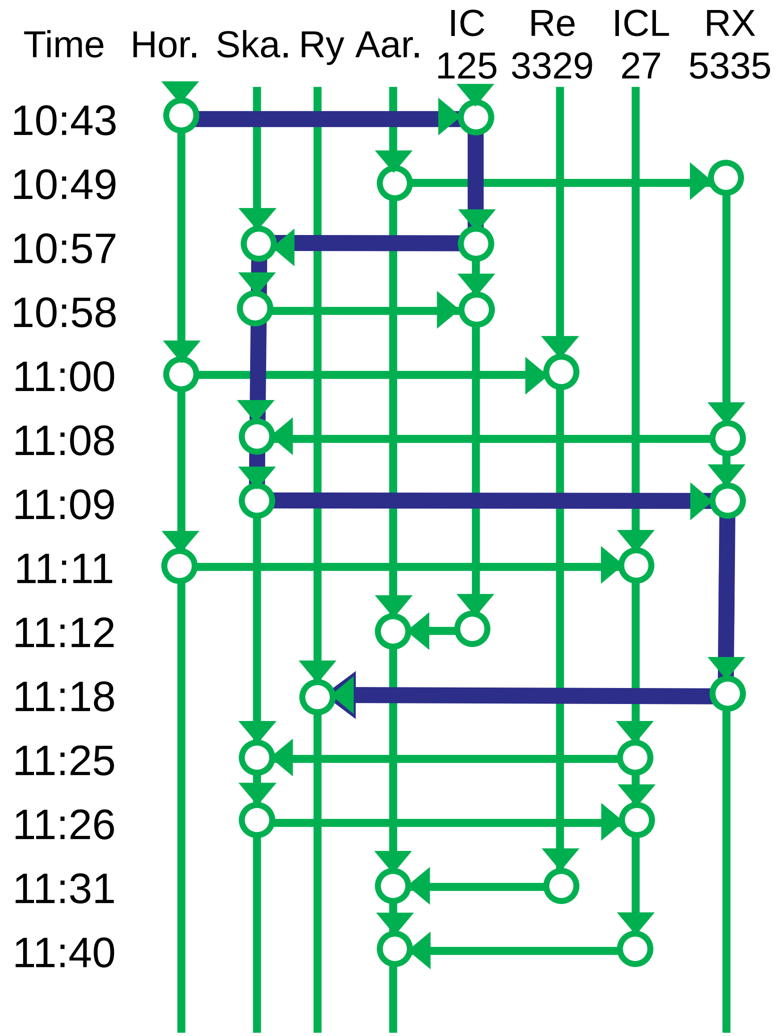

5 min
1,8 km
5 min
1,7 km
Arbejde
Kamtjat


17 min
1,5 km
21 min
1,7 km
Arbejde
Kamtjatka

How many vertices/nodes do you need to properly represent a four-way intersection?

a) 1
b) 2
c) 4
d) 5
e) 8
f) 9
g) 12
h) Don’t know

Example 2: Itenary (Horsens to Ry)

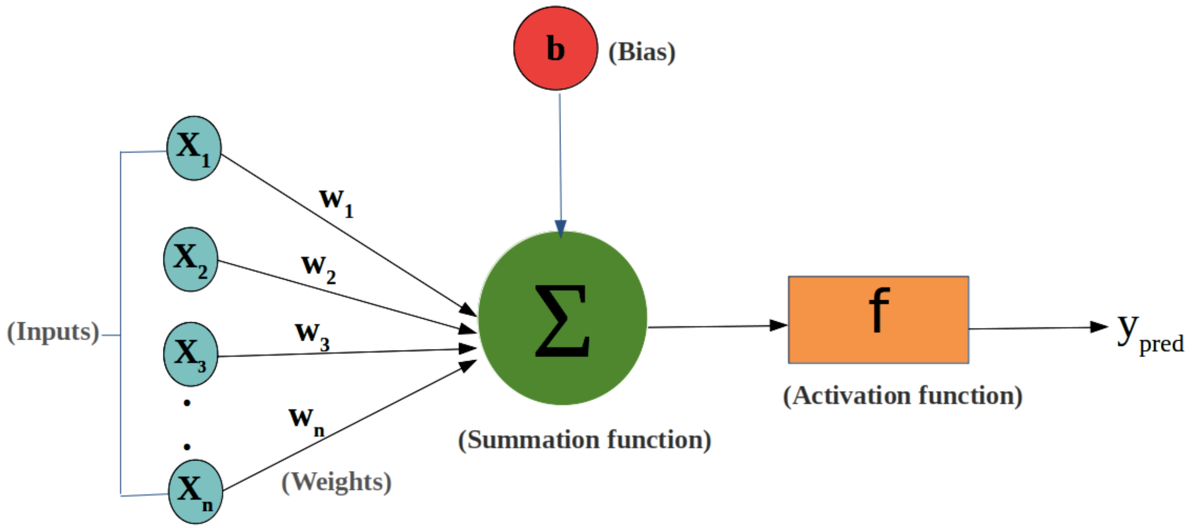


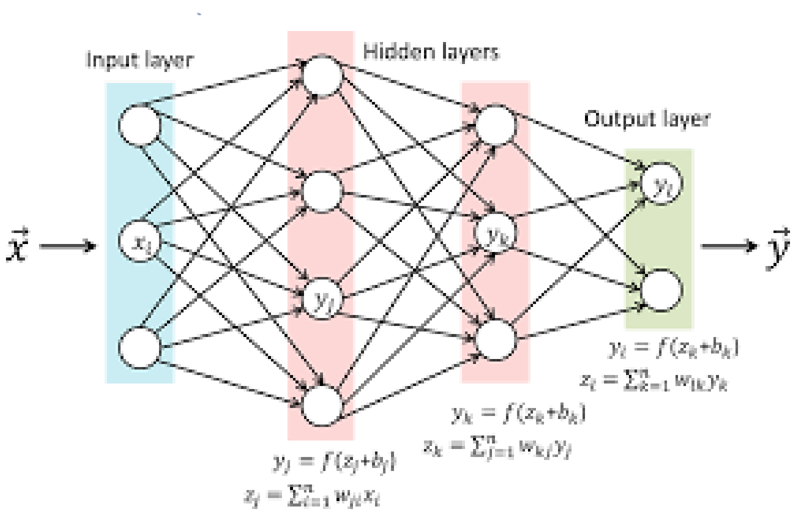
Algorithm
Find earliest vertex for Ry that can be reached from a start vertex in Horsens

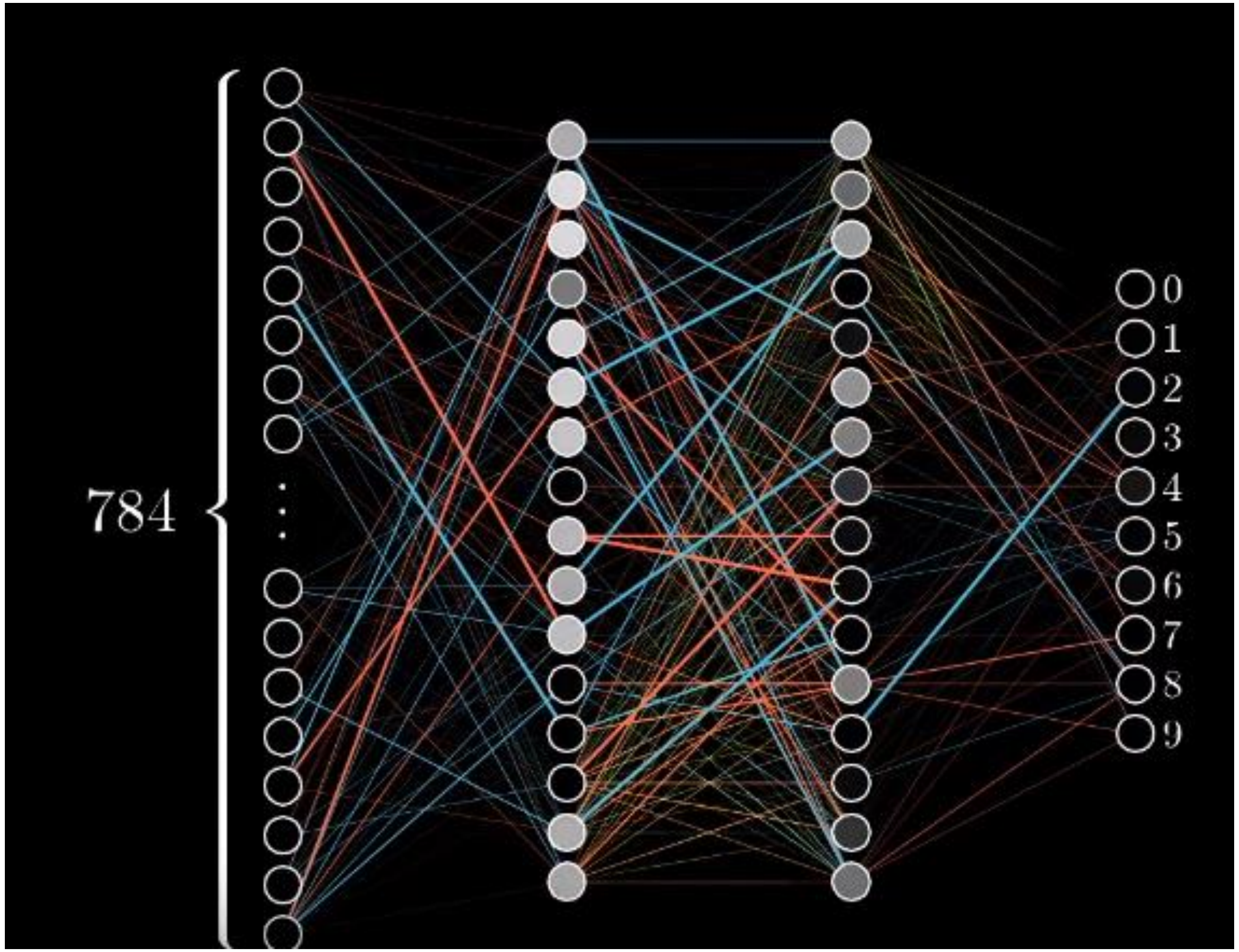
Train	Arr	Dep	Station
		10:43	Horsens
IC125	10:57	10:58	Skanderborg St
	11:12		Aarhus H
		11:00	Horsens
Re3329	11:31		Aarhus H
		11:11	Horsens
ICL27	11:25	11:26	Skanderborg St
	11:40		Aarhus H
		10:49	Aarhus H
RX5335	11:08	11:09	Skanderborg St
	11:18		Ry St

Travel schedule

Example 3: Neural Networks

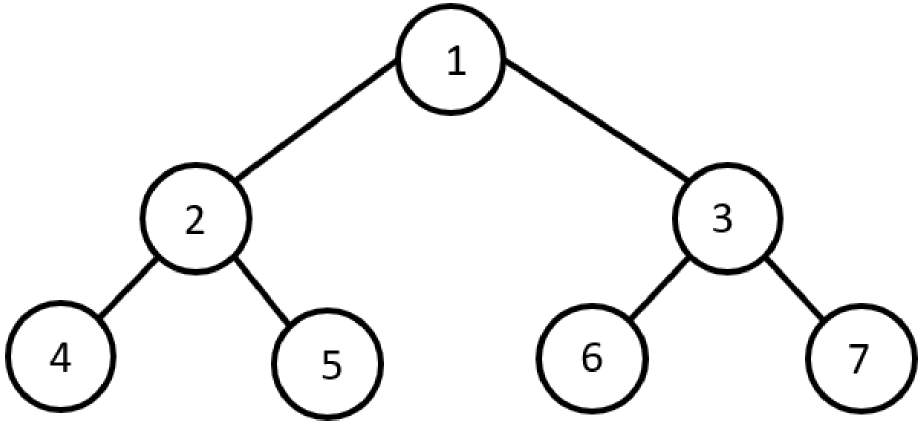






Searching a graph

Breadth First vs. Depth First

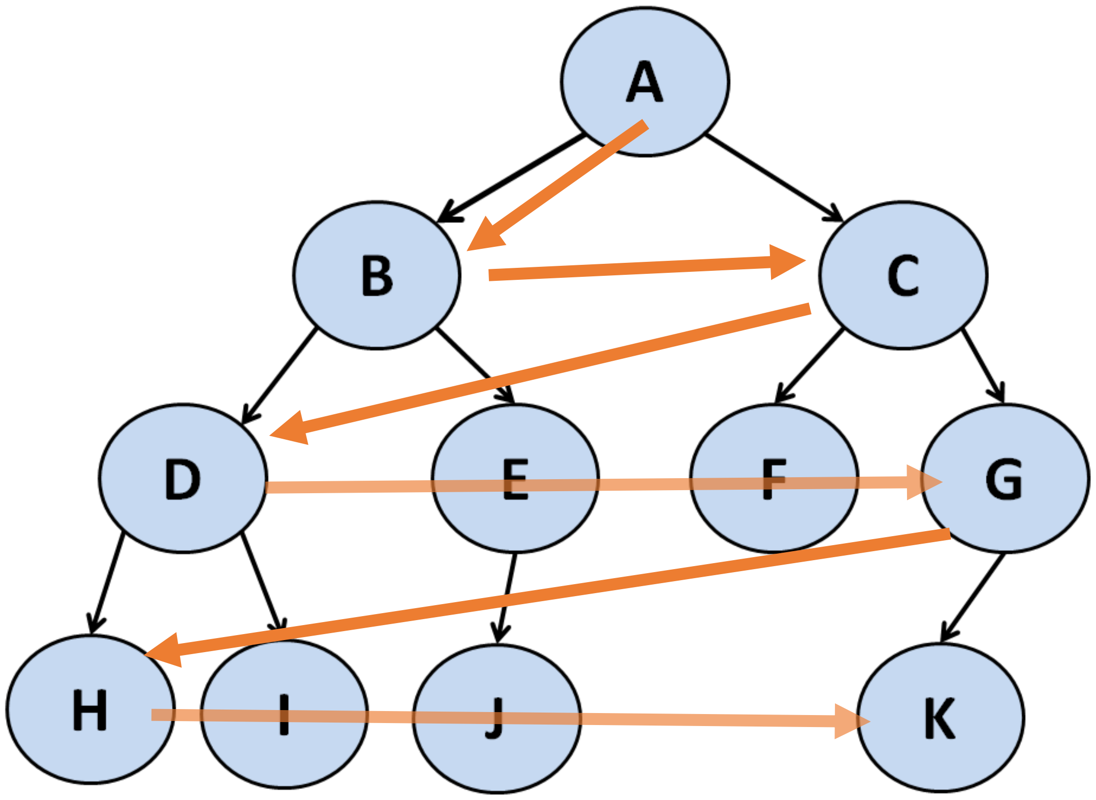


BFS: 1, 2, 3, 5, 6, 7 → Like level-order

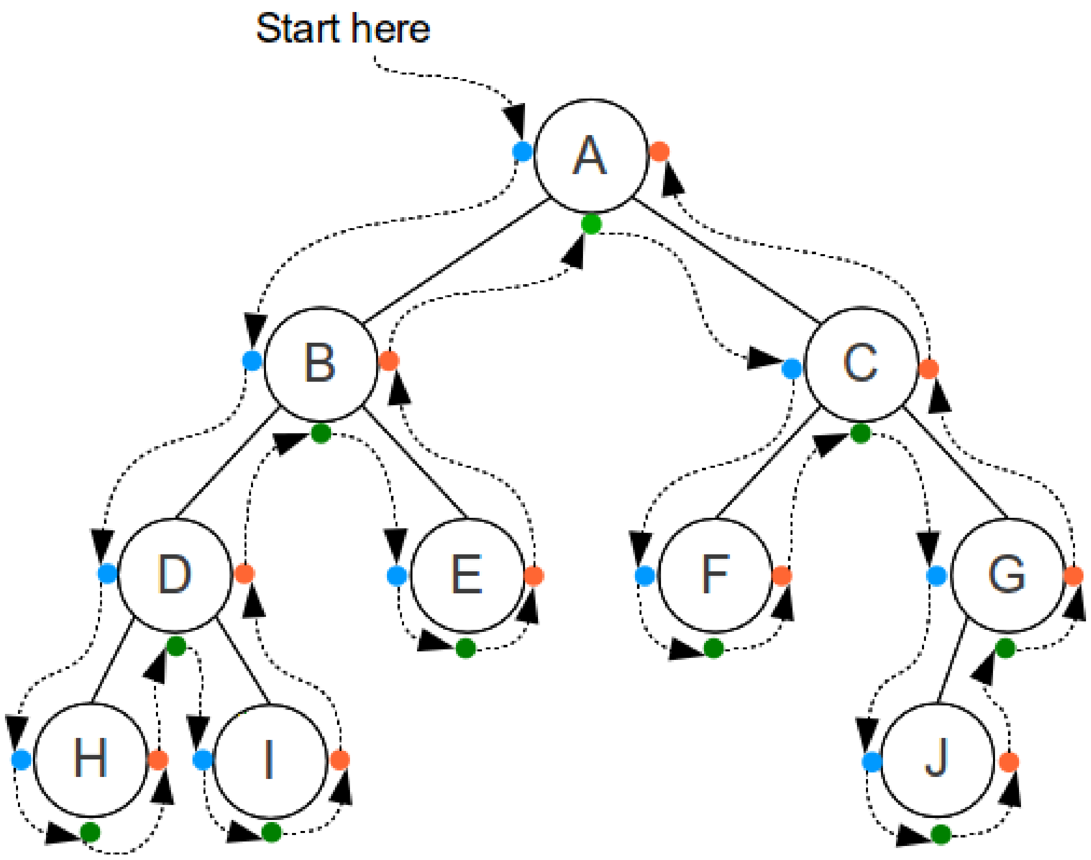
DFS: 1, 2, 4, 5, 3, 6, 7 → Like pre-order

“Depth First” vs “Breadth First” in binary search trees

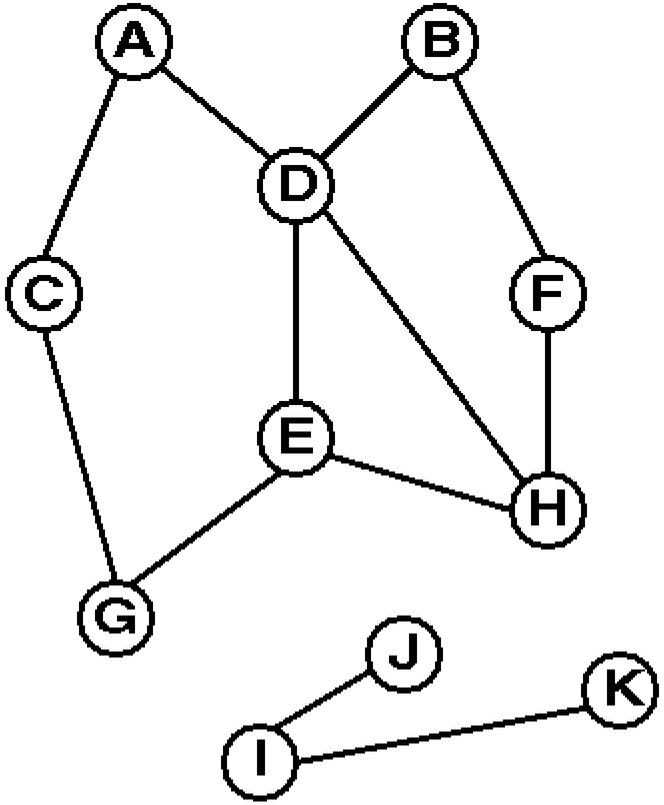
Breadth First



Depth First



Depth first search: Is there a path from A to G?



Recursive:

A asks each of its neighbors whether they have a path to G. Each of these asks each of their own neighbors, and so on.

It could for example go like this:

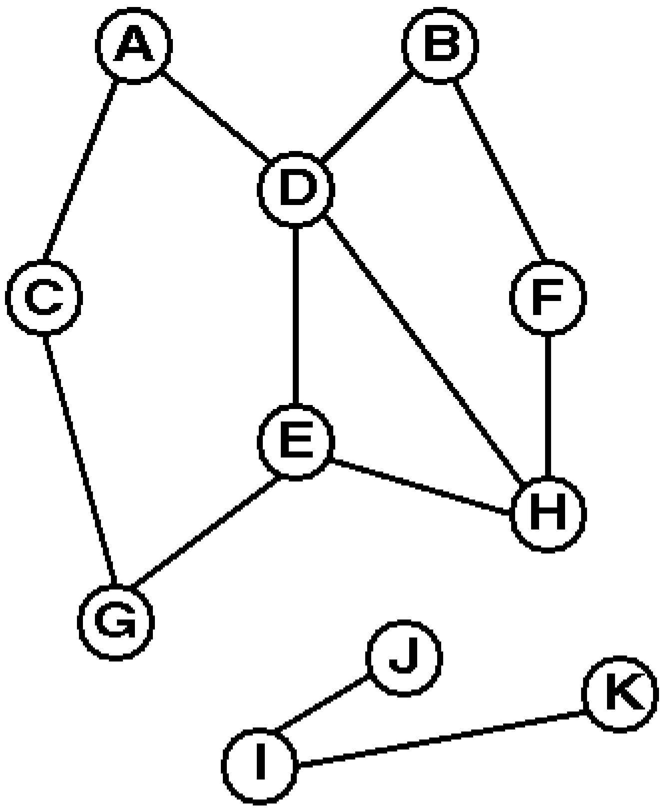
hasPath(A, G)? Yes!

hasPath(D, G)? Yes!

hasPath(E,G)? Yes!

hasPath(G,G)? Yes!

Breadth first search: Is there a path from A to G?



A asks each of its neighbors whether they are directly connected to G. Each return an answer to A imediately. If no path was found, A ask all of its neighbors neighbors whether they are directly connected to G, and so on.

It could for example go like this:

Level 0:

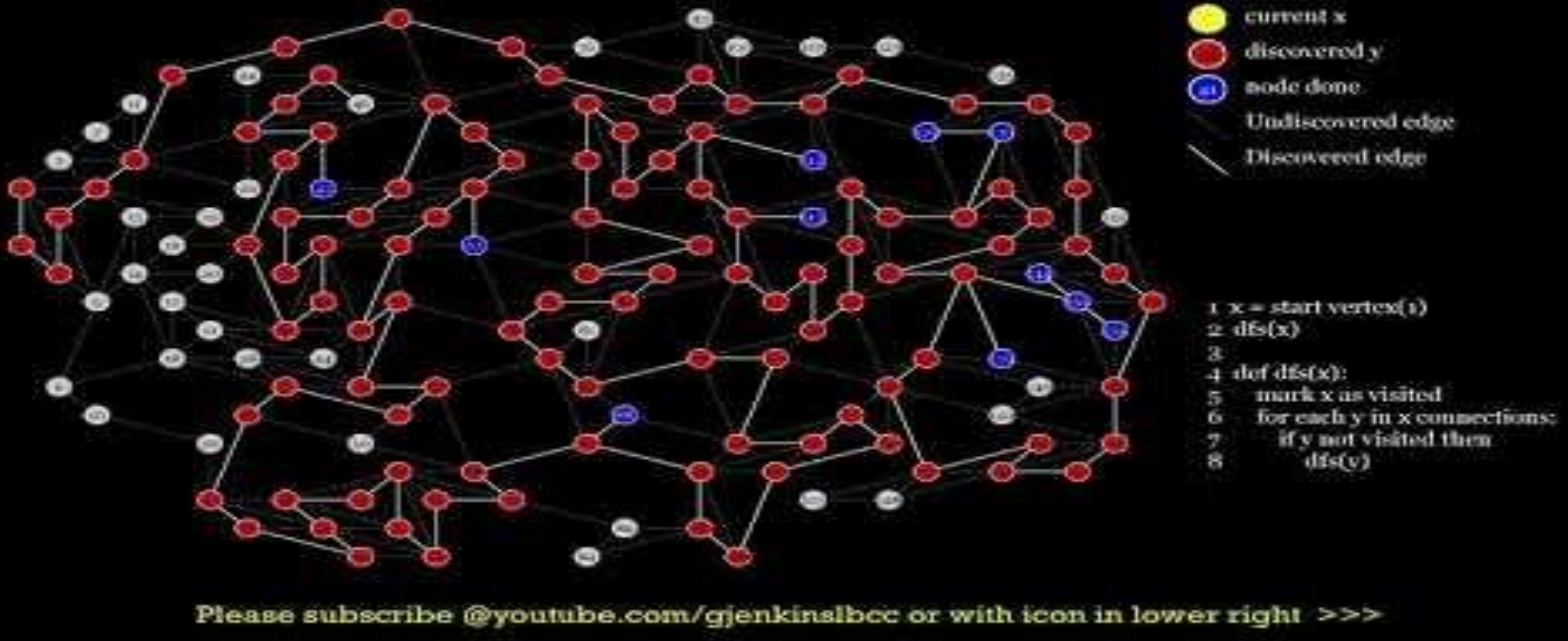
hasPath(A, G)? No!

Level 1:

hasPath(D, G)? No!

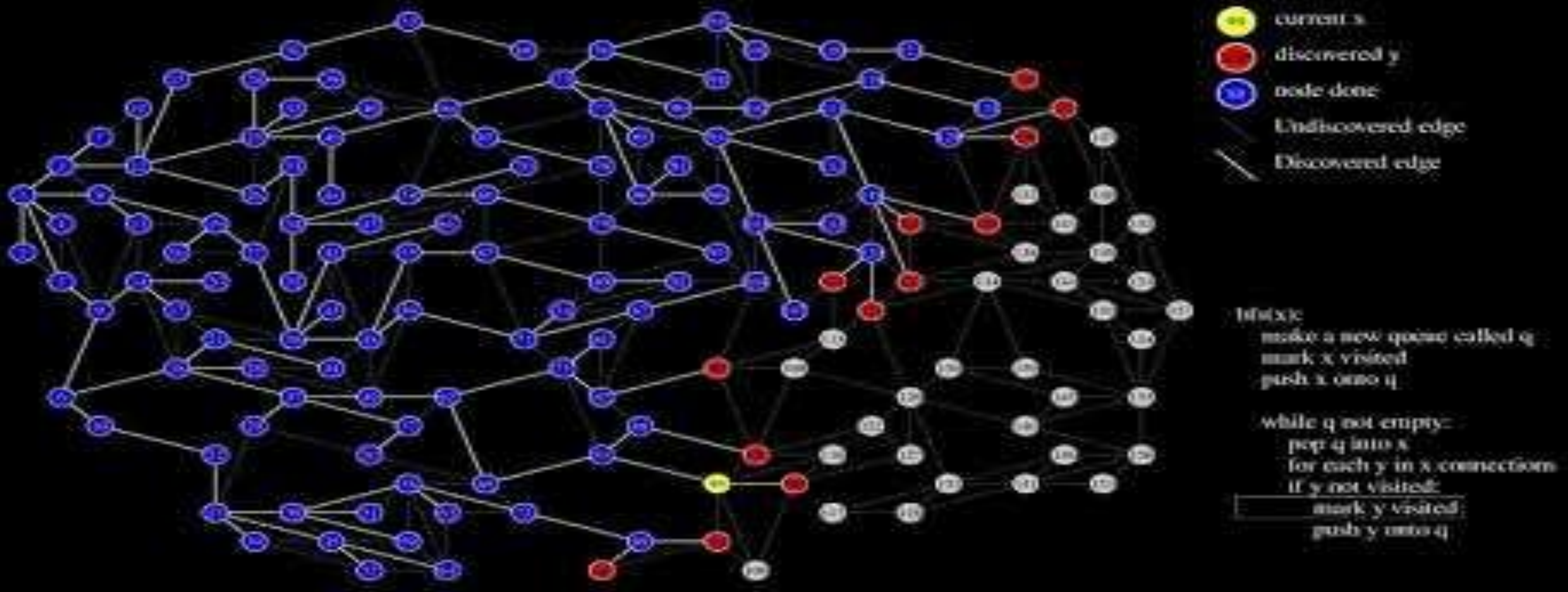
hasPath(C, G)? Yes!

Depth first search animation



Source: https://www.youtube.com/watch?v=NUgMa5coCoE

Breadth first search animation



Source: https://www.youtube.com/watch?v=x-VTfcmrLEQ

Implementation of Breadth First Seach (BFS)

BFS (Breadth first search) algorithm

```
Mark all vertices white, except the start node s, which is grey.
Add s to an empty queue Q.
while Q is nonempty:
    node = Dequeue(Q)
    for each neighbor in Adj[node]:
        if neighbor.color is white:
            neighbor.color = gray
            Enqueue(Q, neighbor)
    node.color = black
```

Breadth-First Search from CLRS

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

Color all nodes except s white (= “unexplored”) and set all node’s parents to NIL and set the distance to s to ∞

Color s gray (“discovered”), set the distance from s to s to 0, and set s’s parents to NIL.

Create an empty queue and add s to the queue.

Visit the front node (u) in the queue

Visit all unexplored child-nodes of u

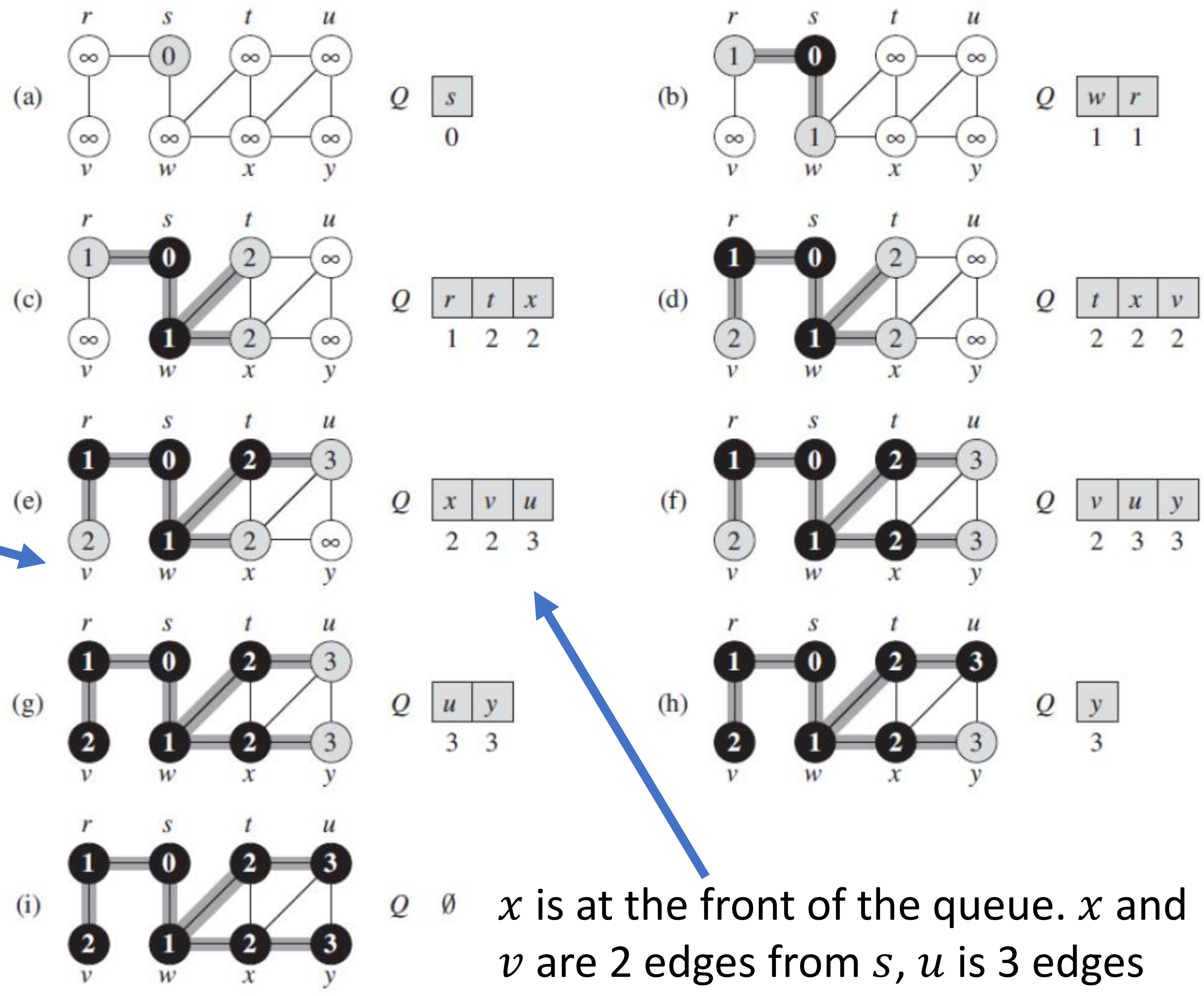
Color each of the child-nodes grey, record their distance to s and list u as their parent

Add all of the child-nodes to the queue

Color u black (= “done”)

BFS-example

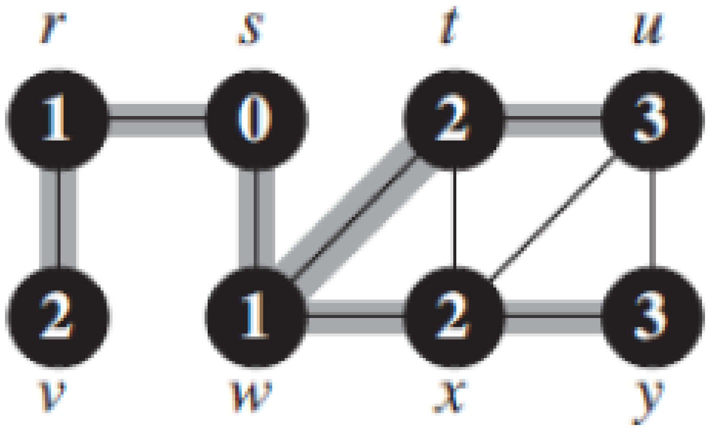
This means that node v is found at a distance of 2 edges from s.



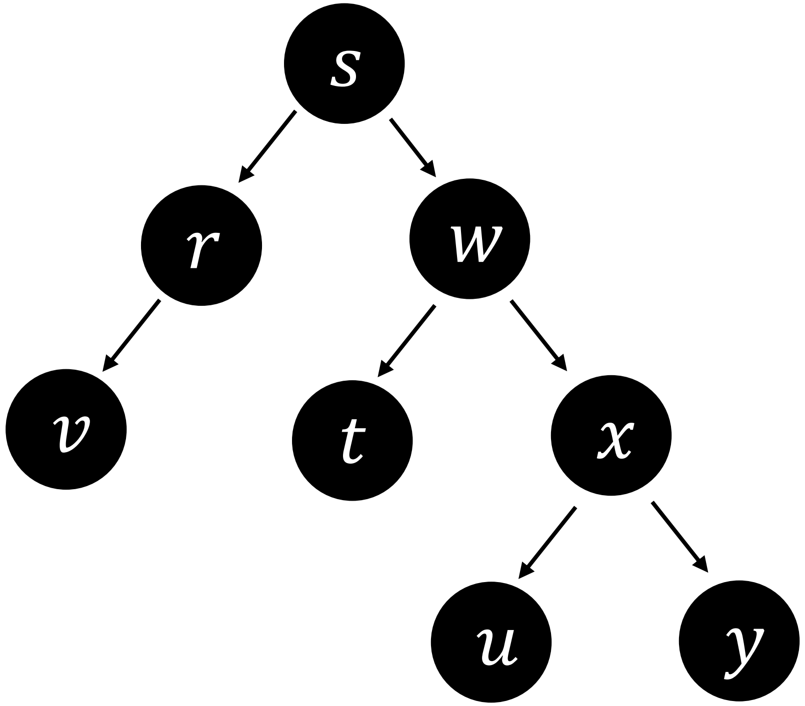
x is at the front of the queue. x and v are 2 edges from s, u is 3 edges from s.

BFS search tree

The “Breadth First Search” from the previous slide can be represented in a tree-structure:



=



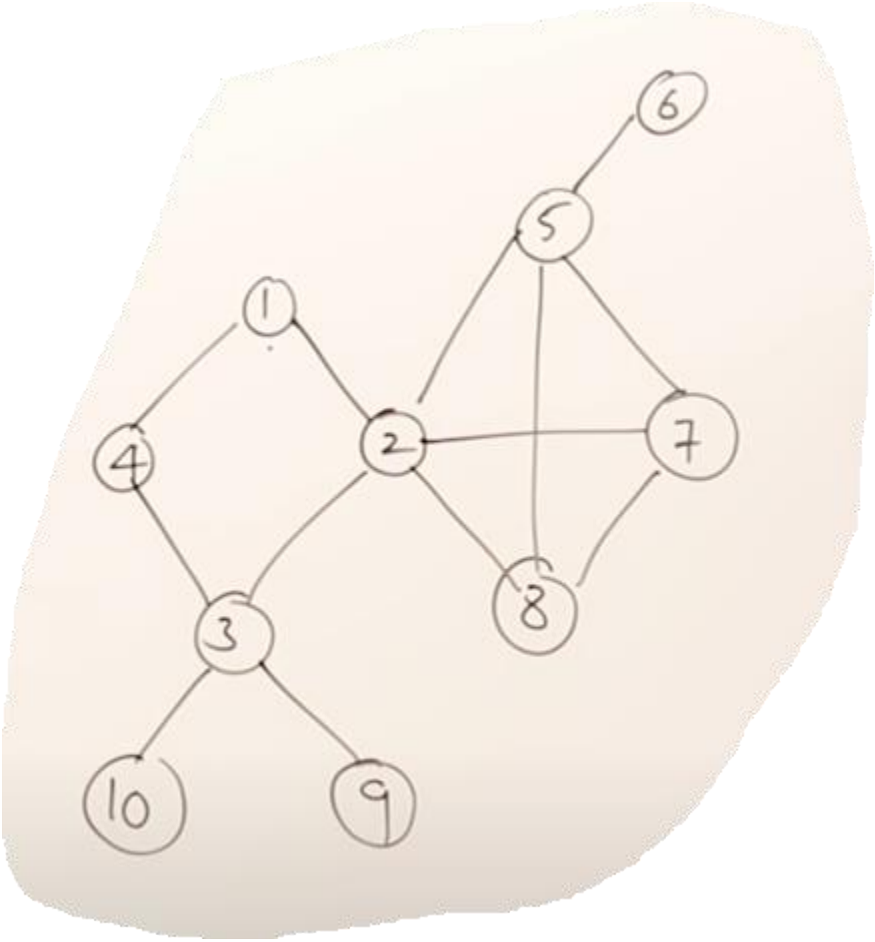
The search can be described by noting in which order the edges are passed (note not literal order in below list). A legal BFS could be

$$(s,r)\ (s,w)\ (w,t)\ (r,v)\ (w,x)\ (x,y)\ (x,u)$$

BFS

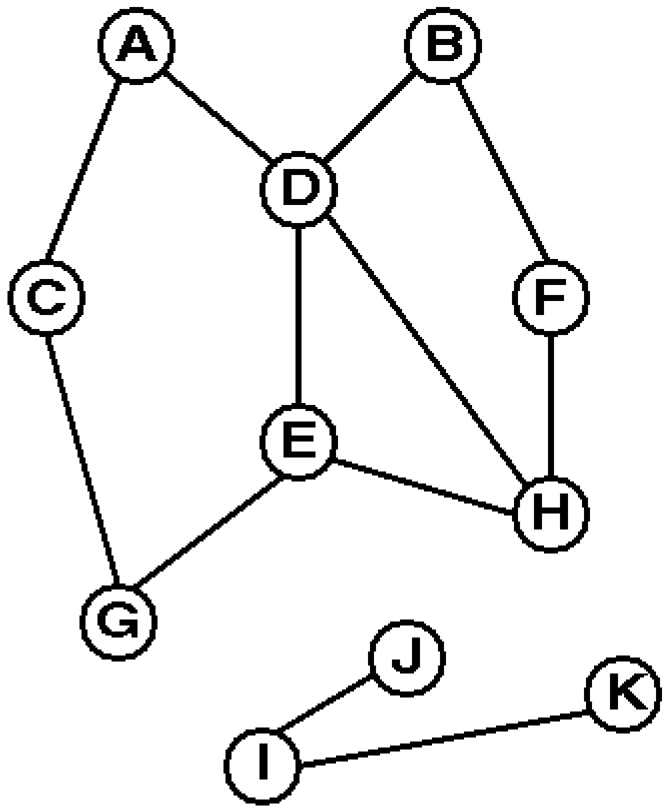
Q

Result:



Implementation of Depth First Seach (DFS)

Keeping track…



We need to keep track of which nodes are *unexplored*, which nodes are *discovered* (but haven't returned an answer yet) and which nodes are *done***.**

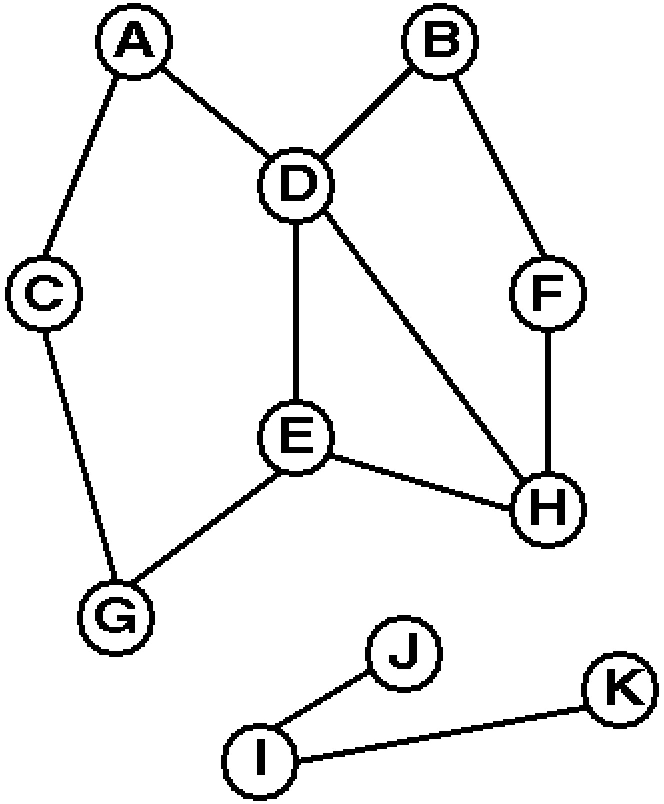
A is discovered hasPath(A, G)?

D is discovered hasPath(D, G)?

E is discovered hasPath(E,G)?

G is discovered hasPath(G,G)?

Keeping track…



We need to keep track of which nodes are *unexplored*, which nodes are *discovered* (but haven't returned an answer yet) and which nodes are *done*.

A is discovered A is done hasPath(A, G)? Yes!

D is discovered D is done hasPath(D, G)? Yes!

E is discovered E is done hasPath(E,G)? Yes!

G is discovered G is done hasPath(G,G)? Yes!

Depth-first search from CLRS

```
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)
```

Color all nodes white (= “unexplored”) and set all nodes’ parents to NIL

Keep track of time for each path (assume each visit takes 1 time unit)

Visit all unexplored nodes

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

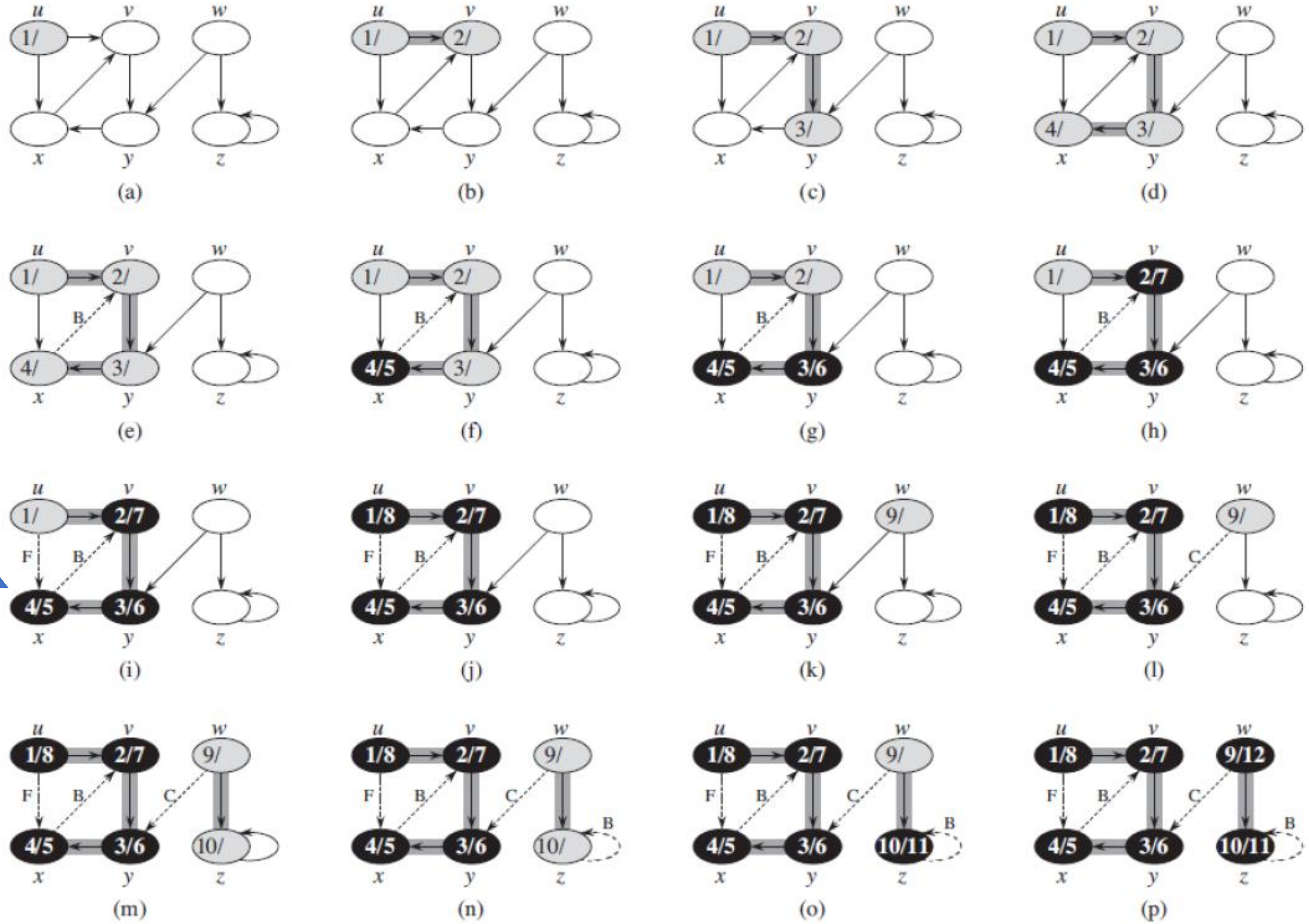
Visiting node u → increment time and save start-time
color node grey (= “discovered”)

Visit all unexplored child-nodes of u, and set their parent to be u.

For each child, recursively visit all their children

Color u black (= “done”) and record finish-time

DFS-example



This means that node x was discovered in step 4 and "done" in step 5

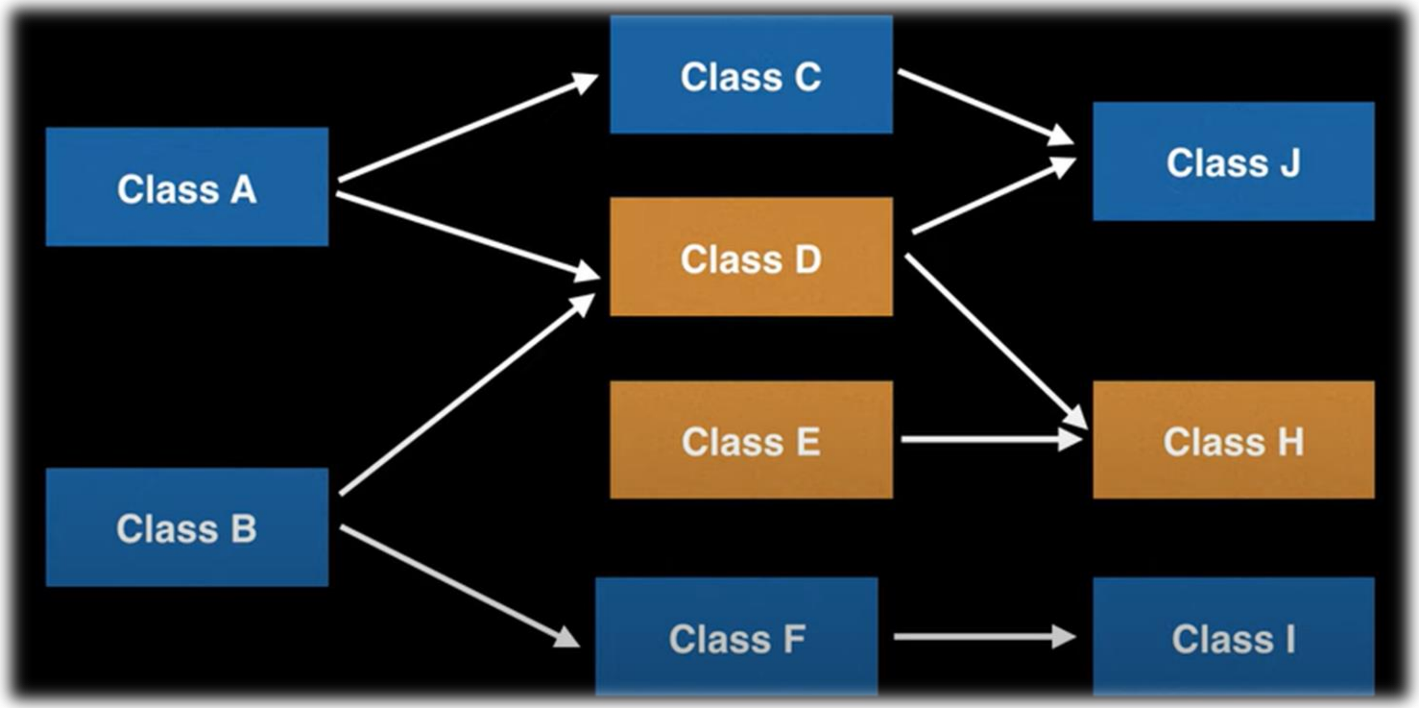
B: Back edge
F: Forward edge
C: Cross edge
Grey: Tree edge

Ordering in directed graphs

Ordering in directed graphs

The arrows in a directed graph represent an *ordering* of the nodes. This can for example be used to keep track of

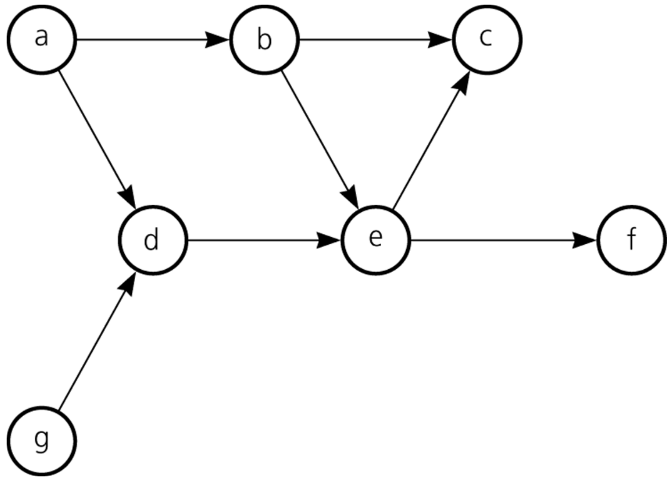
- course dependencies
- program dependencies
- task dependencies



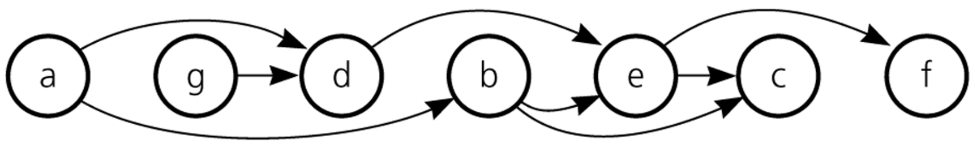
Source: https://www.youtube.com/watch?v=eL-KzMXSXXI&t=28

Topological Sort

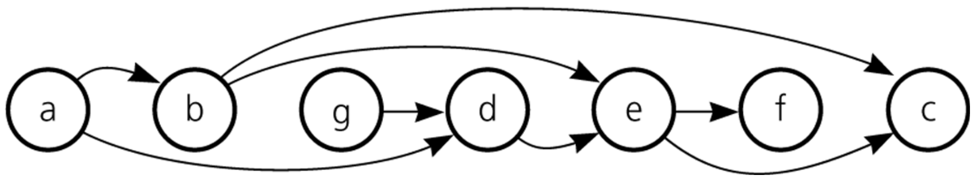
- Directed graph G.
- Rule: if there is an edge u → v, then u must come before v.



(a)



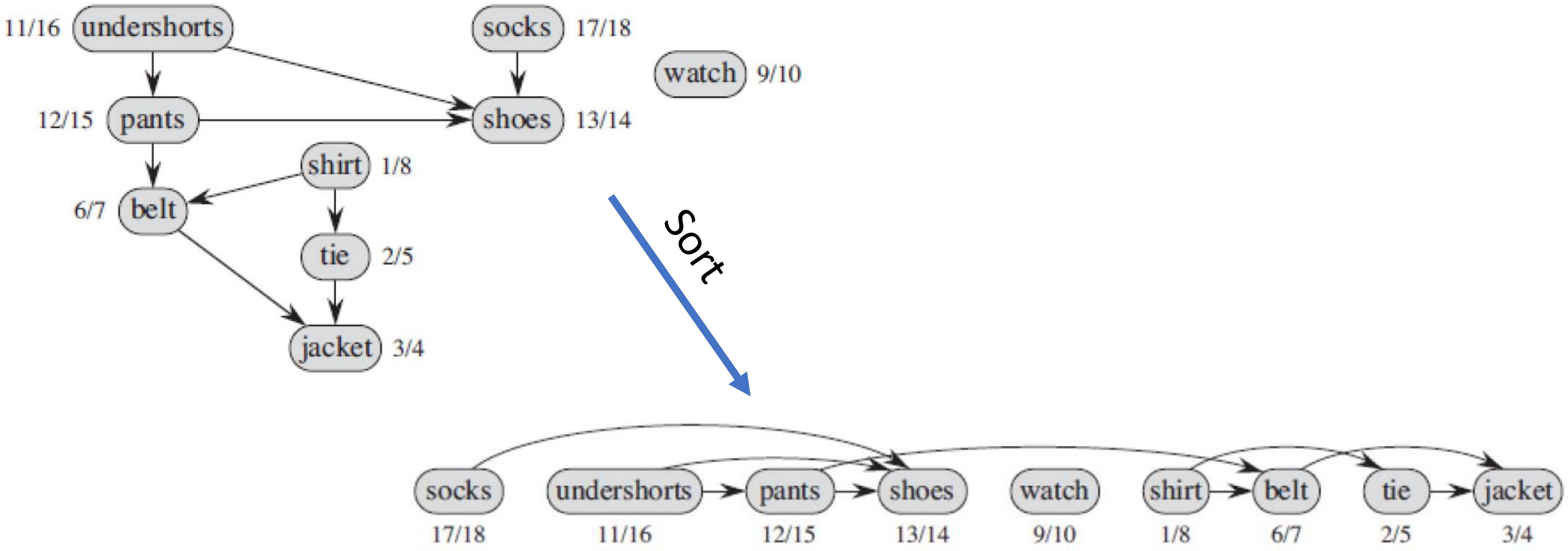
(b)



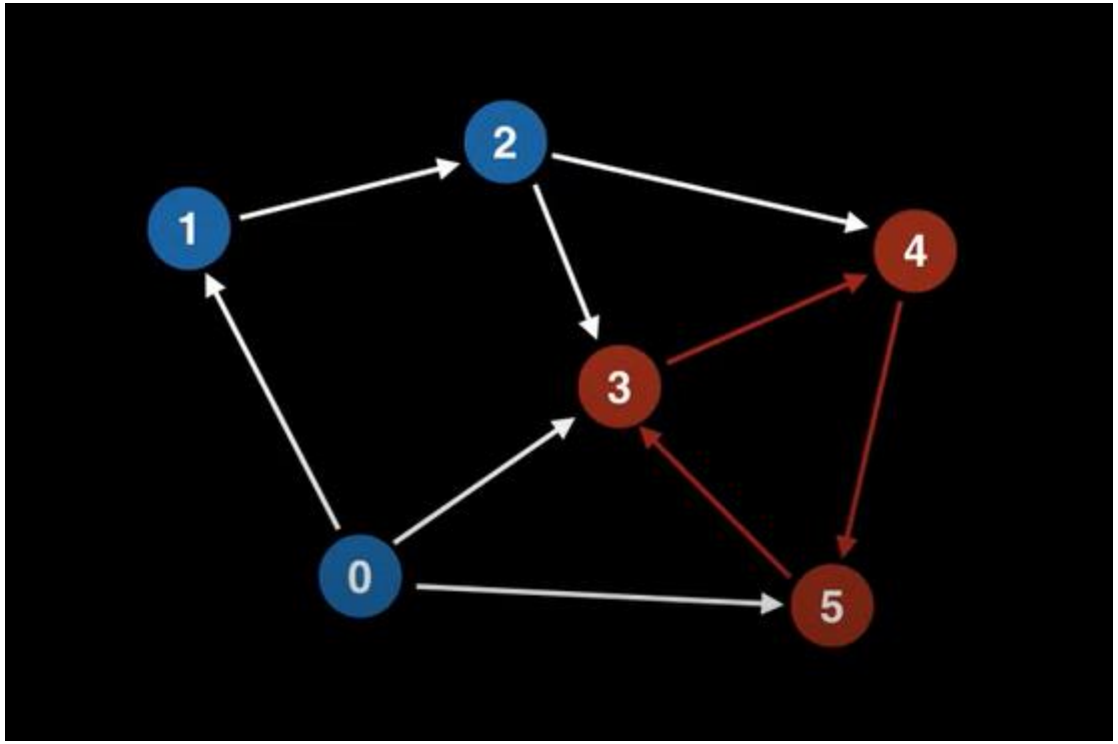
Sorting a directed graph

From the graph below, it is not immediately clear what should be done first.

We need a sorting algorithm for graphs!



Not all graphs can be topologically sorted



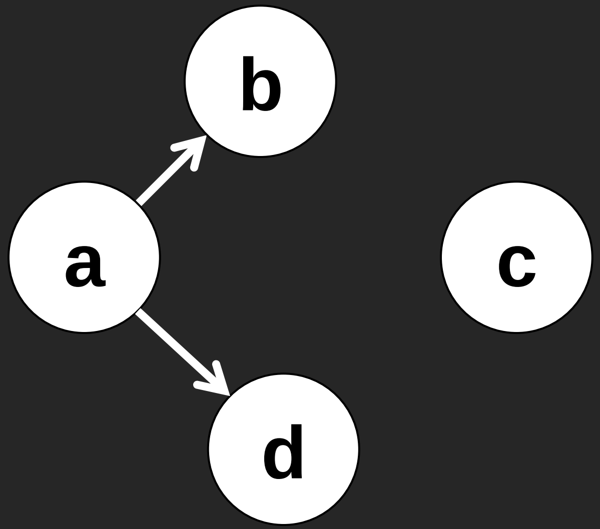
If the graph has a cycle, it is impossible to do a topological sorting.

But all other directed graphs can be topologically sorted!

A directed graph with no cycles is called a "Directed Acyclic Graph" (DAG)

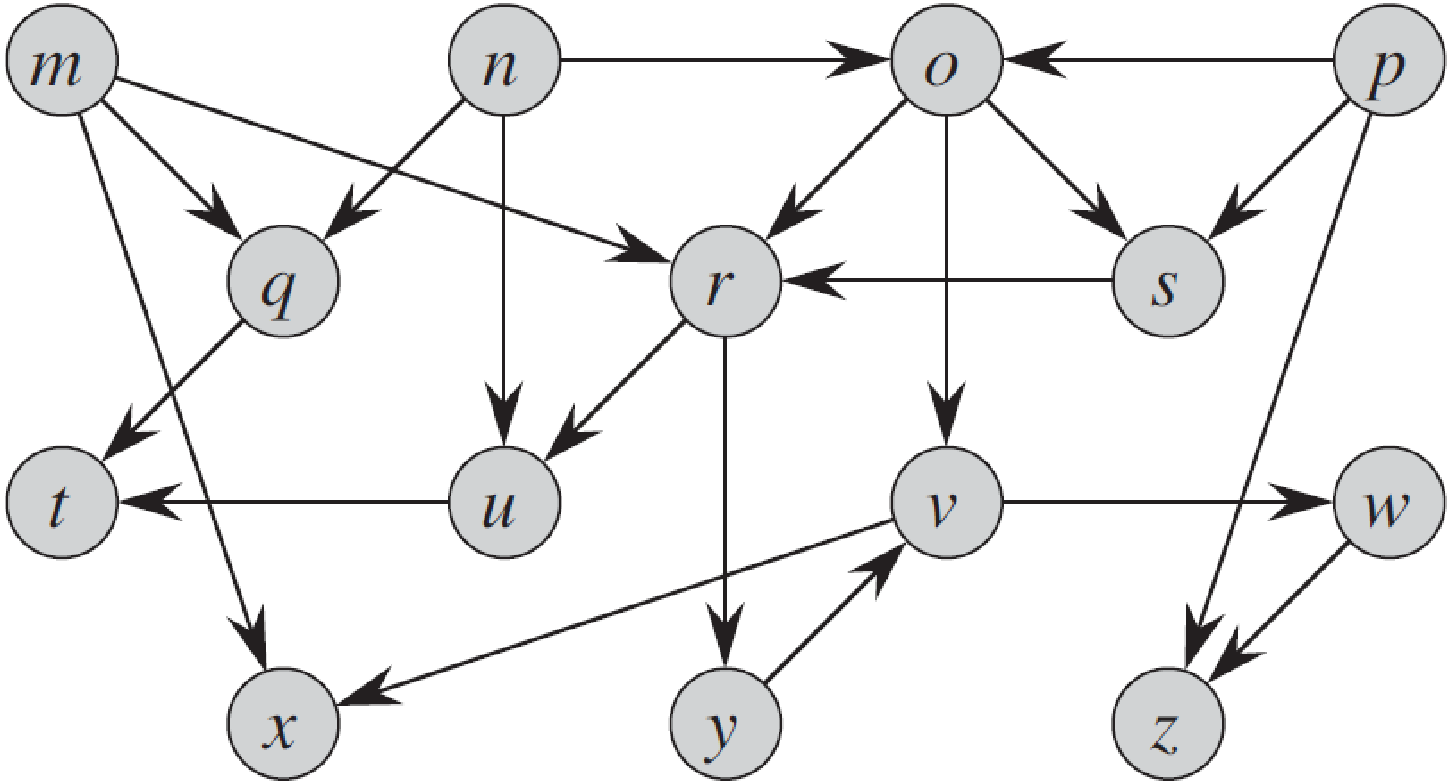
In how many ways can you topologically sort the graph?

a) 1
b) 2
c) 3
d) 4
e) 5
f) 6
g) 7
h) 8
i) 9
j) Don't know



Topological Sort NOT based in DFS

- Delete a Vertex with in degree 0 and add to the end of topological order



Topological Sort based in DFS

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