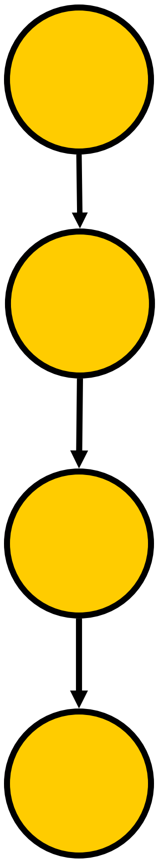
CS1102: Lecture 12

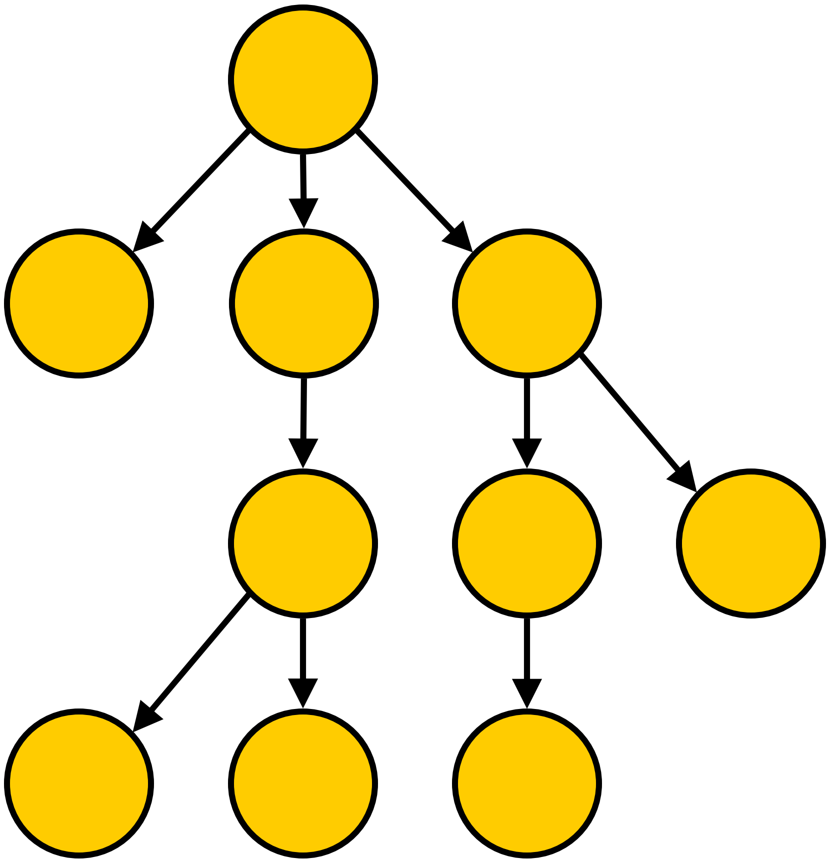
Graphs

Chapter 14: pages 735 - 769

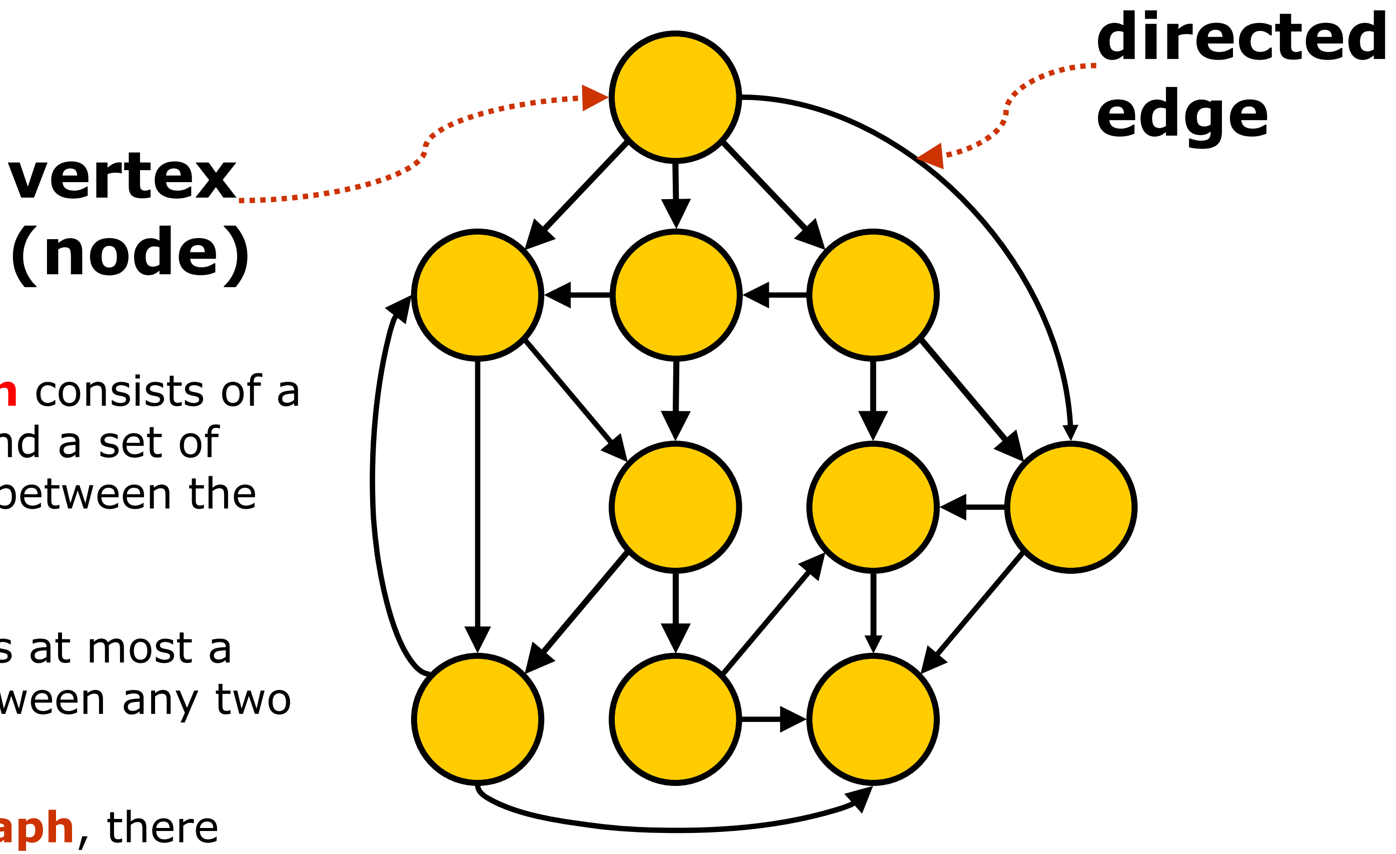
Linked list



Tree



Directed graph

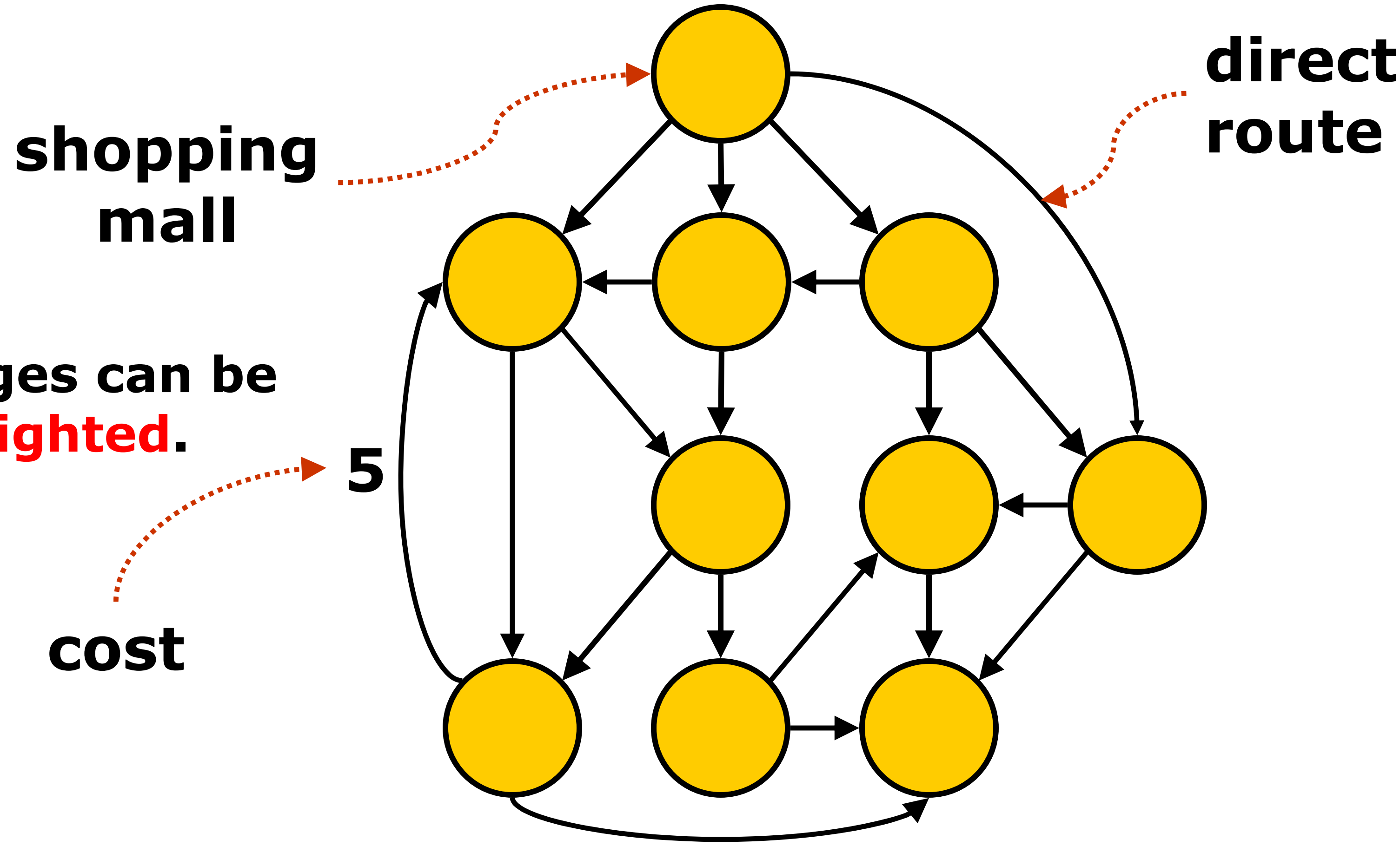


A **directed graph** consists of a set of **vertices** and a set of **directed edges** between the **vertices**.

In a **tree**, there is at most a **unique path** between any two nodes.

However, in a **graph**, there may be more than one path between two nodes.

Example: travel planning

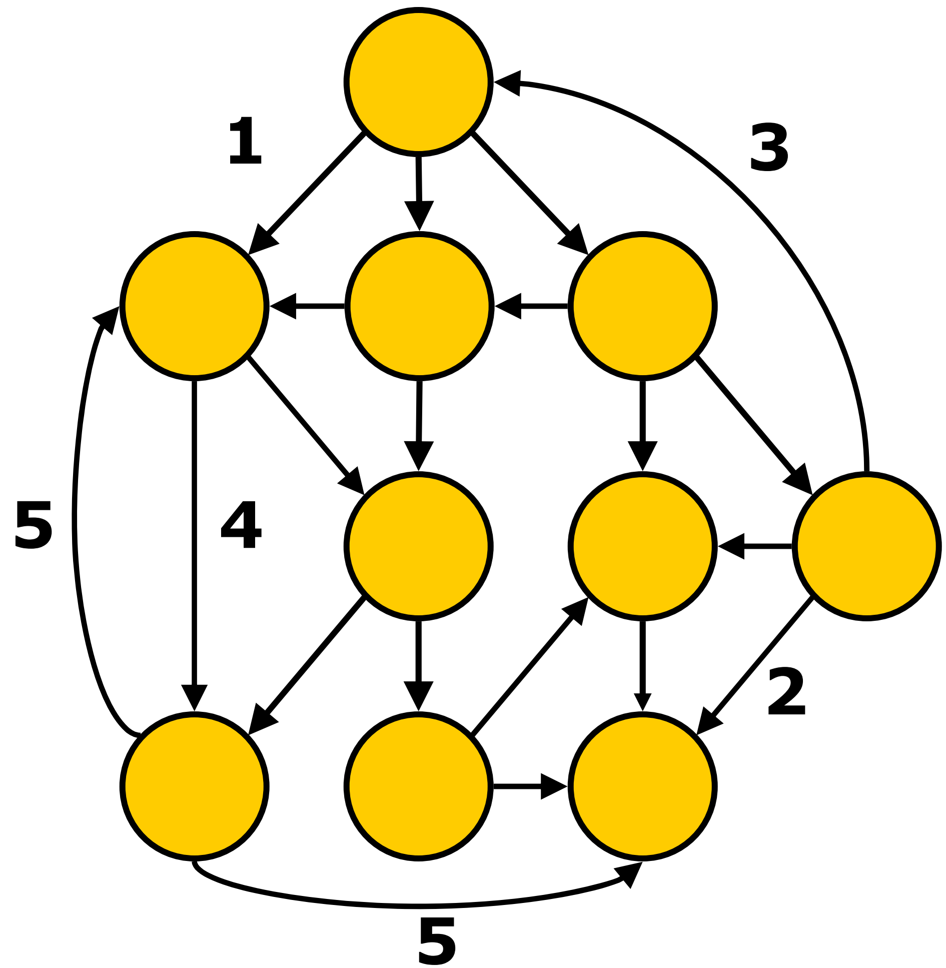


Edges can be **weighted**.

Weighted directed graph

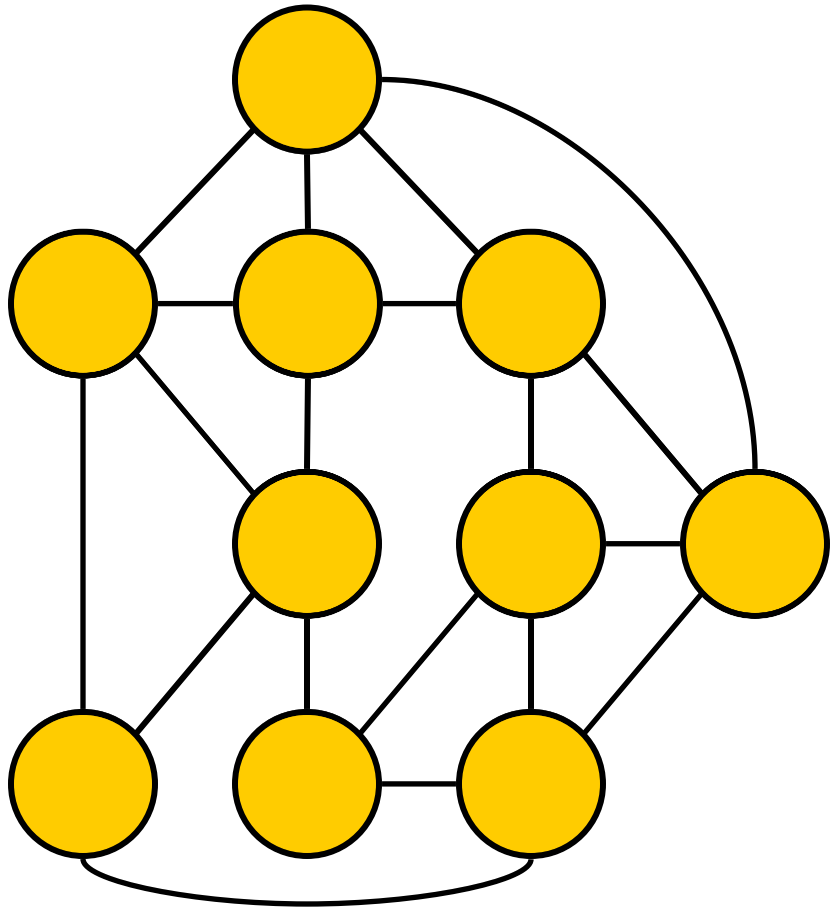
In a **weighted graph**, **edges** have a **weight** (or **cost**) associated with it.

Not all weights are labeled in this slides for simplicity.

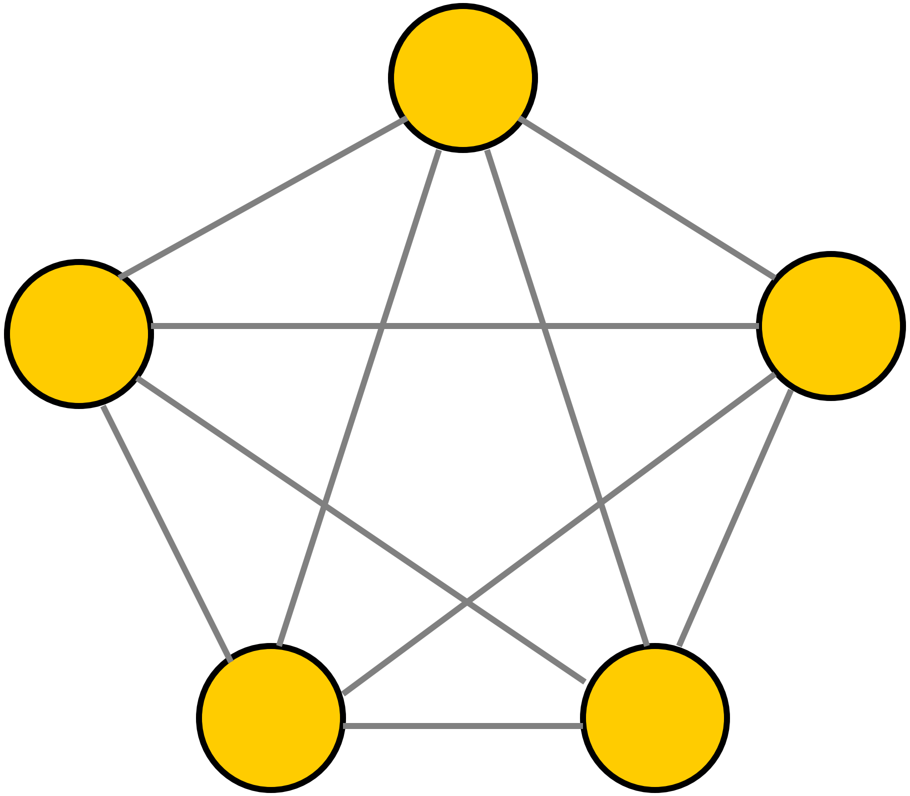


Undirected graph

In an **undirected** graph, edges are **bidirectional**



Complete graph



A graph is **complete** if every pair of vertices has an edge between them.

The number of edges in a complete graph is **n(n-1)/2**, where n is the number of vertices.

So, the number of edges is **$O(n^2)$**.

A complete graph is also called a **clique**.

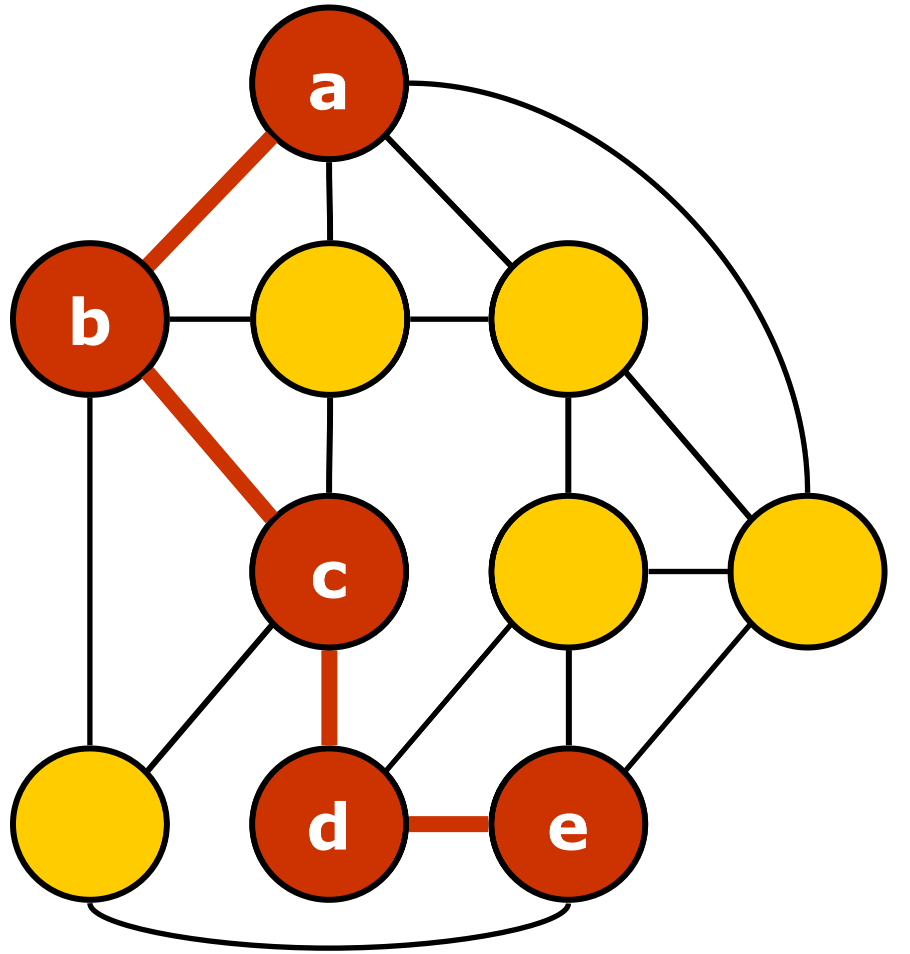
Path

A **path** between two vertices is a sequence of edges that begin at one vertex and end at another.

The **length** of a path p is the number of edges in p.

E.g. the length of the highlighted path is 4.

A **simple path** never visits the same vertex more than once.

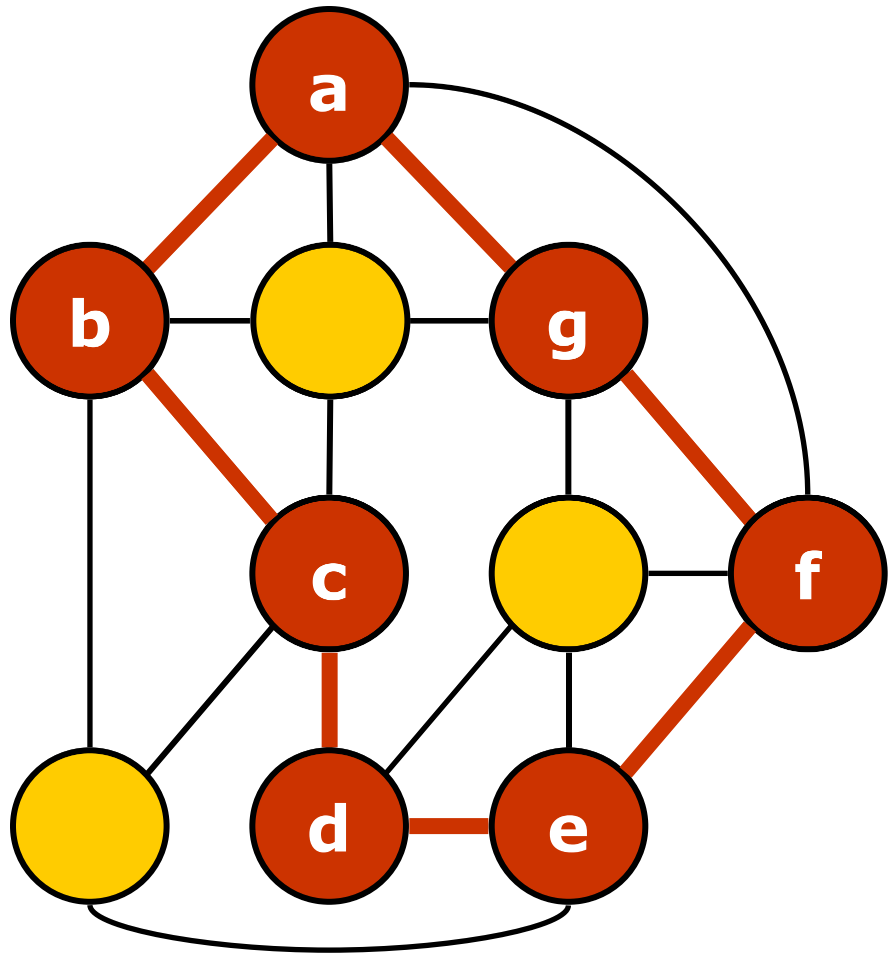


Cycle

A **cycle** is a path that begins and ends at the **same vertex**.

A **simple cycle** is a simple path that is a cycle.

Note that the definitions of path and cycle apply to **directed graph** as well.

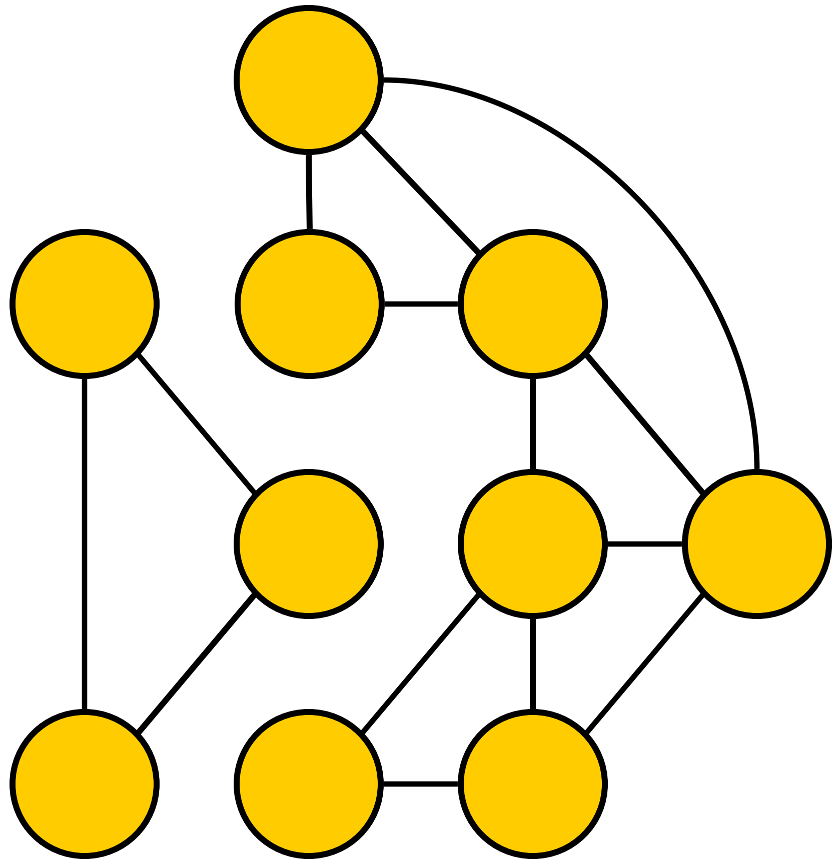


Disconnected graph

A graph is a **connected graph** if each pair of distinct vertices has a path between them.

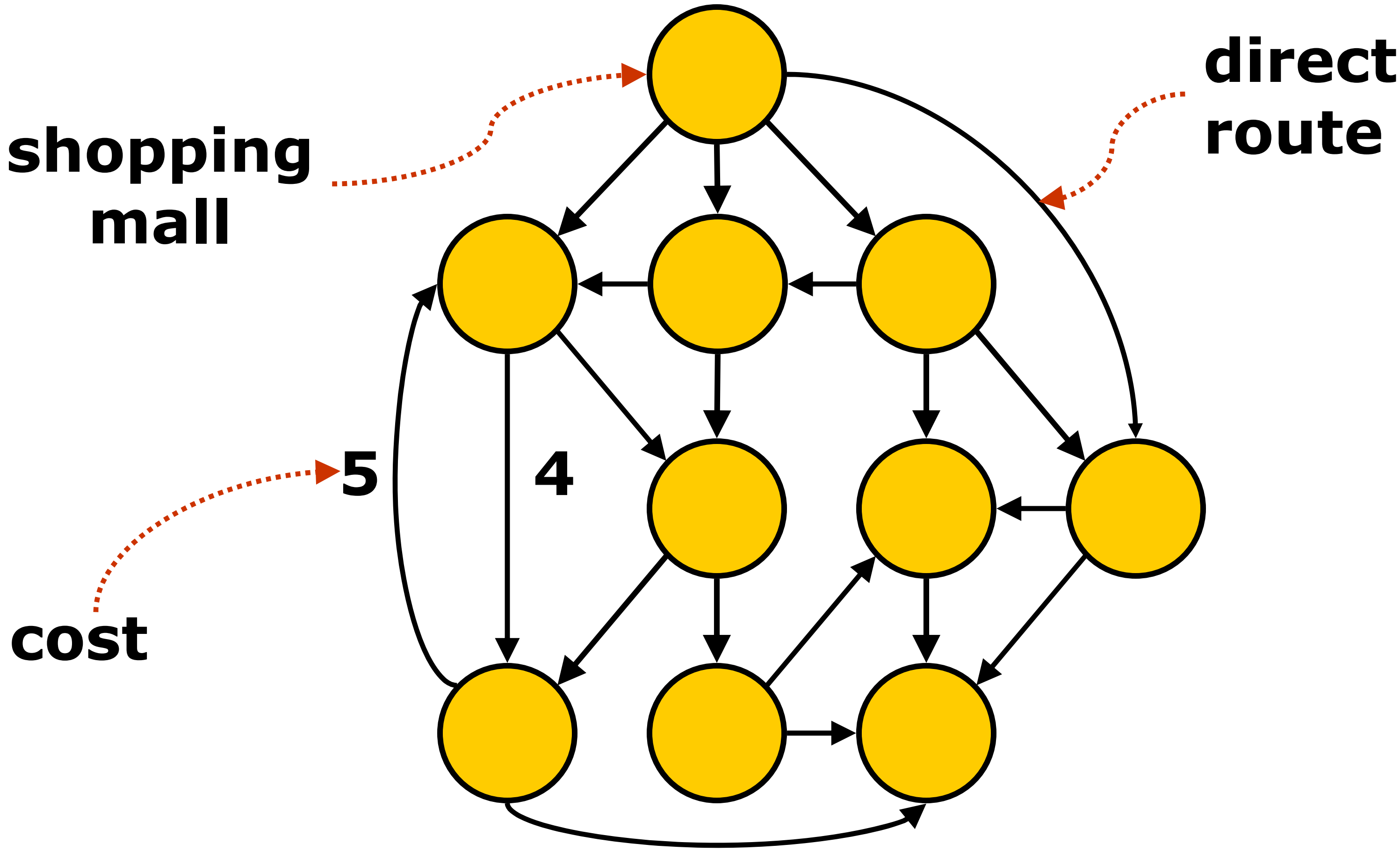
A graph does not have to be connected.

This graph is a **disconnected graph** and has two or more **connected components**.



Applications

Travel Planning



Questions

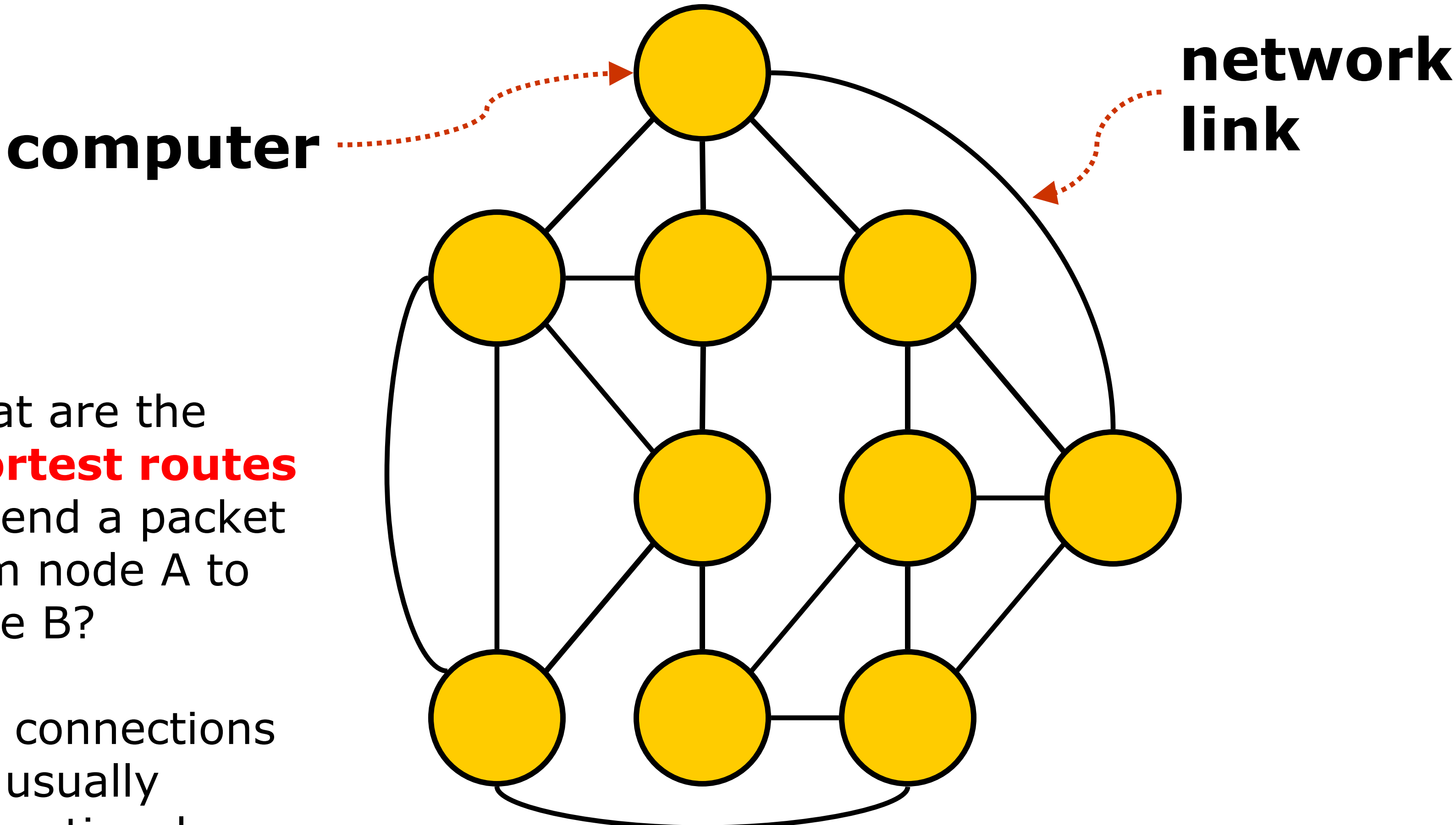
- What is the shortest way to travel between A and B?

  "**SHORTEST PATH PROBLEM**"

- How to minimize the cost of visiting n cities such that we visit each city exactly once, and finishing at the city where we start from?

  "**TRAVELING SALESMAN PROBLEM**"

Internet

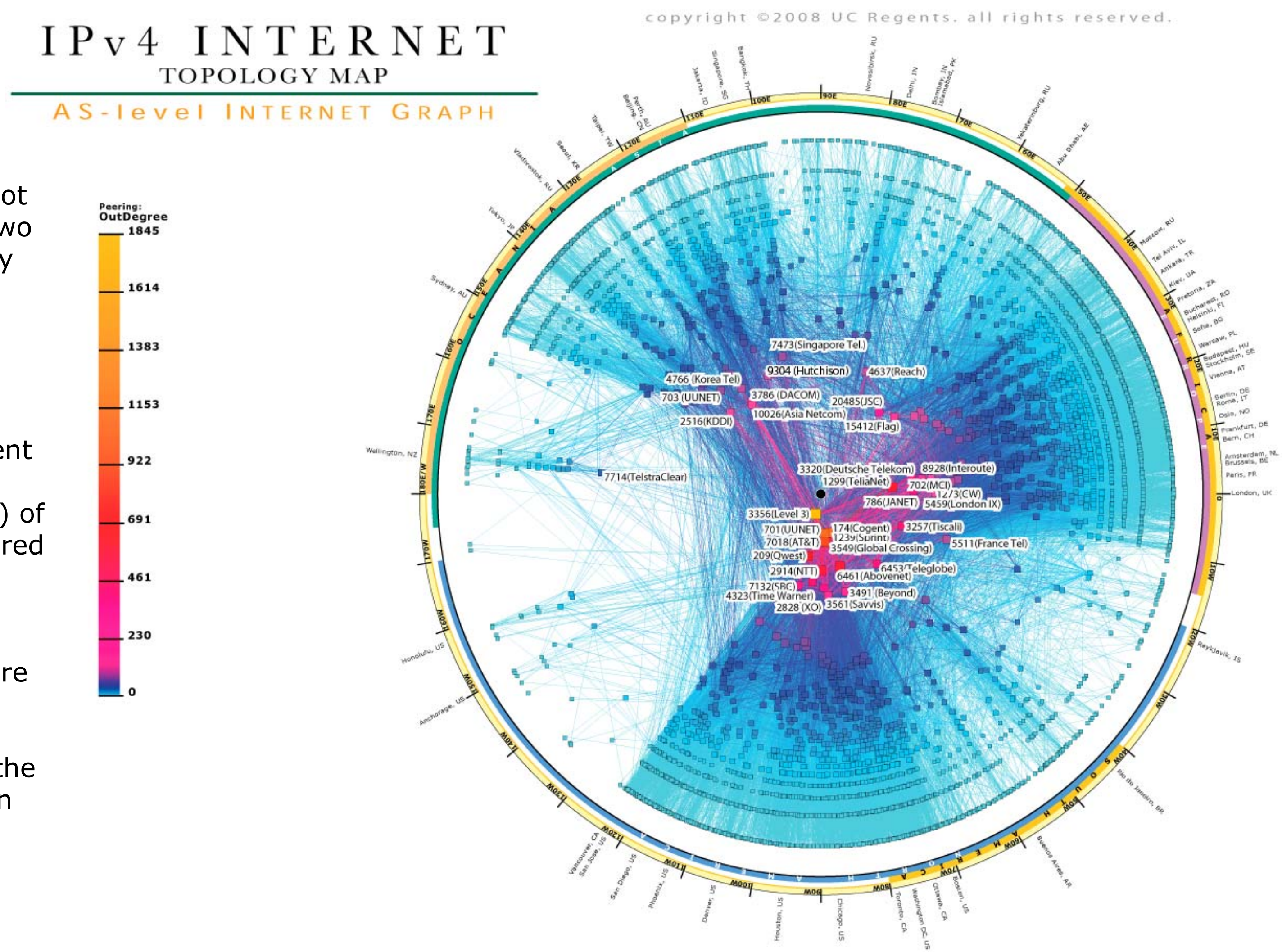


What are the **shortest routes** to send a packet from node A to node B?

The connections are usually bidirectional.

Internet Graph

This visualization represents a macroscopic snapshot of the Internet for two weeks: 1-17 January 2008. The graph reflects 4,853,991 observed IPv4 addresses and 5,682,419 IP links (immediately adjacent addresses in a traceroute-like path) of topology data gathered from 13 monitors probing 48,535,339 /24s spread across 235,286 (49.3% were reached) globally routable network prefixes (94.9% of the total prefixes seen in RouteViews on 1 January 2008).



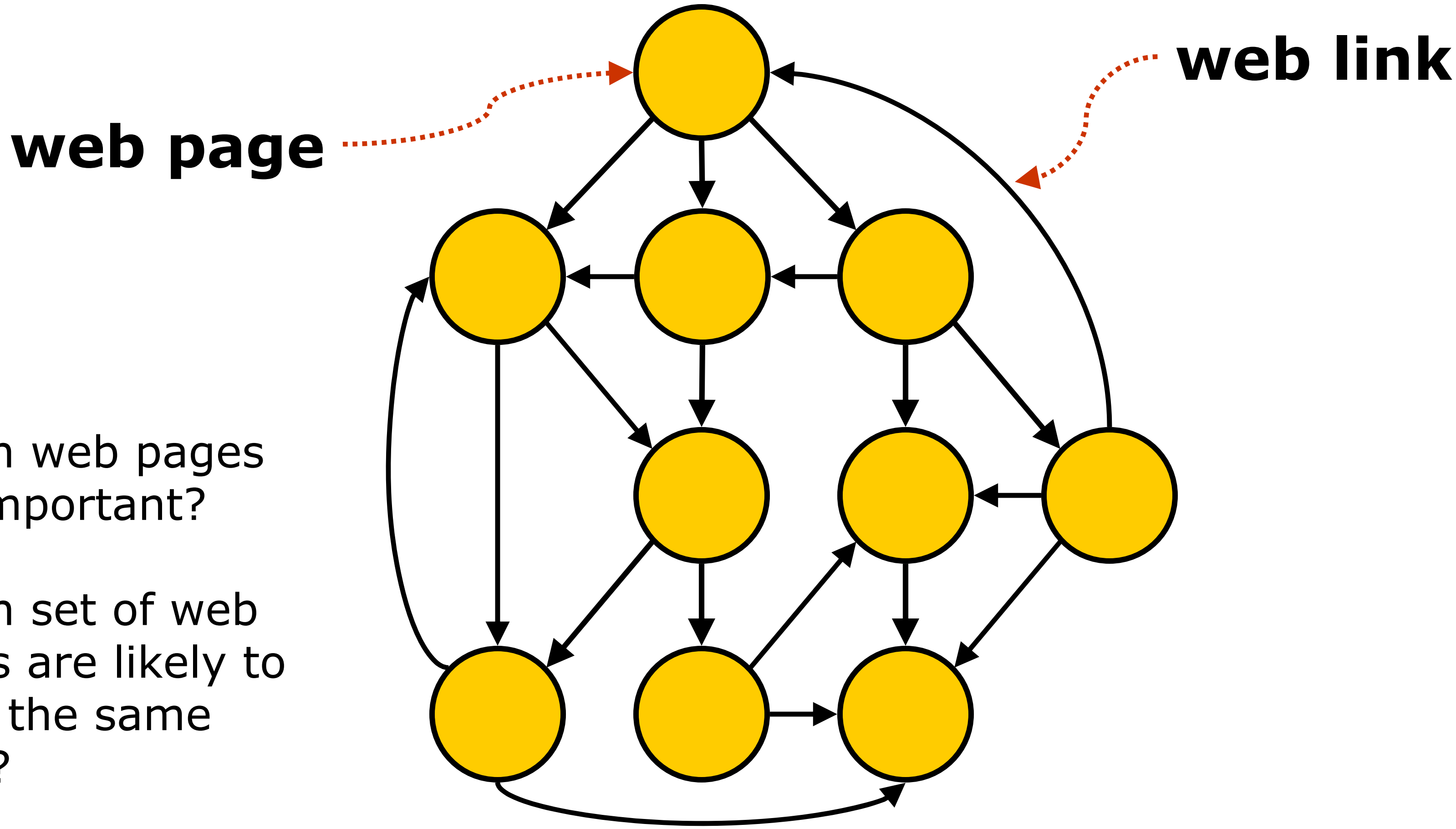
Question

- What is the shortest route to send a packet from computer A to computer B?

  This is the same as

  “**SHORTEST PATH PROBLEM**”

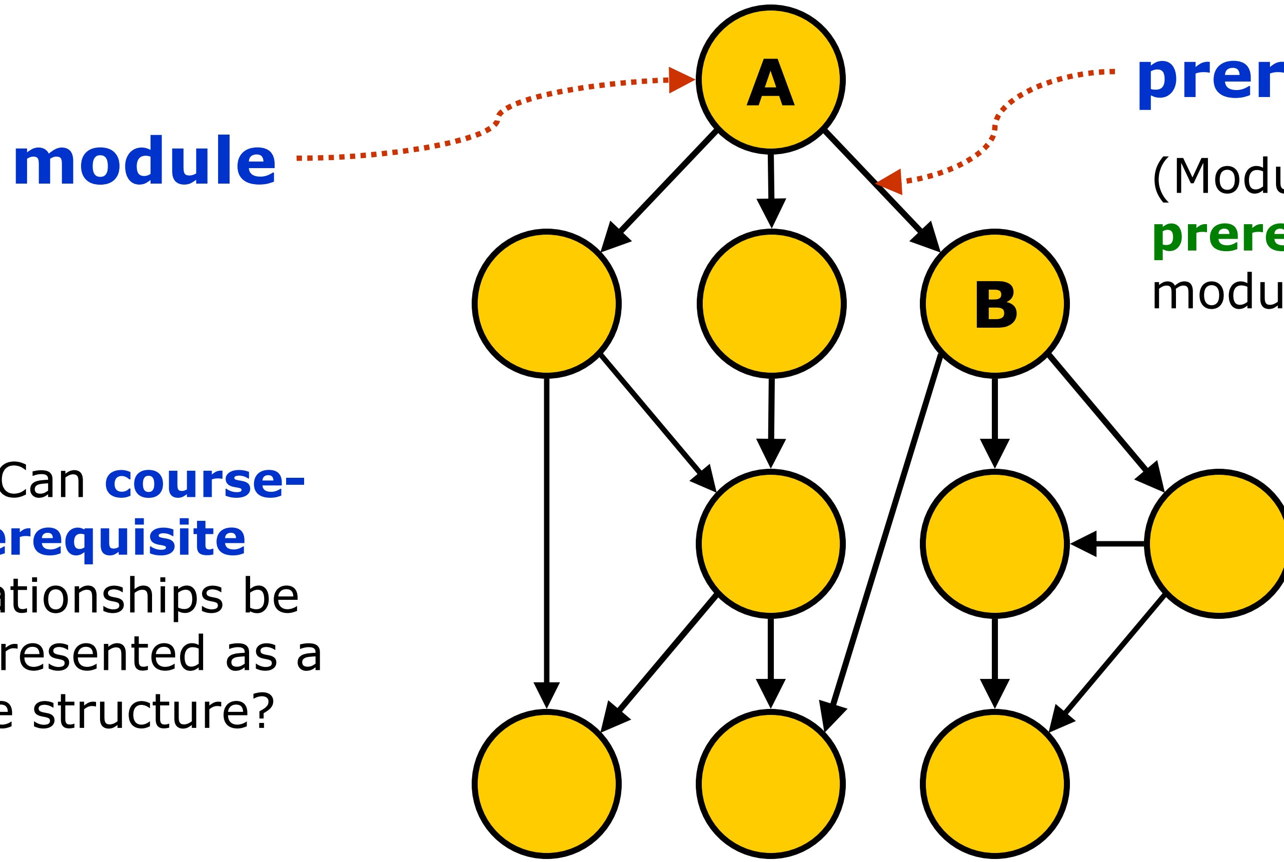
The Web



Which web pages are important?

Which set of web pages are likely to be of the same topic?

Module Selection



(Module A is a **prerequisite** of module B)

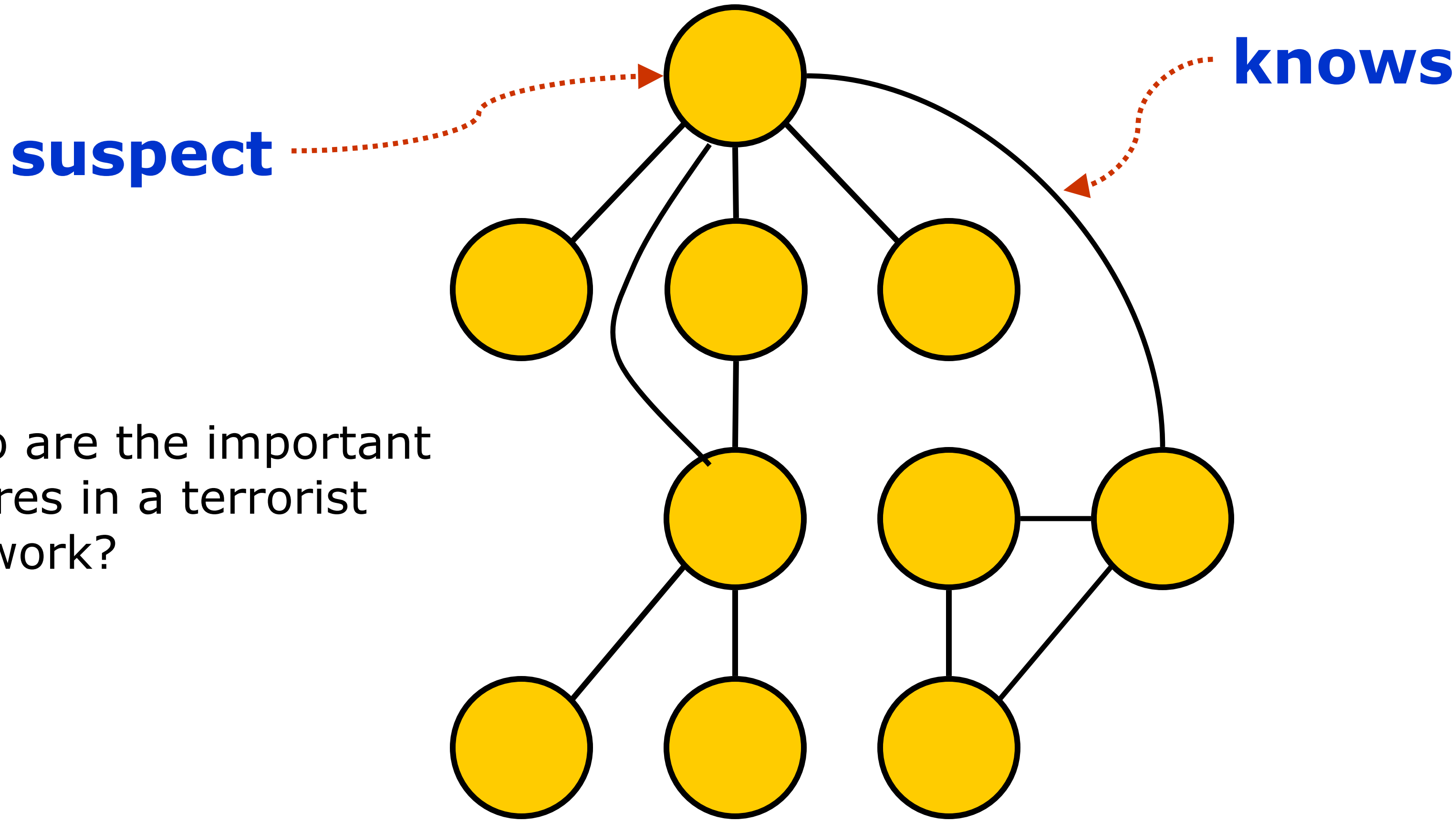
Q: Can **course-prerequisite** relationships be represented as a tree structure?

Question

- Find a sequence of modules to take that satisfy the prerequisite requirements of a given module.

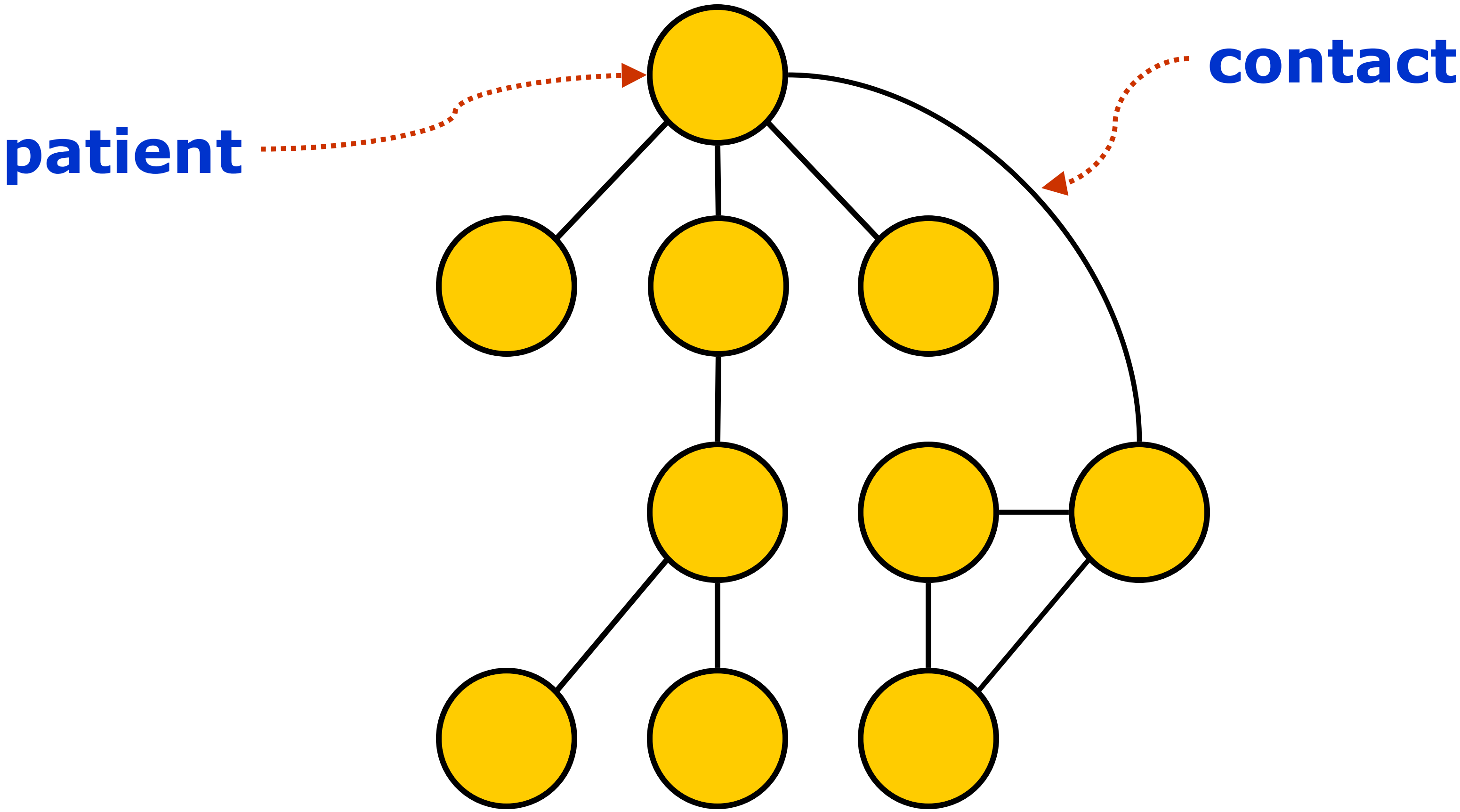
"TOPOLOGICAL SORT"

Terrorist



Who are the important figures in a terrorist network?

Epidemic Studies



Other applications

- Biology
- VLSI layout
- Vehicle routing
- Job scheduling
- Facility location

:

:

Implementation

Formally

A **graph** $G = (V, E, w)$, where

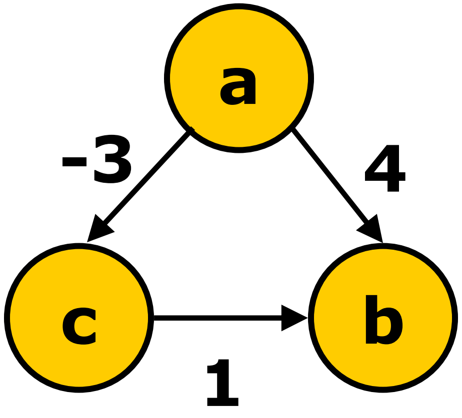
- V is the set of vertices
- E is the set of edges
- w is the weight function

Example

V = { a, b, c }

E = { (a,b), (c,b), (a,c) }

w = { ((a,b), 4), ((c, b), 1), ((a,c),-3) }



Adjacent vertices

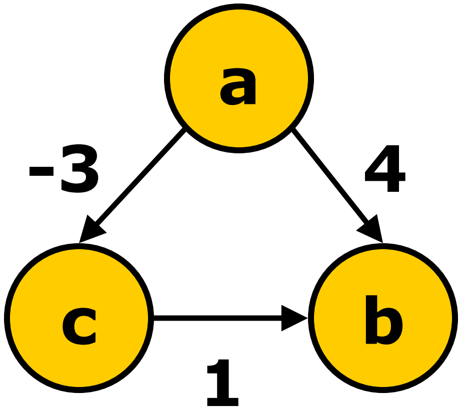
- **adj(v)** = set of vertices **adjacent** to v

  adj(a) = {b, c}

  adj(b) = {}

  adj(c) = {b}

- $\sum_v |adj(v)| = |E|$
- adj(v): **Neighbours** of v



Note: Vertices adjacent to v are called neighbours or successors of v

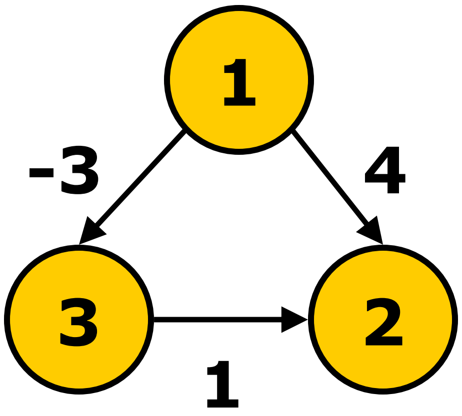
Adjacency matrix

double vertex[][];

To

from

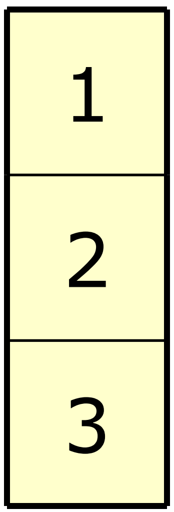
	1	2	3
1	∞	4	-3
2	∞	∞	∞
3	∞	1	∞

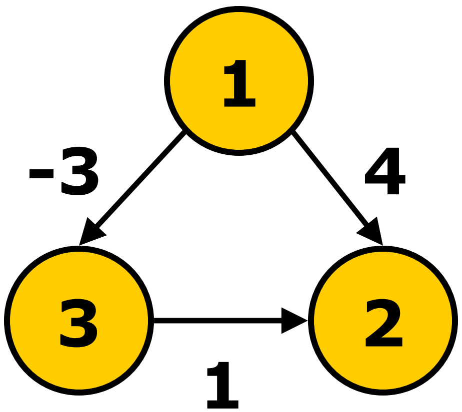


- This requires **$O(N^2)$ memory**, and is not suitable for **sparse graph**.
- However, an edge can be accessed in O(1) time.
- How about undirected graph? How would you represent it?

Adjacency list

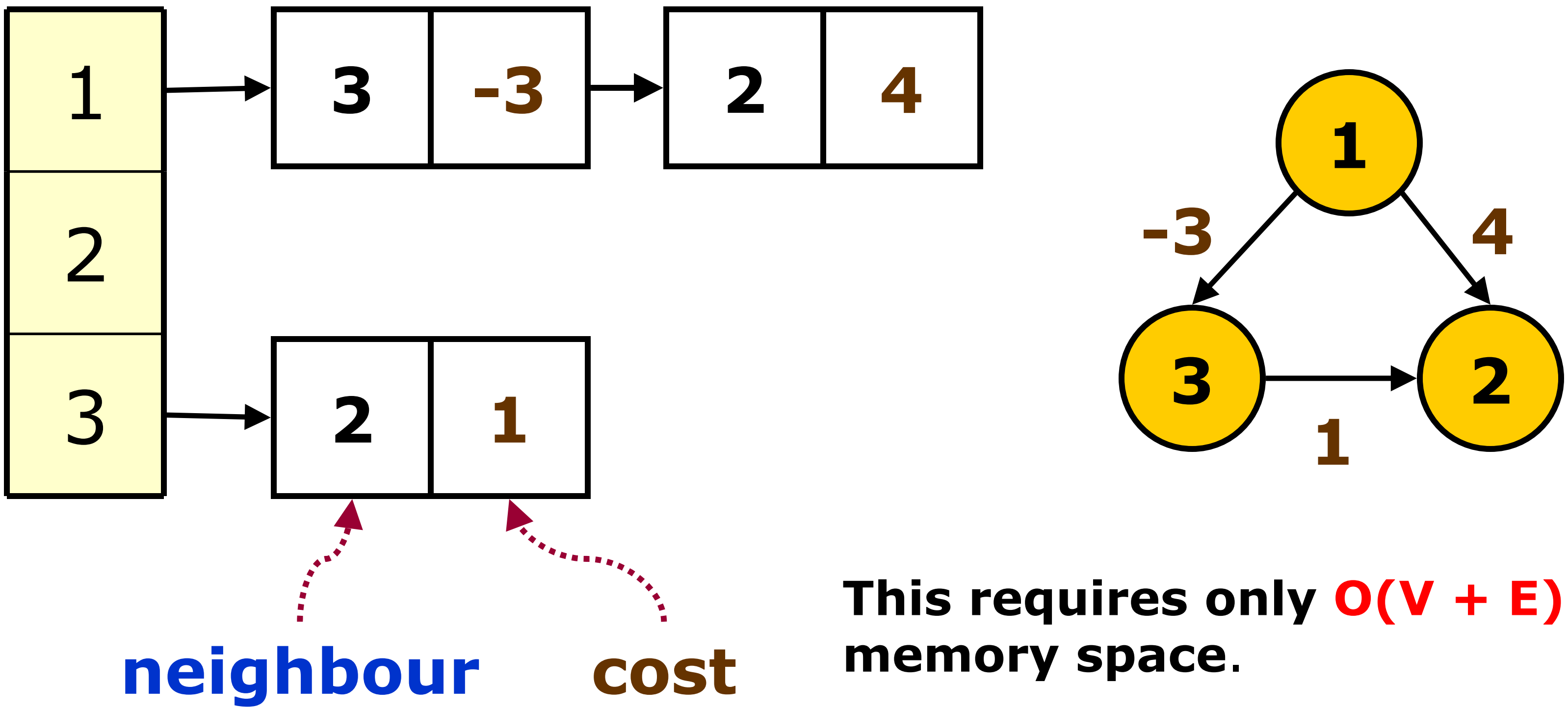
EdgeList vertex[];





Adjacency list

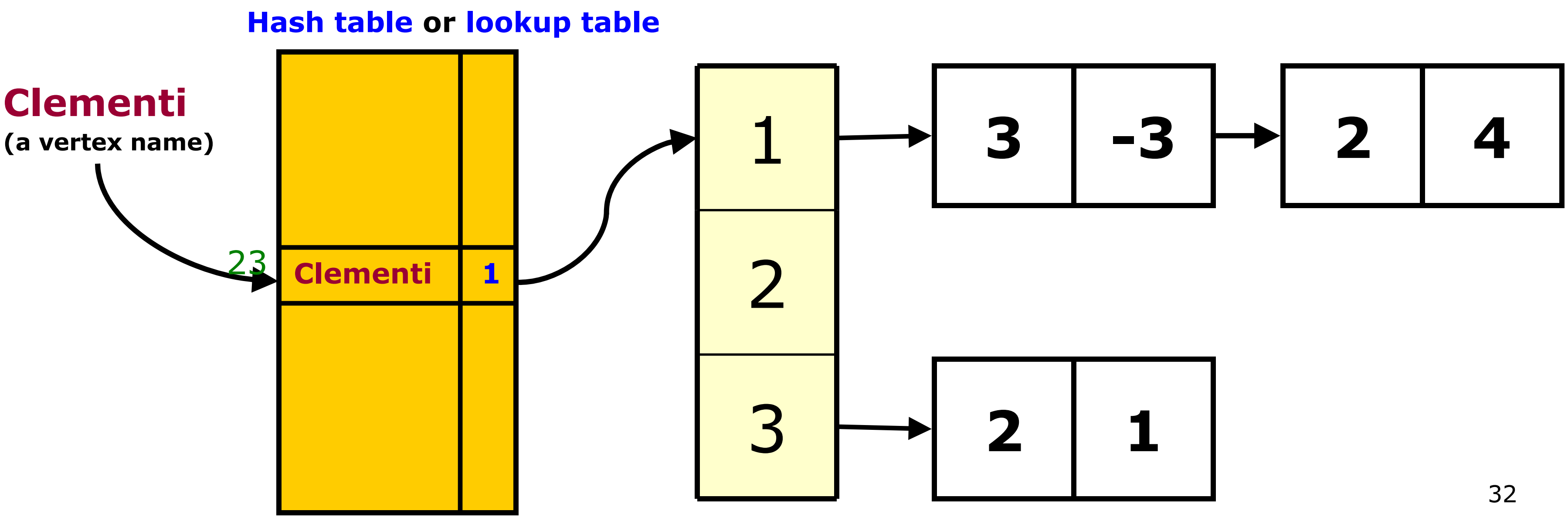
EdgeList vertex[];



This requires only O(V + E) memory space.

Vertex map

- Since vertices are usually identified by names (e.g. person, city), not integers, we can use a hash table (or lookup table) to map names to indices in our adjacency list/matrix.
- The indices in this case is NOT the bucket/slot number you get after hashing.
- We actually store the indices as data in the hash table.



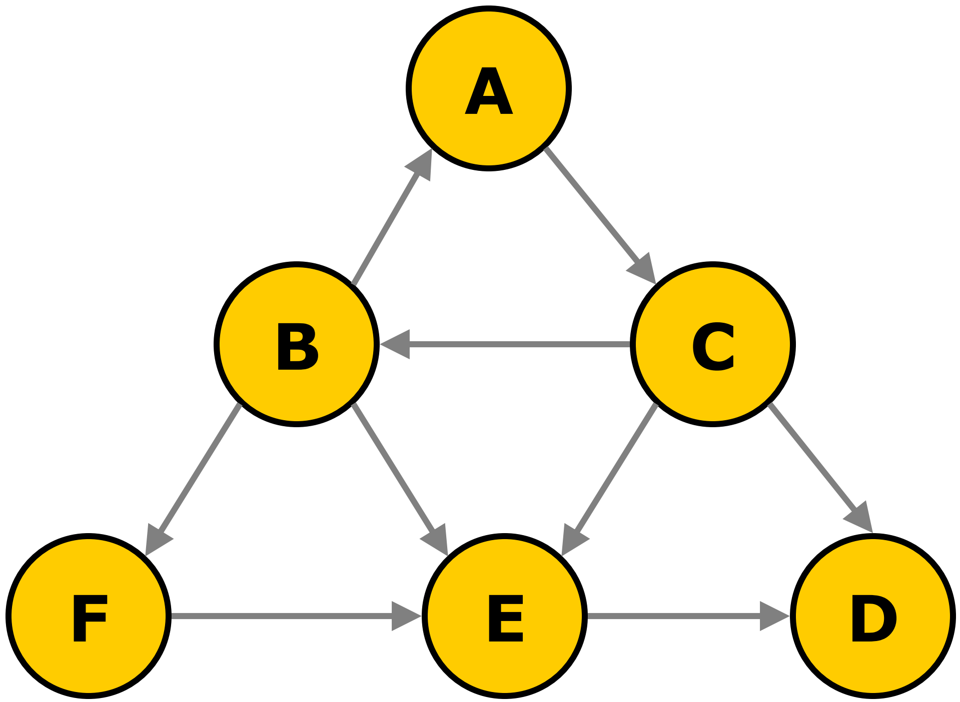
Breadth-First Search (BFS)

Traversing a Graph

Breadth-first search (BFS)

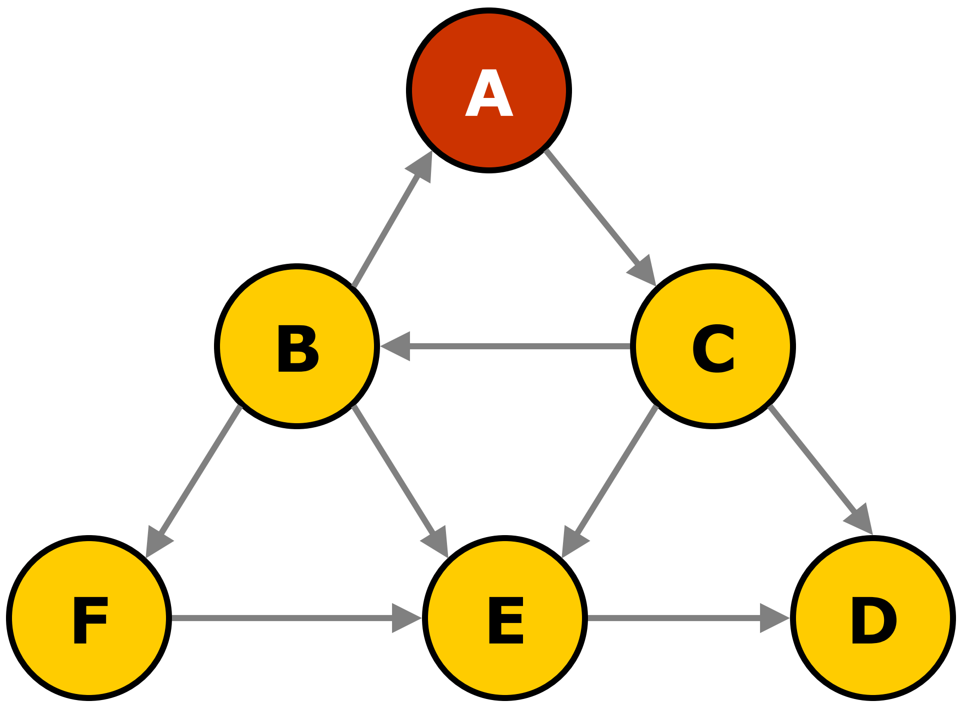
- Given a **source vertex**, we like to start searching from that source.
- The idea of **BFS** is that we visit all vertices that are of **distance i** away from the source **before** we visit vertices that are of **distance i+1** away from the source.
- The order of search is **not unique** and depends on the order of neighbours visited.

Breadth-first search (BFS)

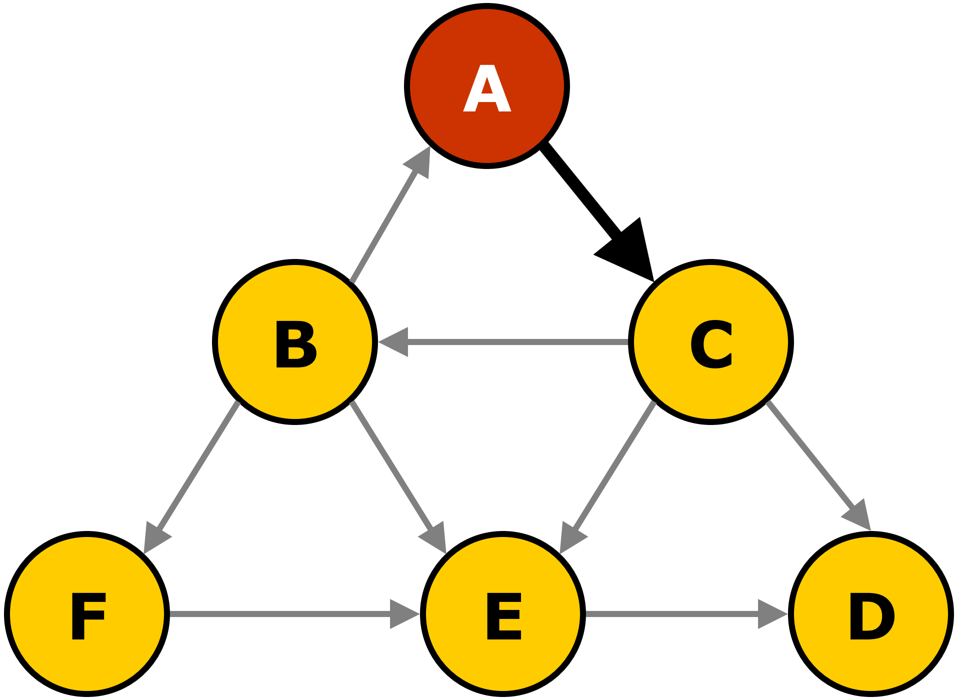


Breadth-first search

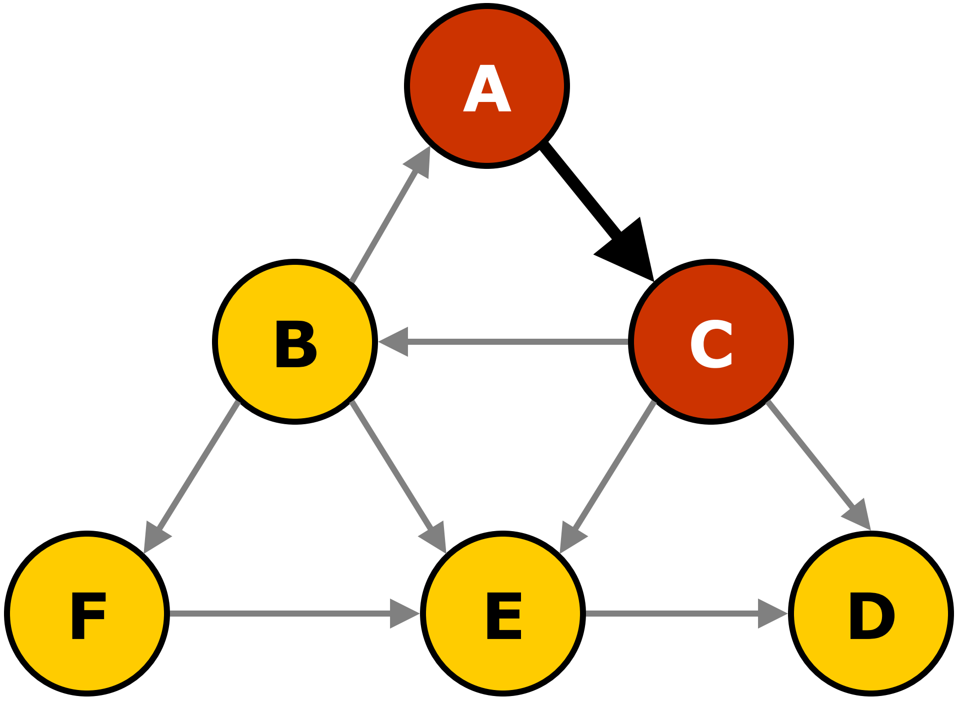
Start searching from a given **source**, say A



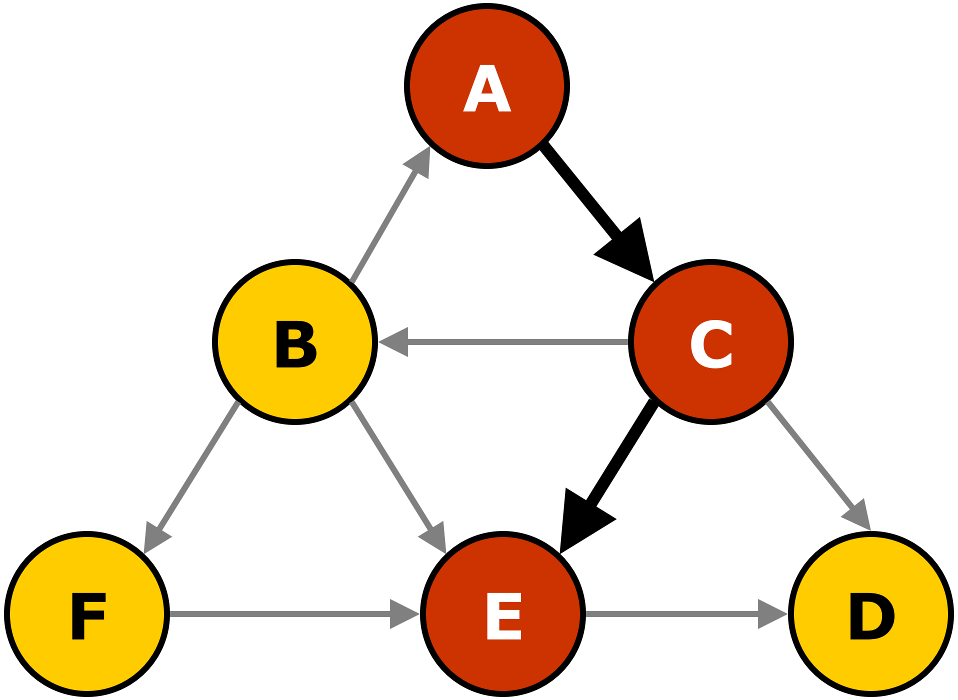
Breadth-first search



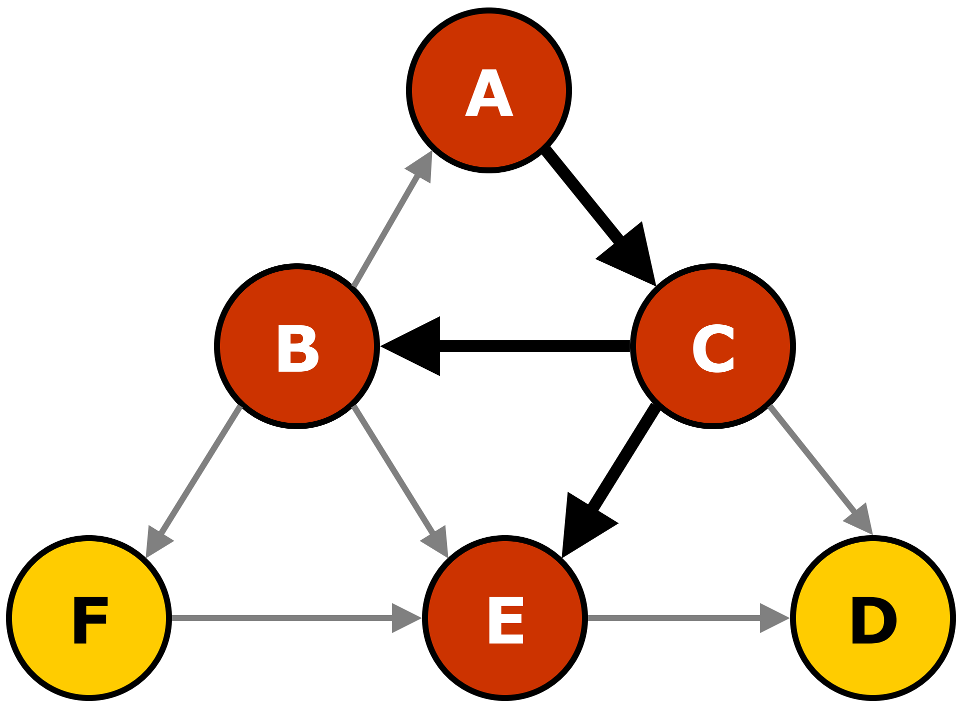
Breadth-first search



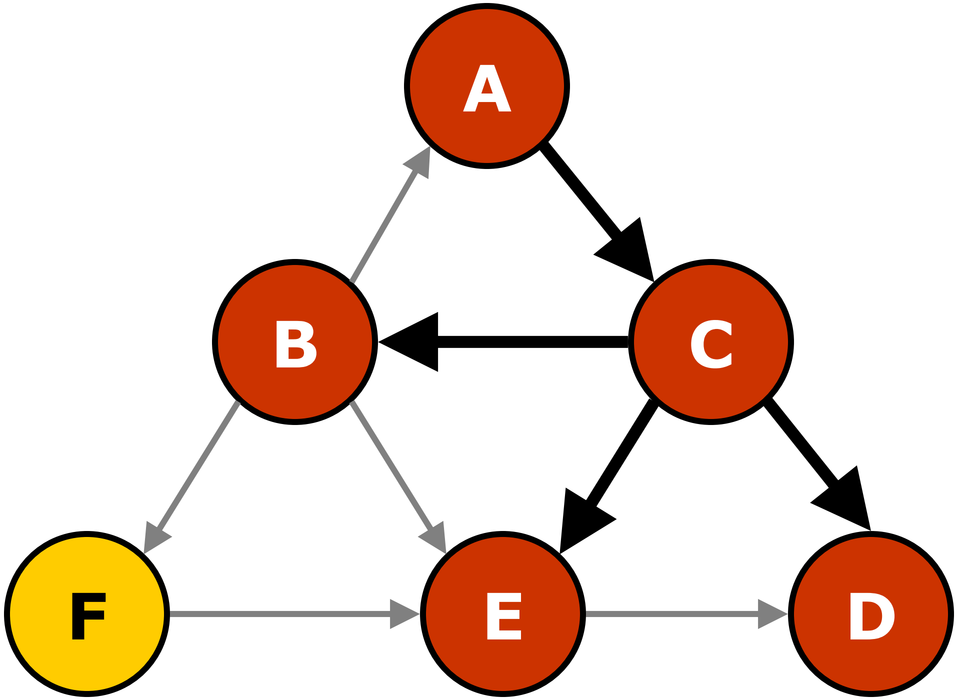
Breadth-first search



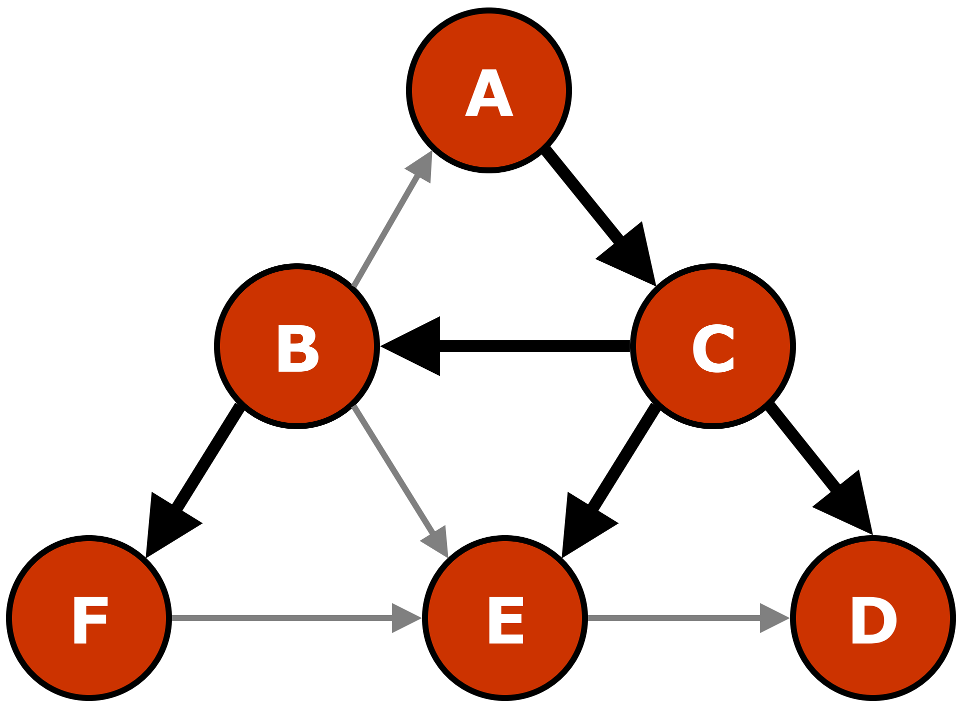
Breadth-first search



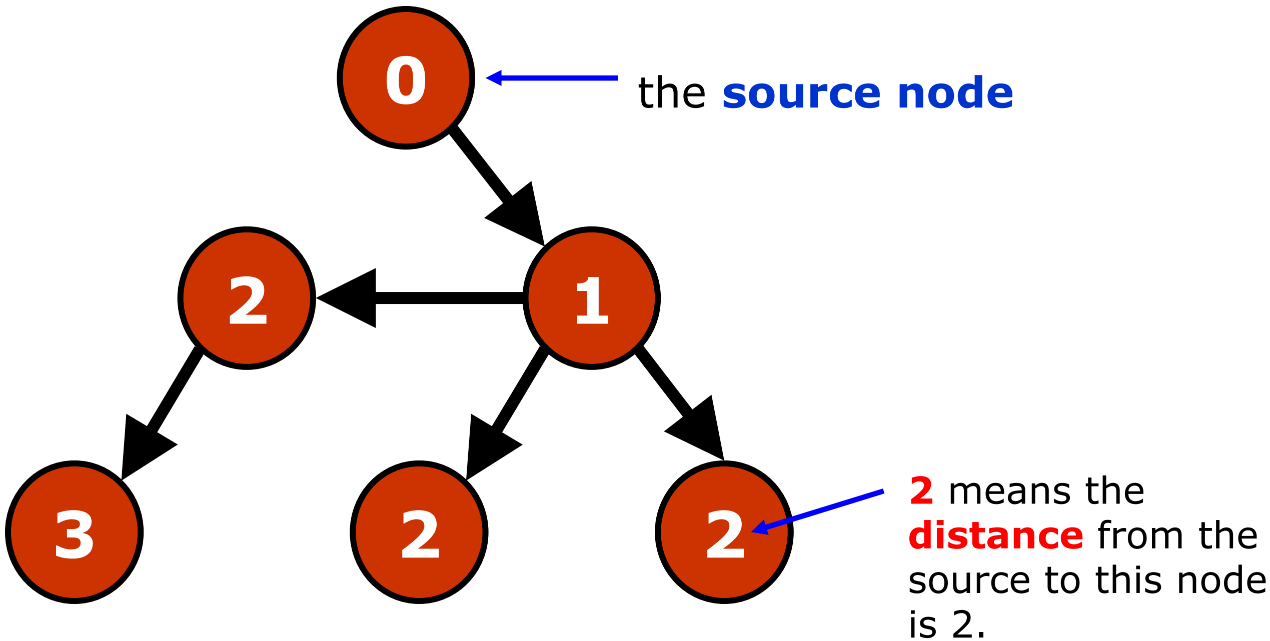
Breadth-first search



Breadth-first search



Breadth-first search

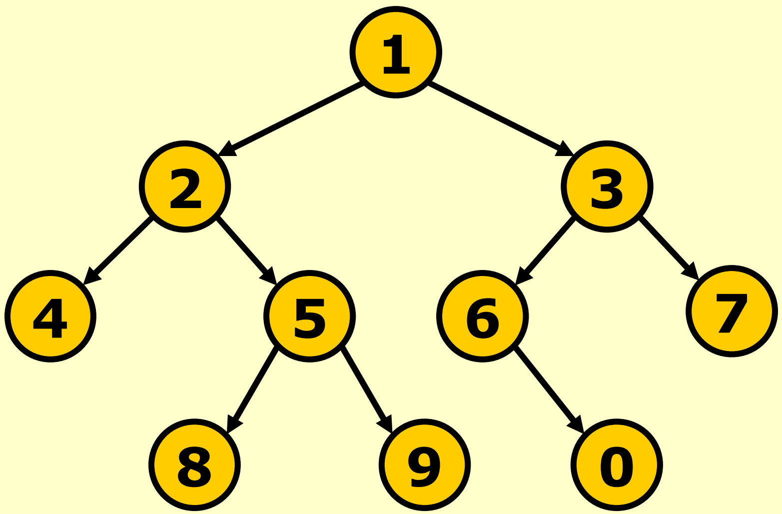


- After BFS, we get a **tree** rooted at the source node. Edges in the tree are edges that we followed during searching. We call this a **BFS tree**.
- Vertices in the figure are labeled with their **distance** (or level) from the source.

Recall

Level-Order on Binary Tree

```
if T is empty return
Q = new Queue
Q.enq(T)
while Q is not empty
   curr = Q.deq()
   print curr.element
   if T.left is not empty
        Q.enq(curr.left)
   if curr.right is not empty
        Q.enq(curr.right)
```



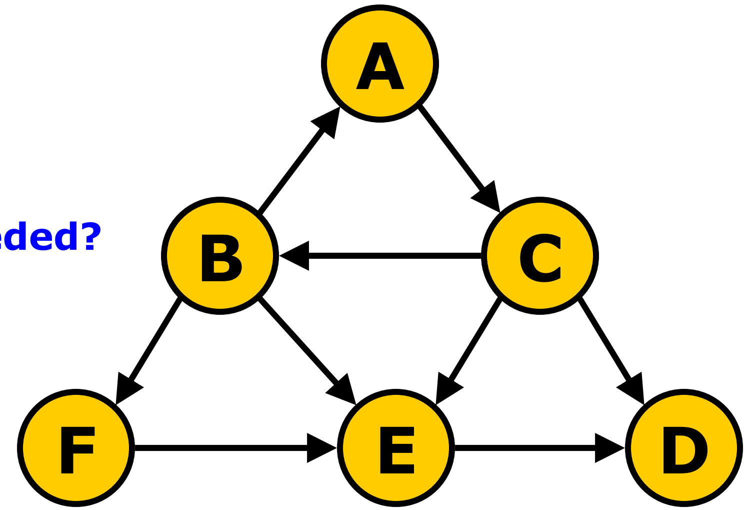
Note: Need to use a **queue**.

Breath First Search

BFS(v)

where v is the given source

```
Q = new Queue
Q.enq (v)
mark v as visited // Why needed?
while Q is not empty
   curr = Q.deq()
   print curr
   foreach w in adj(curr)   // w is a neighbour of curr
      if w is not visited   //Why needed?
         Q.enq(w)
         mark w as visited  //Why needed?
```

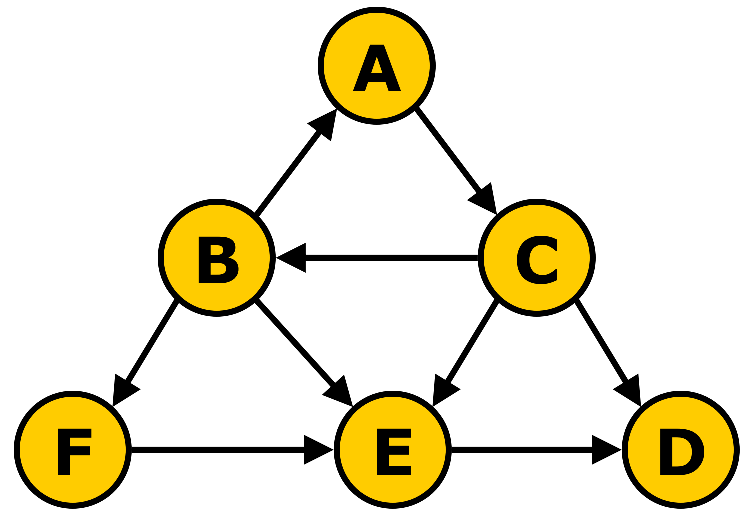


BFS(v) **(cont.)**

- If the graph is a **connected** graph, then BFS(v) will visit **all** the vertices. Why?
- If the graph is a **disconnected** graph with **k** connected components, then we need to call BFS **k** times for one node in each of the connected components. Why?

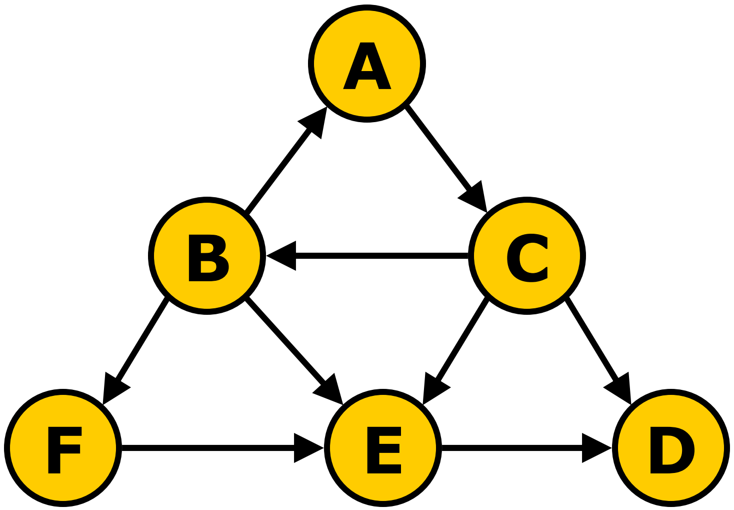
Building the BFS Tree

```
Q = new Queue
Q.enq (v)
mark v as visited
while Q is not empty
   curr = Q.deq()
   print curr
   foreach w in adj(curr)
      if w is not visited
            Q.enq(w)
            w.parent = curr    // need to remember the parent
            mark w as visited
```



Calculating Level

```
Q = new Queue
Q.enq (v)
mark v as visited
v.level = 0
while Q is not empty
   curr = Q.deq()
   print curr
   foreach w in adj(curr)
      if w is not visited
         Q.enq(w)
         w.level = curr.level + 1
         mark w as visited
```



Search all vertices of a graph

```
Search(G)
   foreach vertex v
      mark v as not visited      // initialization
   foreach vertex v
      if v is not visited
         BFS(v)
```

Note: we need to call BFS more than one time if the graph is a **disconnected graph**.

Running time

```
Q = new Queue
Q.enq (v)
mark v as visited
while Q is not empty
   curr = Q.deq()
   print curr
   foreach w in adj(curr)
      if w is not visited
         Q.enq(w)
         mark w as visited
```

Initialization

$$O(V)$$

Where V means the no. of vertices.

Main Loop

$$O(\sum_{curr \in V} adj(curr)) = O(E)$$

where E means the no. of edges.

Each vertex is enqueued exactly **once**. The for loop runs through all vertices in the adjacency list. Therefore the running time is $O(\Sigma_v \text{ adj}(v)) = O(E)$.

Total Running Time

$$O(V + E)$$

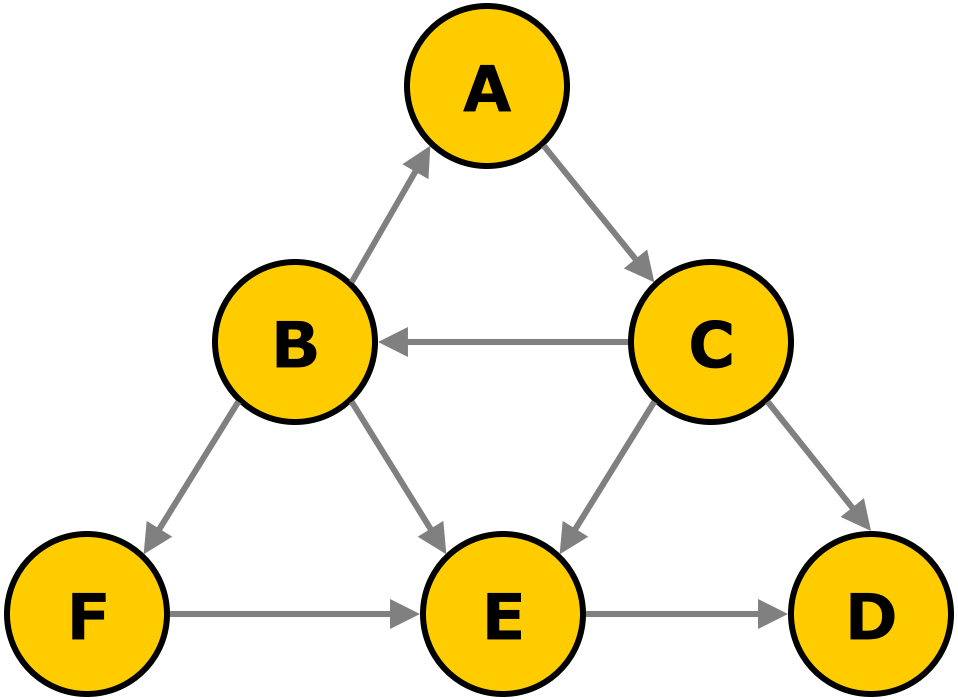
Depth-First Search (DFS)

Traversing a Graph

Idea for **DFS** is to go **as deep as possible**.
Whenever there is an outgoing edge, we follow it.

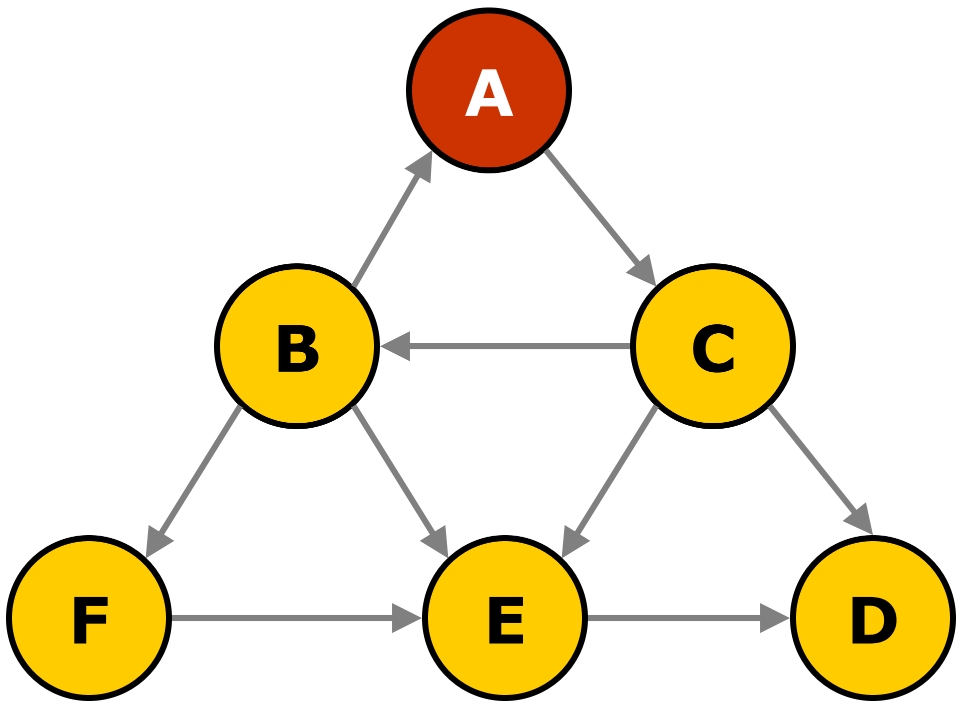
Q: Is it similar to pre-order, in-order, or post-order of binary tree traversal?

Depth-first search (DFS)

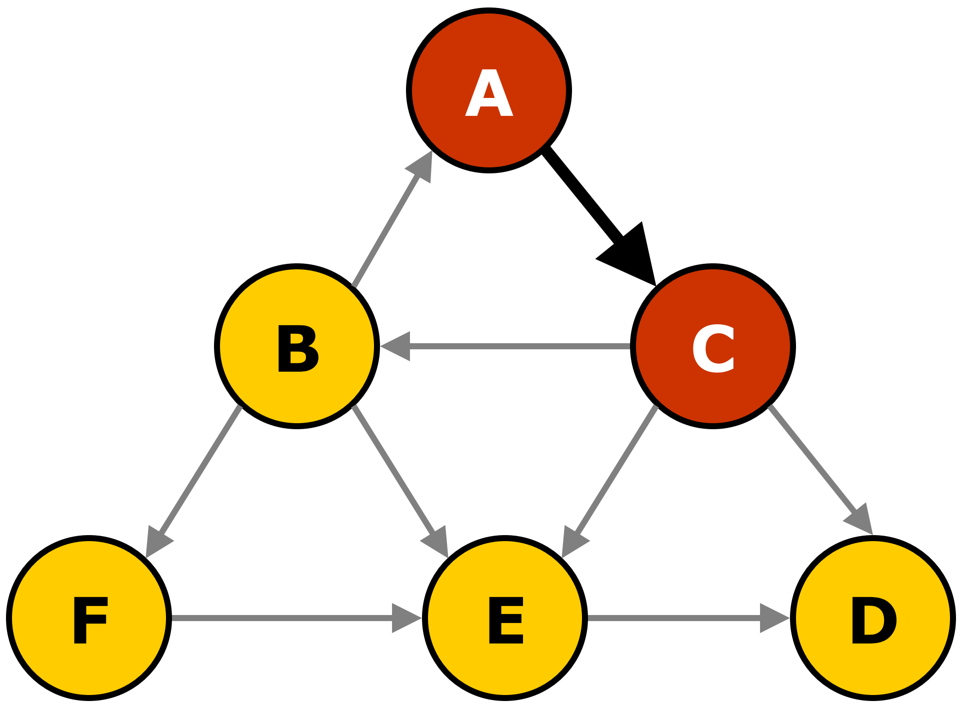


Depth-first search

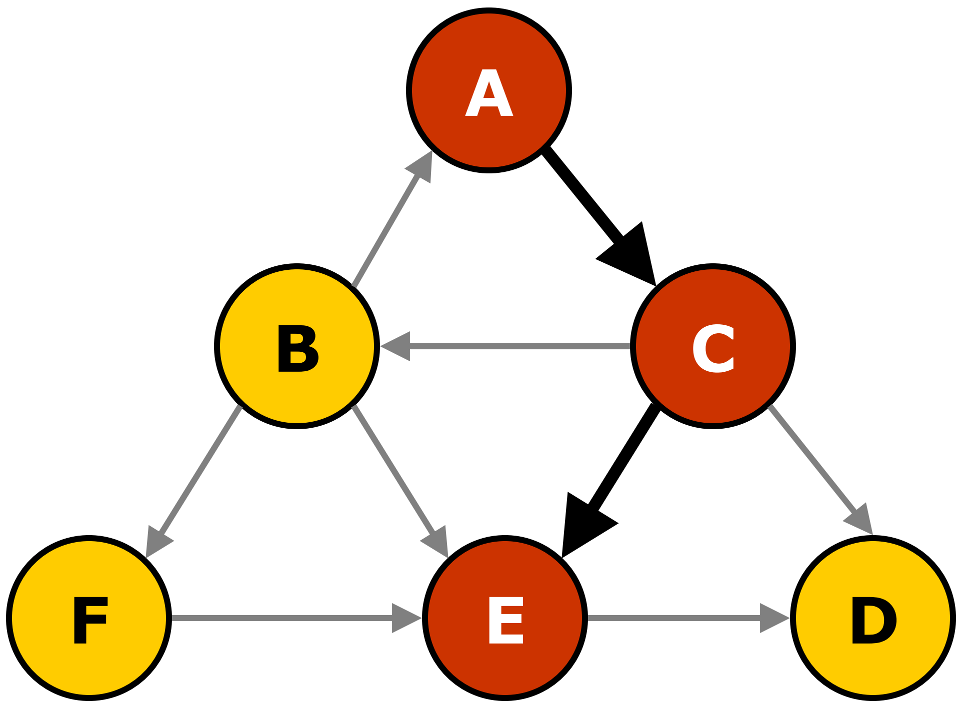
Start searching from a given **source**, say A



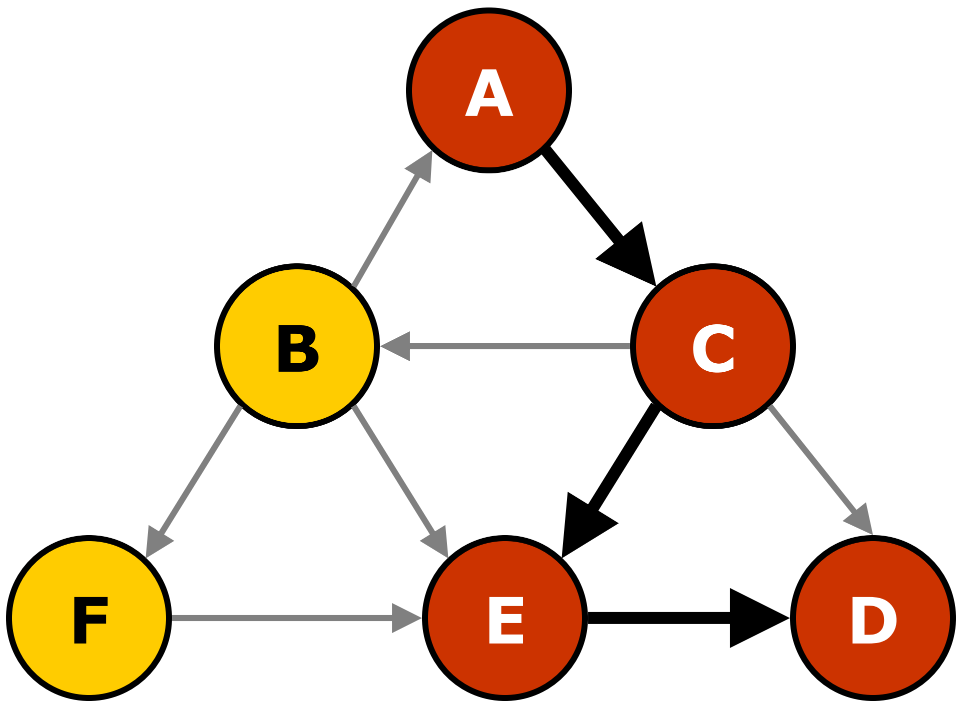
Depth-first search



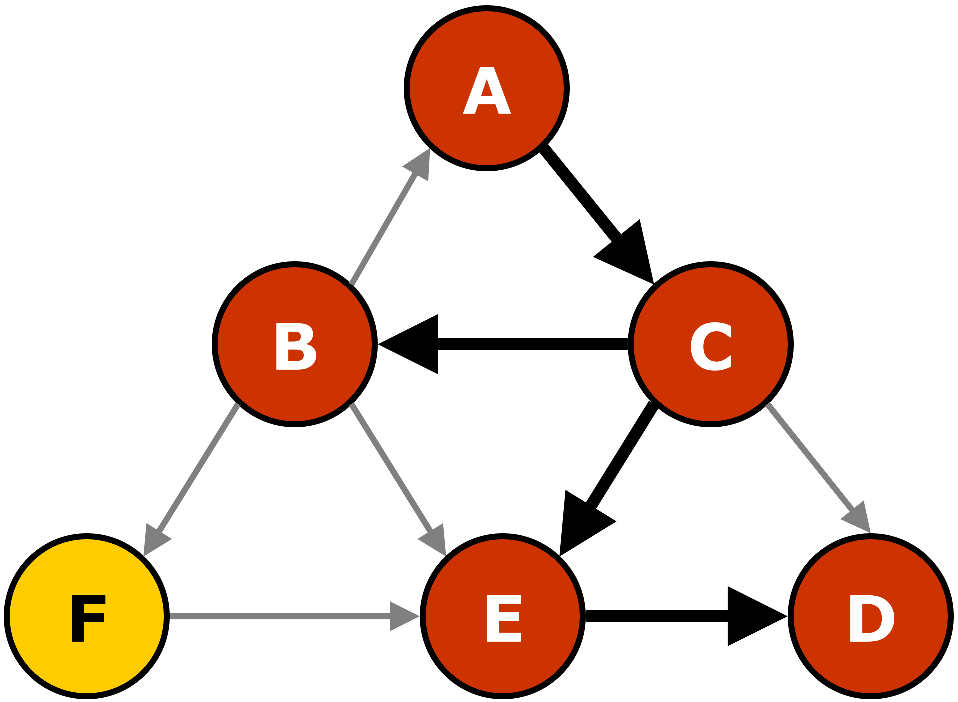
Depth-first search



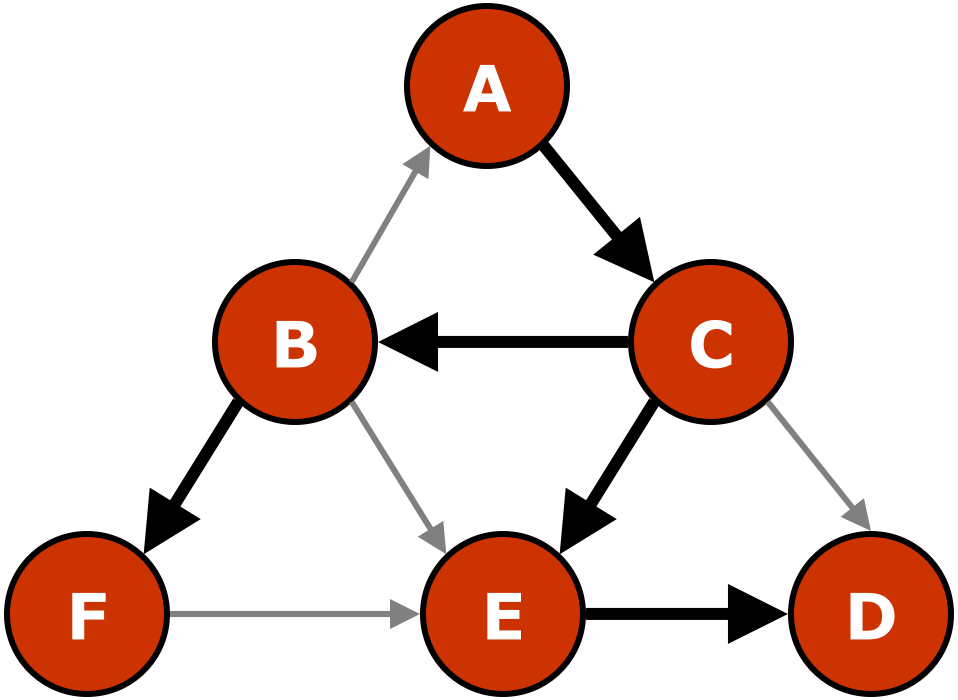
Depth-first search



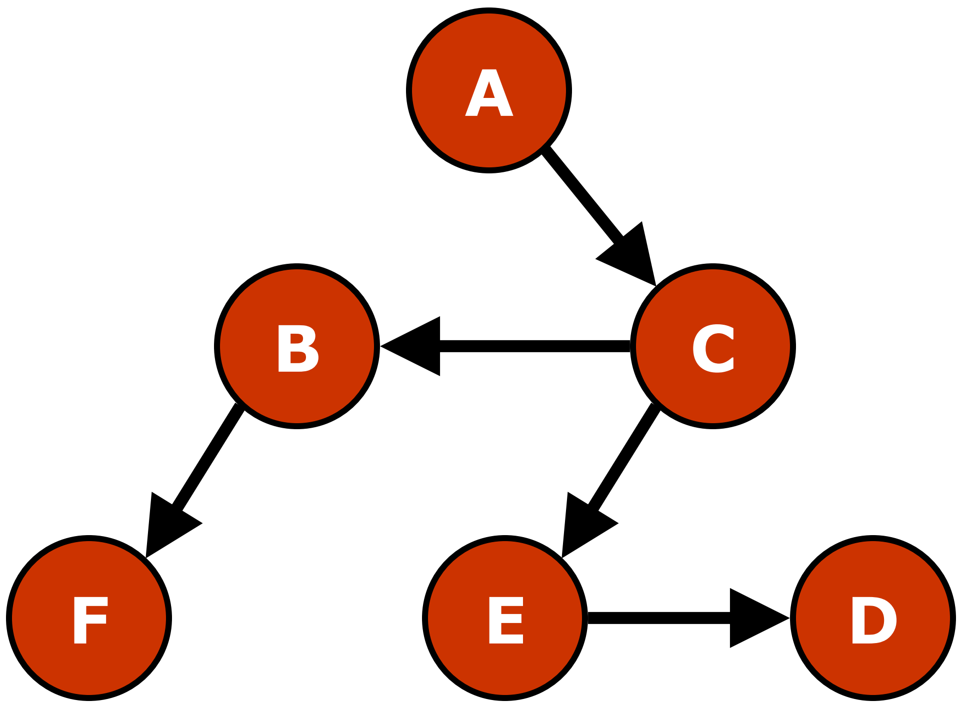
Depth-first search



Depth-first search

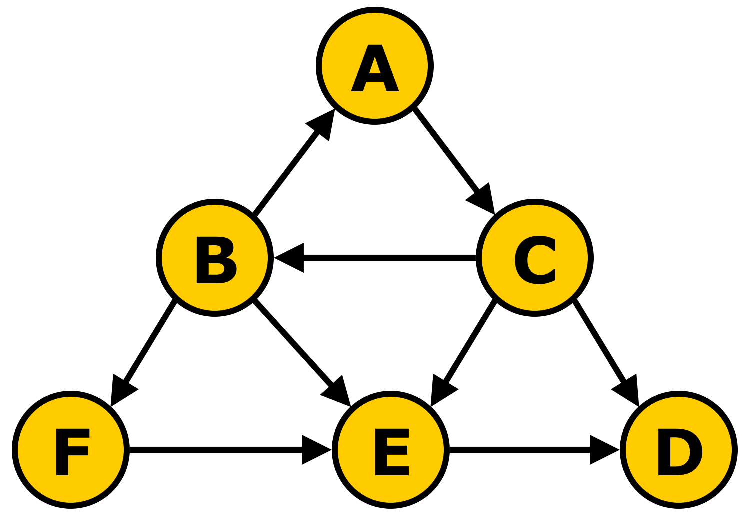


Depth-first search



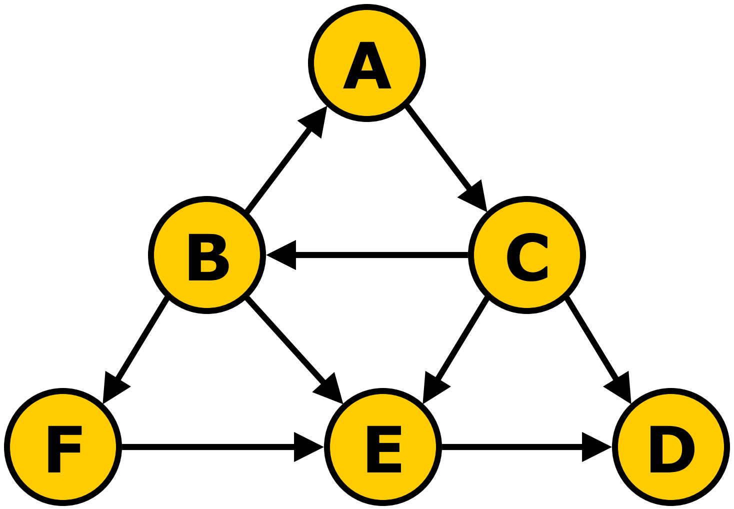
DFS(v) - use a stack to remember where to backtrack to

```
S = new Stack
S.push (v)
print and mark v as visited
while S is not empty
   curr = S.top()
   if every vertex in adj(curr)
      is visited
         S.pop()
   else
         let w be an unvisited vertex in adj(curr)
         S.push(w)
         print and mark w as visited
```



Recursive version: DFS(v)

```
print v
marked v as visited
foreach w in adj(v)
   if w is not visited
      DFS(w)
```



Search all vertices of a graph

```
Search(G)
   foreach vertex v
      mark v as not visited
   foreach vertex v
      if v is not visited
         DFS(v)
```

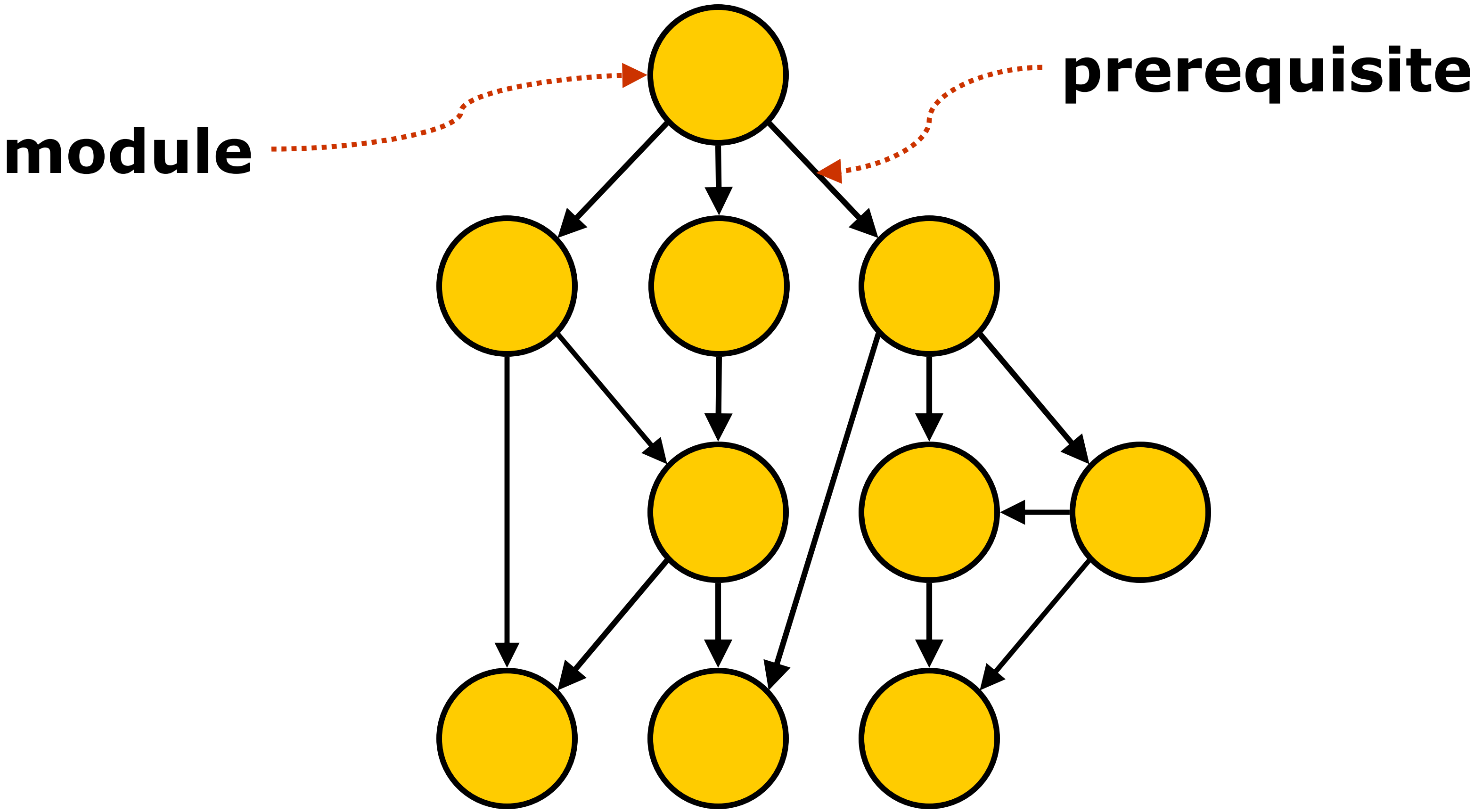
Note: Just like BFS, we need to call DFS() from multiple vertices in order to search all the vertices if the graph is a **disconnected graph**.

Running time of DFS

- DFS: $O(V + E)$

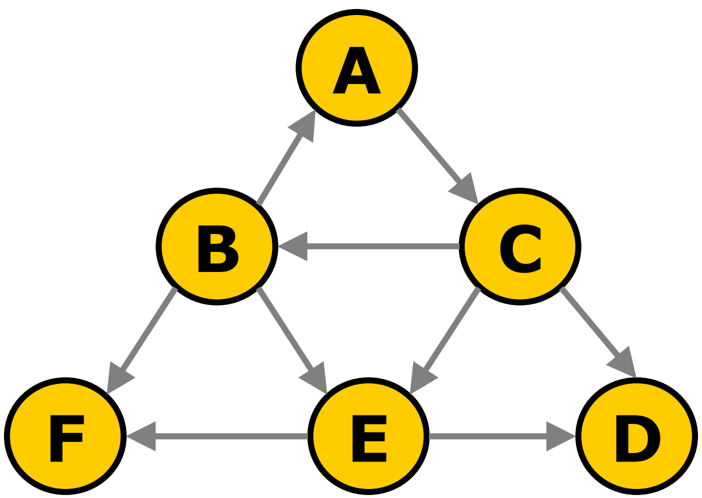
Topological Sort

Module selection

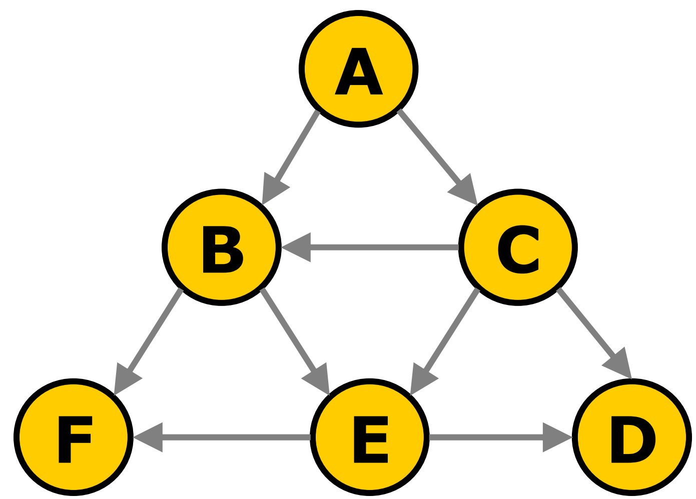


Definition

- **Directed Acyclic Graph** (**DAG**): A directed graph with no cycle.
- **Question:** Are the below 2 directed graphs directed acyclic graphs?



No! **(ACBA is a cycle)**



Yes!

Definitions

- **in-degree** of a vertex
  - number of **incoming edges** to the vertex

- **out-degree** of a vertex
  - number of **outgoing edges** from the vertex

Topological sort

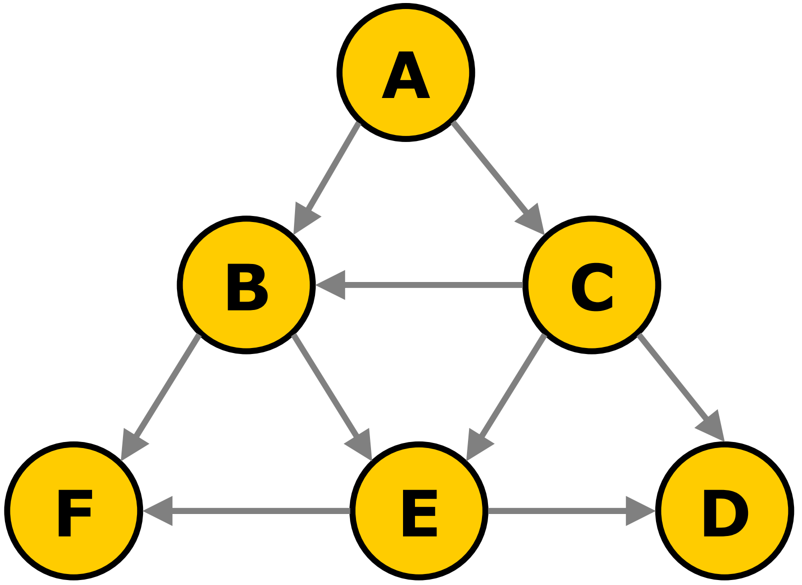
- Goal: Order the vertices, such that if there is a path from u to v, then u appears before v in the output.
- Topological sort output is not unique. Why?

- We perform topological sort by repeatedly enqueueing vertices with in-degree 0 into a queue, output the vertex de-queued from the queue and remove the edges from that vertex.
- Since the order where we enqueued vertices with 0 in-degree into the queue is not unique, the output is not unique.

Topological sort

There are more than one solution:

- ACBEFD
- ACBEDF



Note: ACDBEF is not topologically sorted. Why?

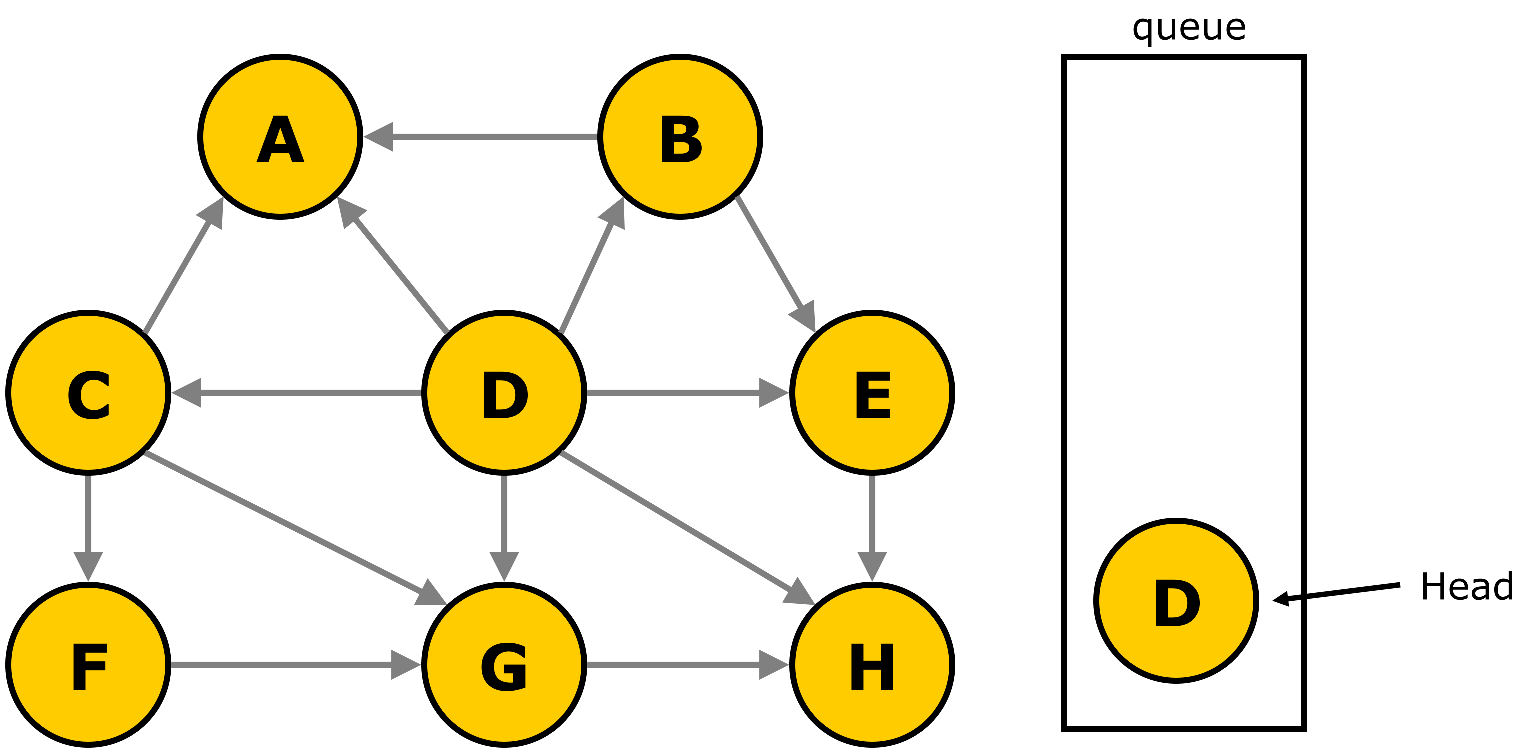
Pseudocode for Toposort

```
q = new Queue()
put all vertices with in-degree 0 into q
while q is not empty
   v = q.dequeue()
   print v
   remove v together with its edges from G
         (i.e. need to re-compute the in-degrees of all the neighbours of v)
   enqueue neighbours of v with in-degree 0
```

Q: What is the running time for Toposort?

Topological sort

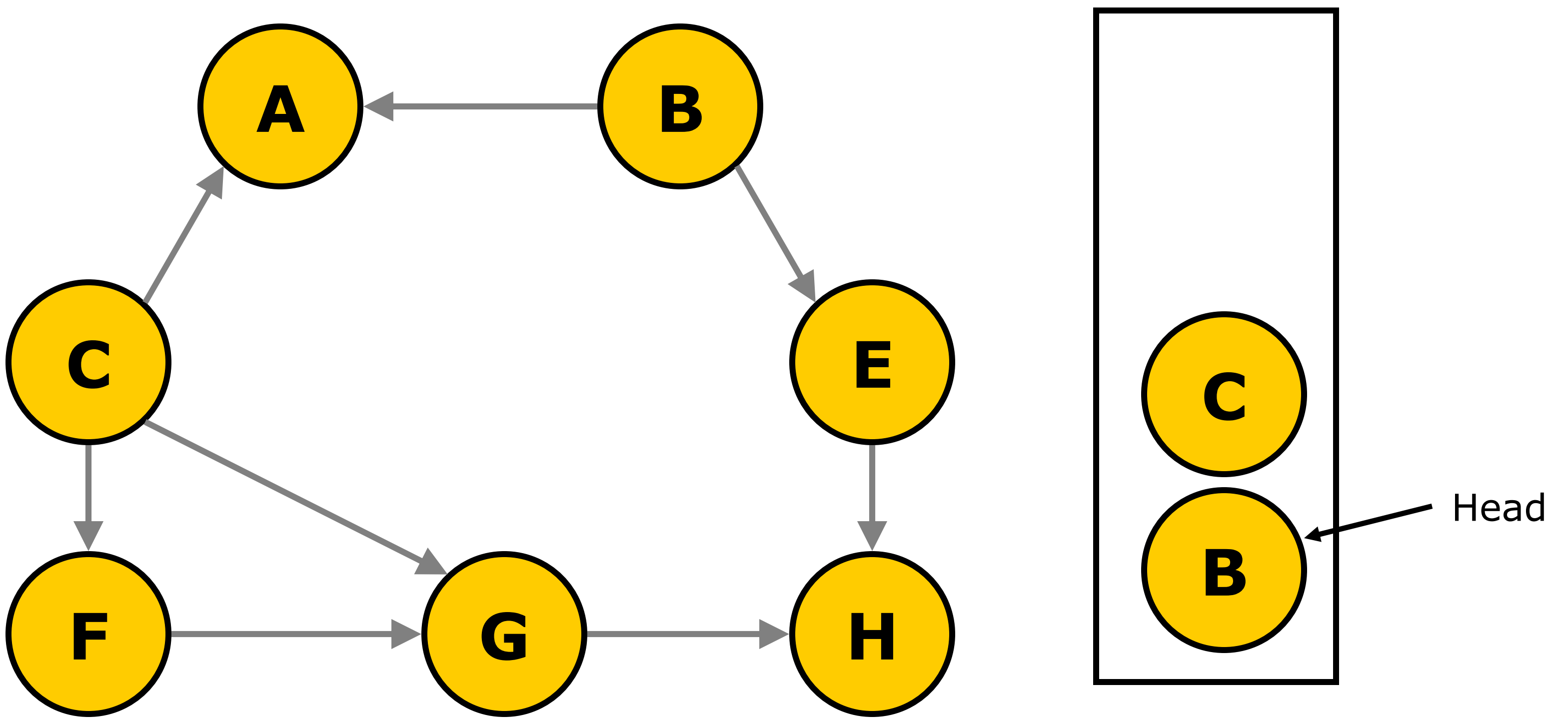
Example



Q: Which nodes have in-degree 0? D! The Queue contains D initially.

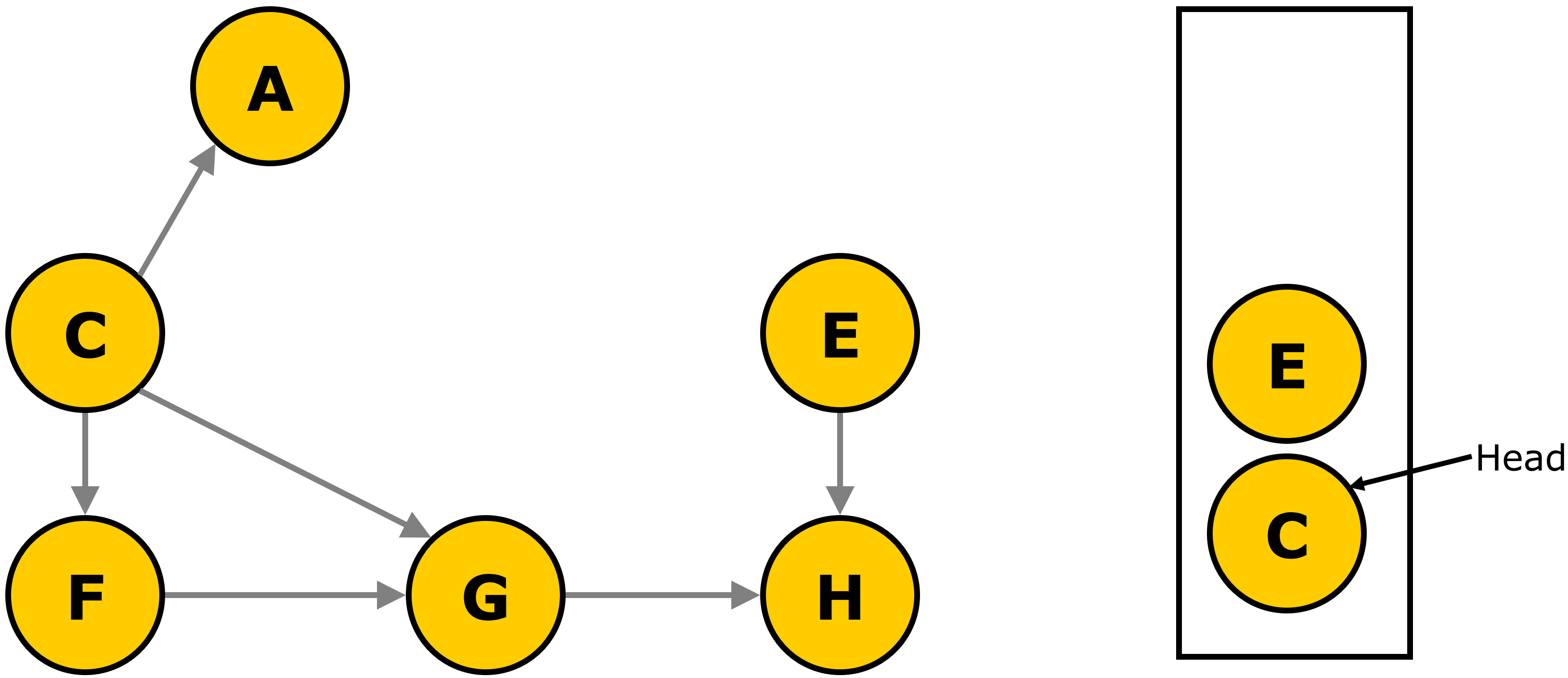
We dequeue D, print it and update the in-degrees of its neighbours.

Output: D

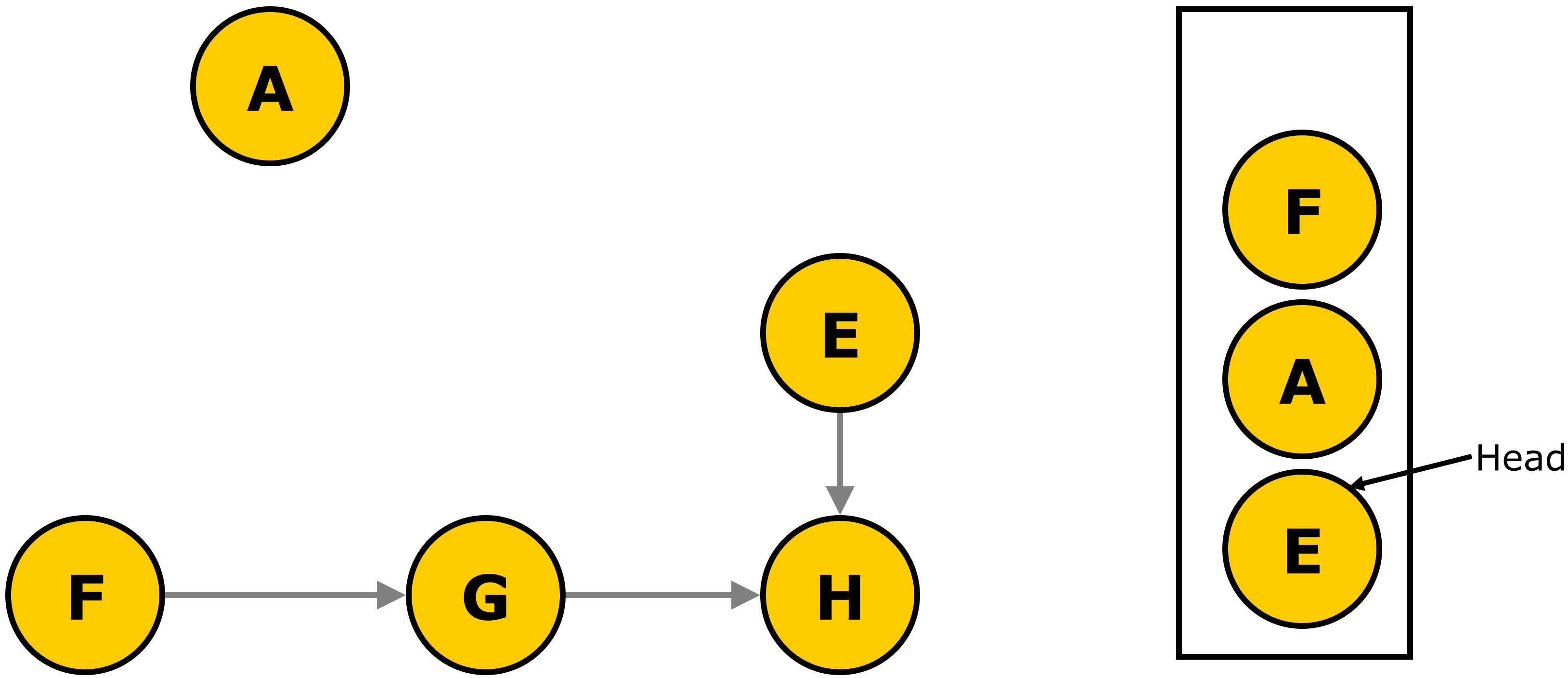


- Both nodes B and C now have in-degree 0, the order to enqueue them into the queue will affect the their output order.
- In this example, we enqueue B then C.

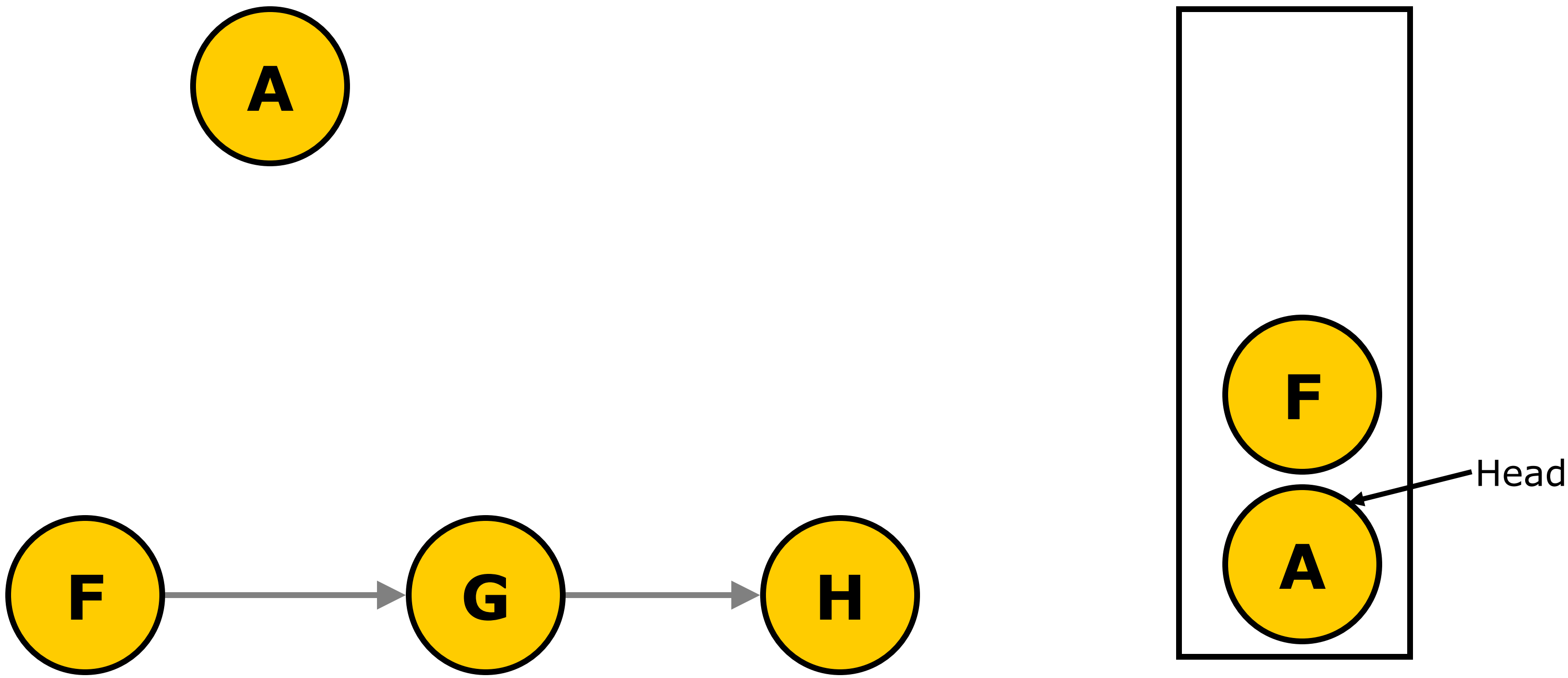
Output: DB



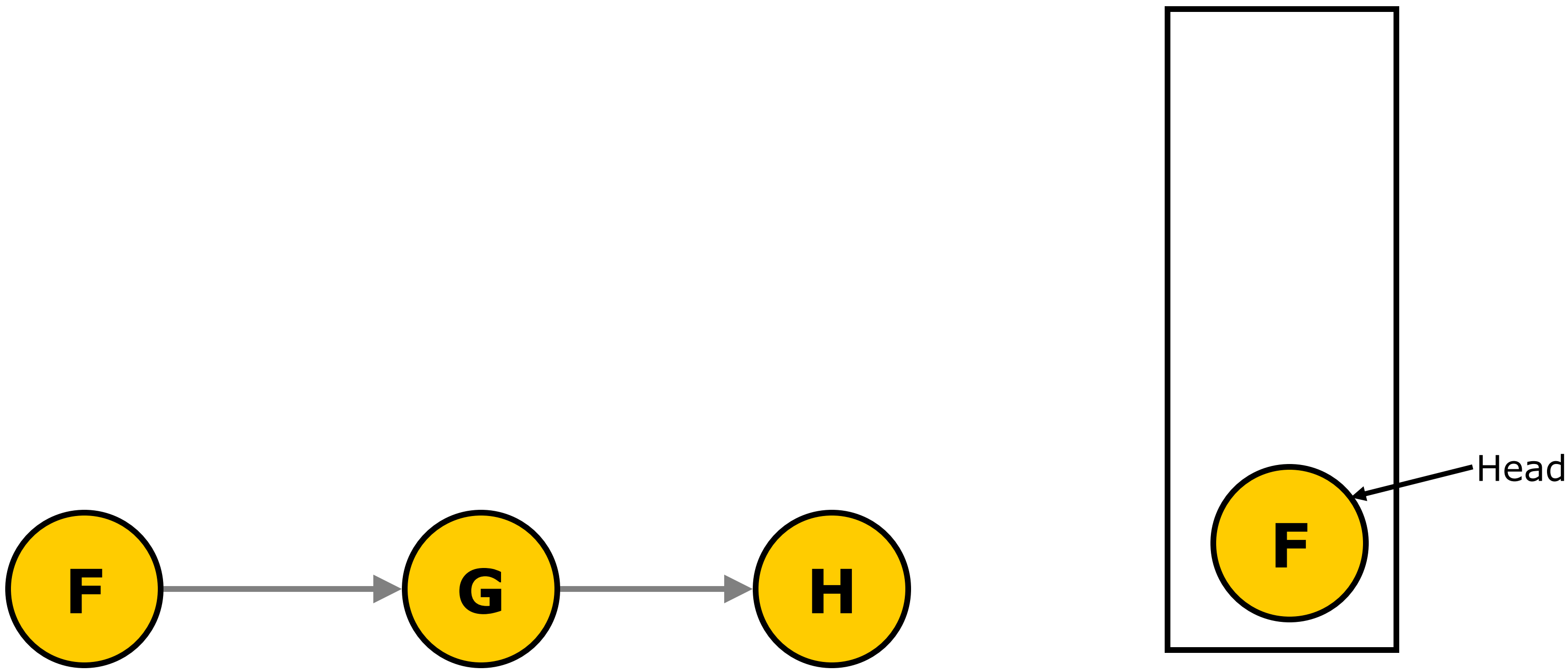
Output: DBC



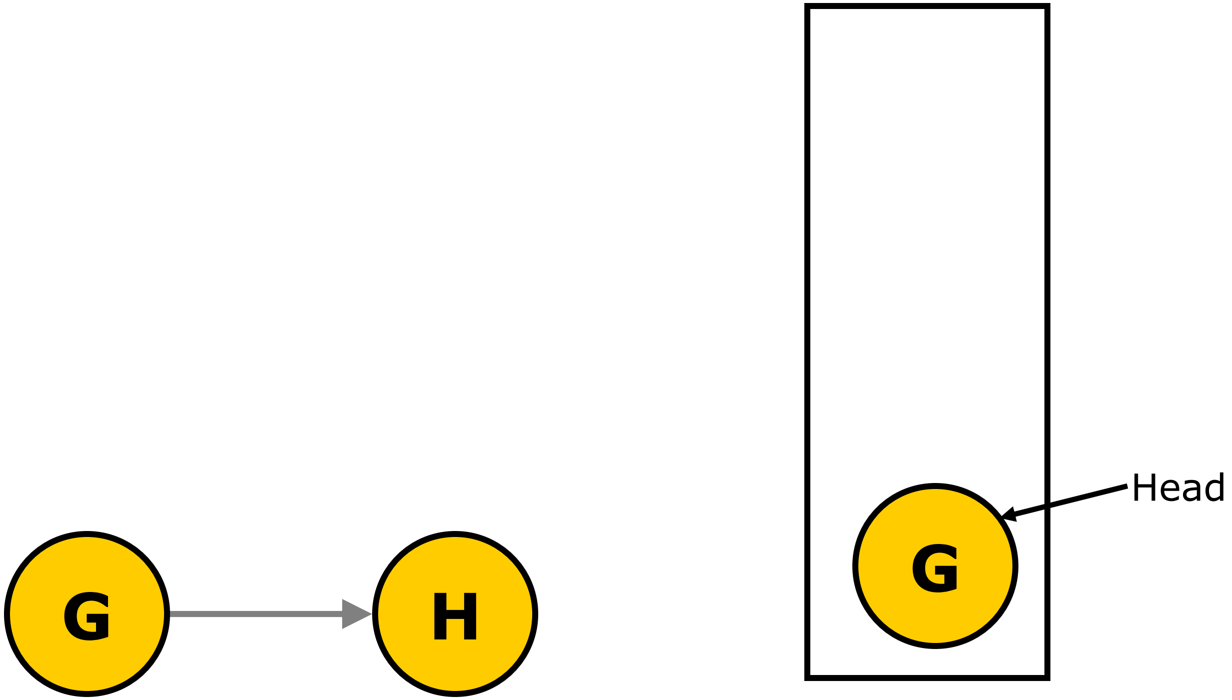
Output: DBCE



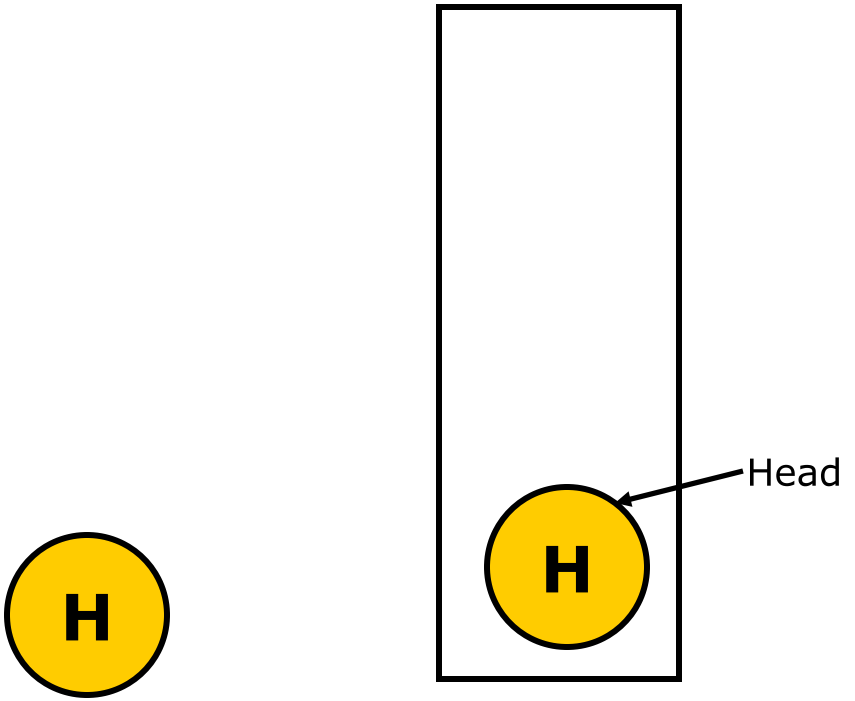
Output: DBCEA



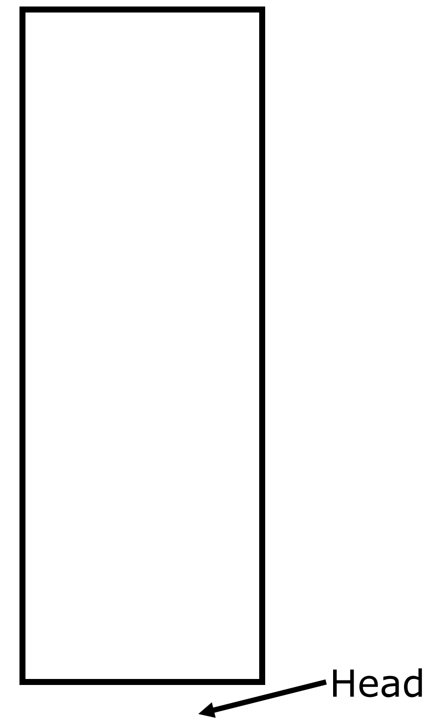
Output: DBCEAF



Output: DBCEAFG



Output: DBCEAFGH



(queue is empty)

Pseudocode for Toposort

```
q = new Queue()
put all vertices with in-degree 0 into q
while q is not empty
  v = q.dequeue()
  print v
  remove v together with its edges from G
          (i.e. need to re-compute the in-degrees of all neighbours of v)
  enqueue neighbours of v with in-degree 0
```

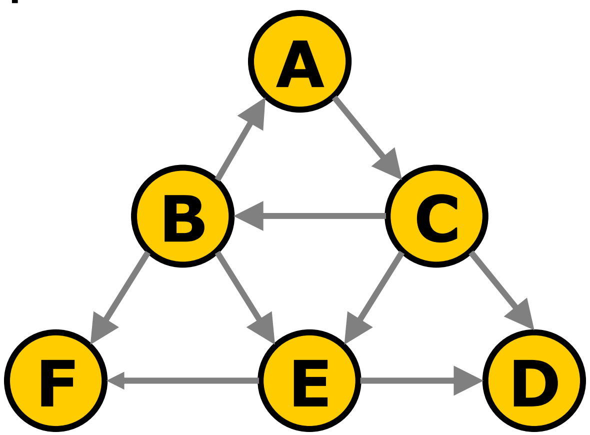
Question: What is the complexity of toposort?

Shortest Path

Note: This topic on shortest path (slides 81 to 112) is not covered in the examination.

Definitions

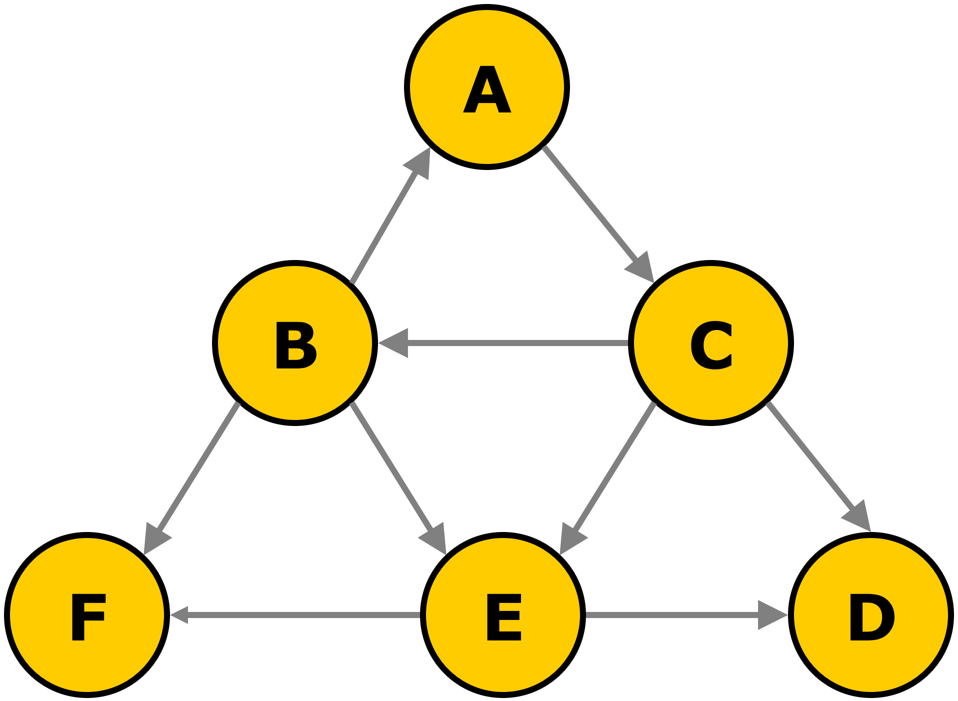
- A path on a graph G is a sequence of vertices v_0, v_1, v_2, .. v_n where $(v_i,v_{i+1}) \in E$
- The cost of a path is the sum of the cost of all edges in the path.



- In single-source shortest path problem, we are given a vertex v, and we want to find the path with minimum cost to **every** other vertices.

Unweighted shortest path

If a graph is unweighted, we can treat the cost of each edge as **1**.



ShortestPath(s)

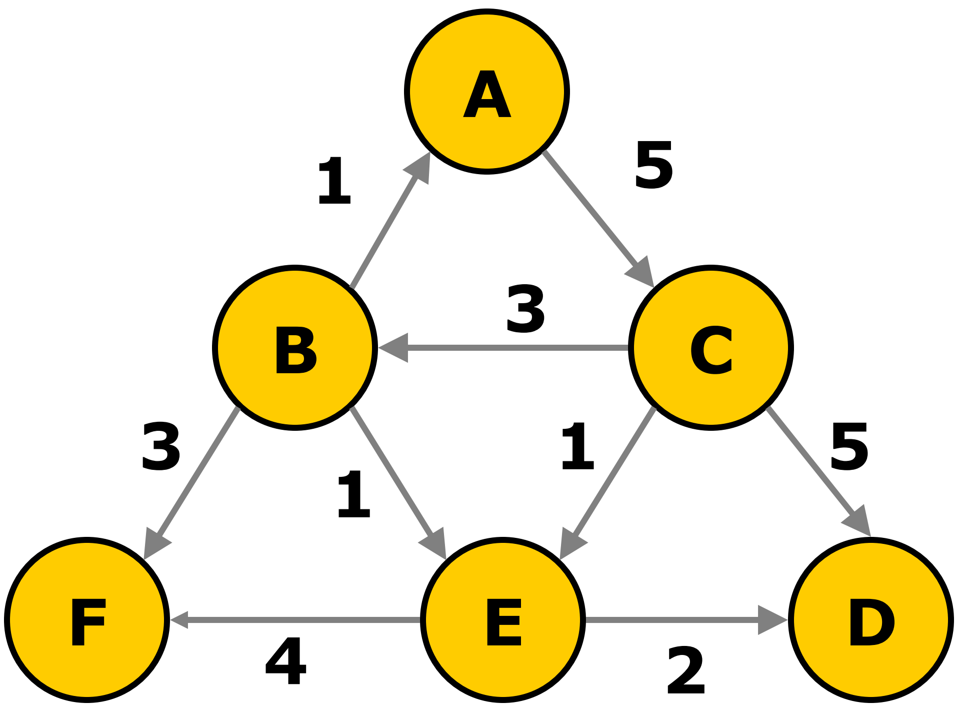
- The shortest path for an unweighted graph can be found using BFS.
- Run BFS(s) where s is the chosen source node
  - w.level: shortest distance from s
  - w.parent: shortest path from s

    by tracing back the **parent**

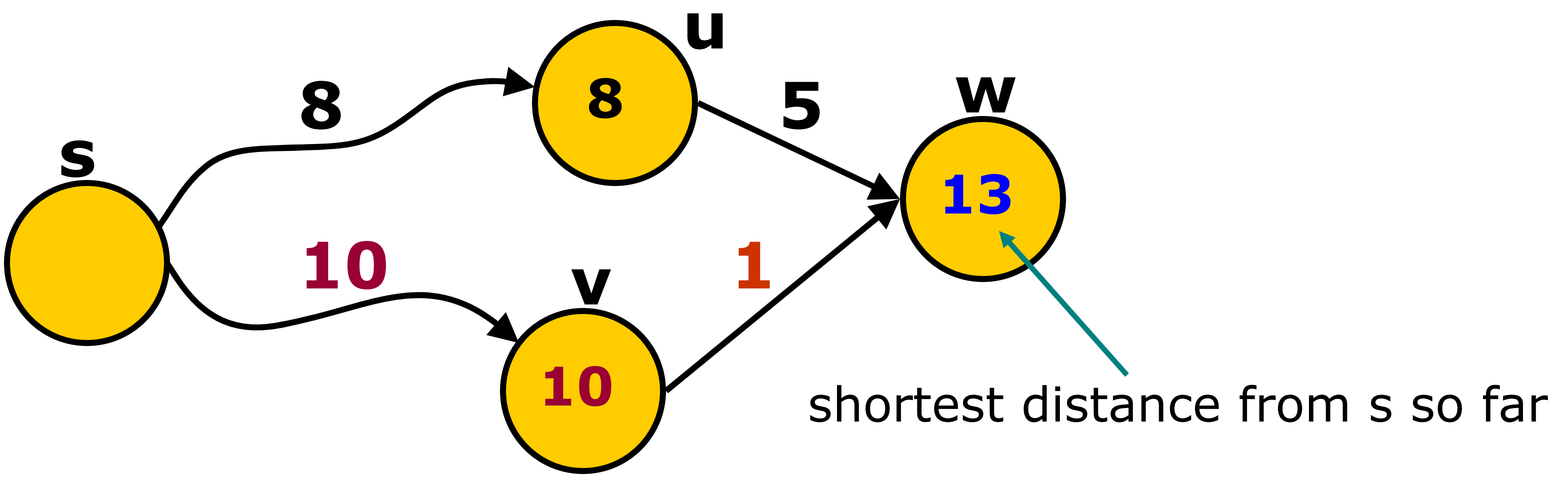
    pointer from w back to s.

Question: Why does BFS guarantee shortest paths?

Positive weighted shortest path

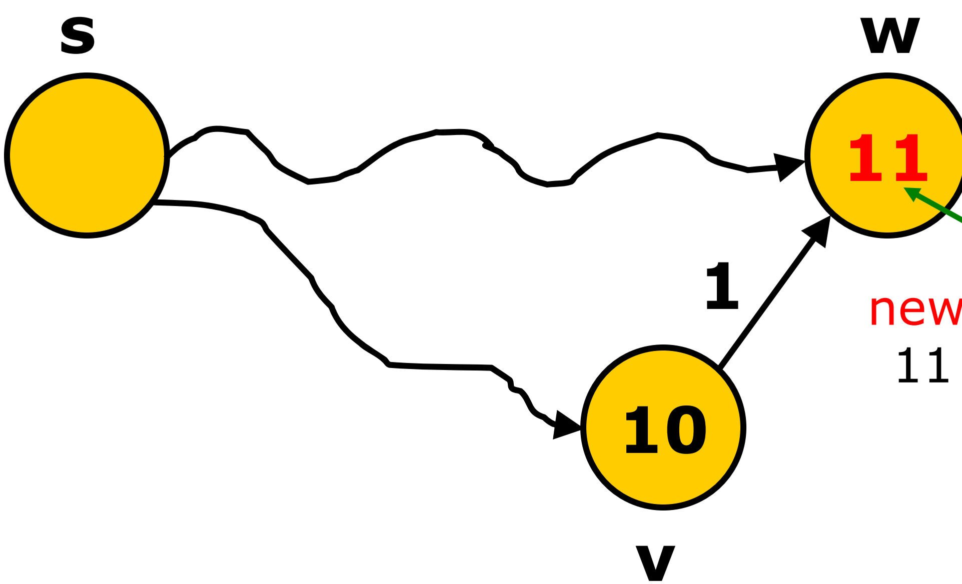


BFS(s) does not work



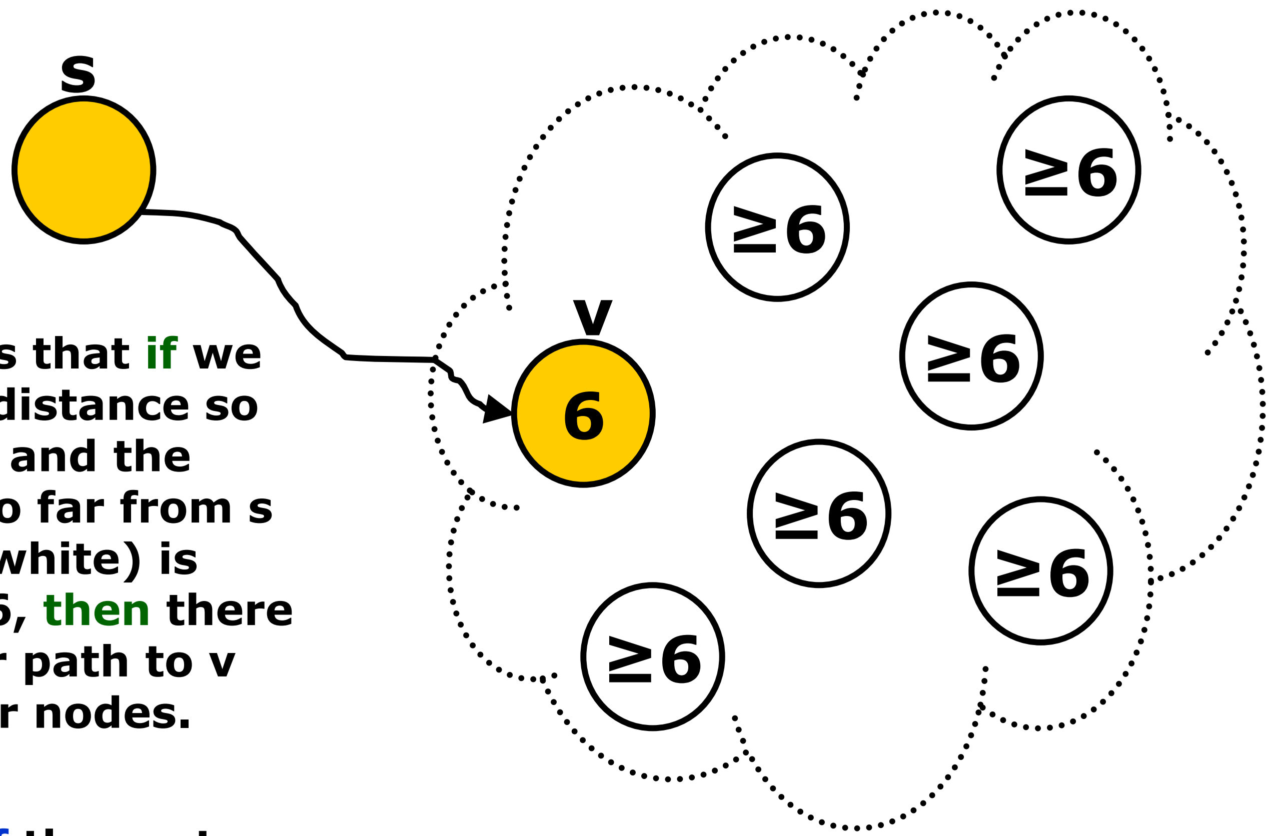
- Must keep track of shortest distance from the source node so far for each node
- Observation 1: If we found a new shorter path, update the distance.

Observation 1



new shortest distance is 11 (the value was 13)

Observation 2 (for positive costs only)



- The second idea is that if we know the shortest distance so far from s to v is 6, and the shortest distance so far from s to other nodes (in white) is bigger or equal to 6, then there cannot be a shorter path to v through these other nodes.

- This is true only if the costs are positive!

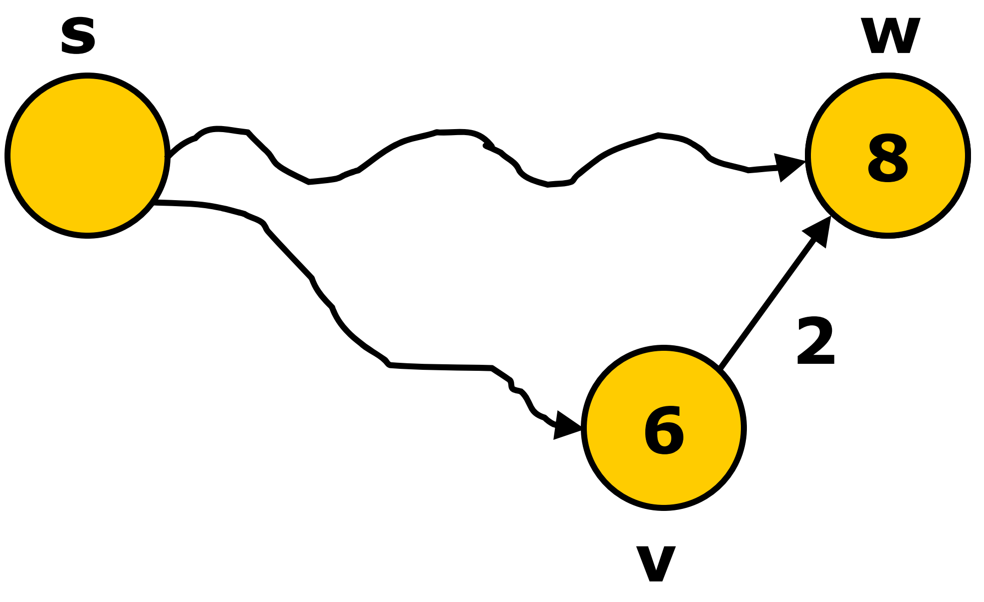
Definitions

distance(v) : shortest distance so far from s to v

parent(v) : previous node on the shortest path so far from s to v

weight(u, v) : the weight (cost) of edge from u to v

Example



distance(w) = 8
weight(v,w) = 2
parent(w) = v

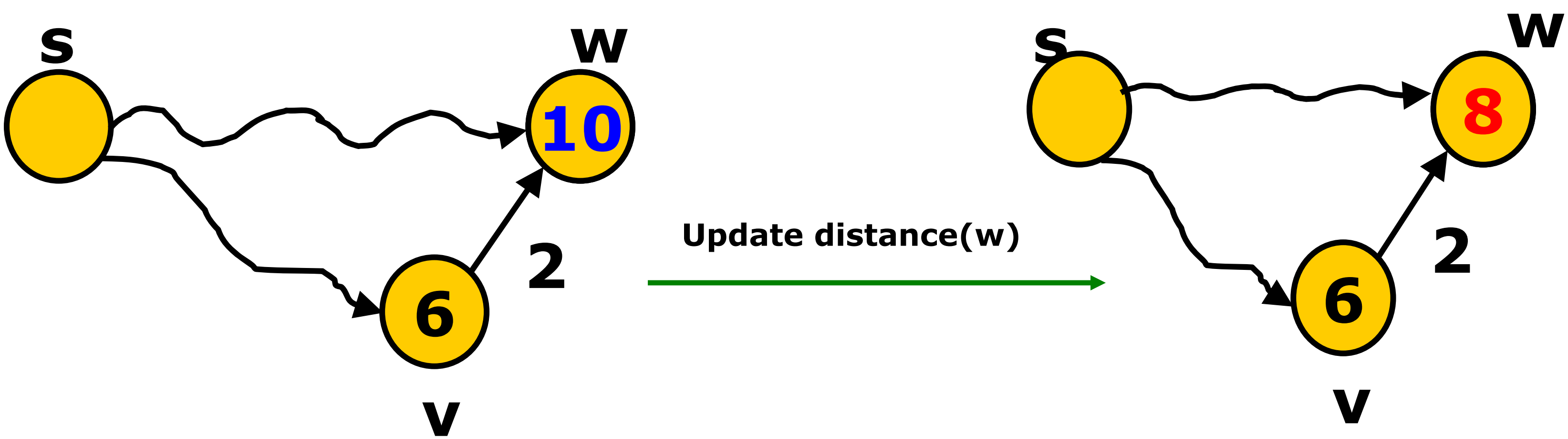
Relax(v,w) - based on observation 1.

d = distance(v) + weight(v,w)

if distance(w) > d **then** // found a new shorter distance

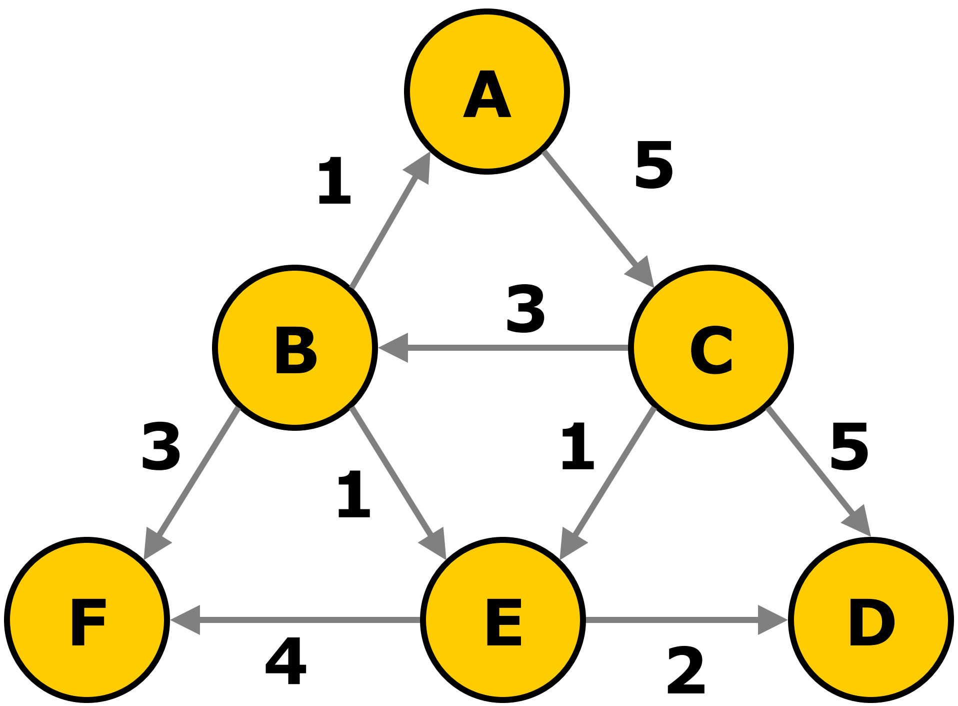
distance(w) = d // update the distance and parent

parent(w) = v



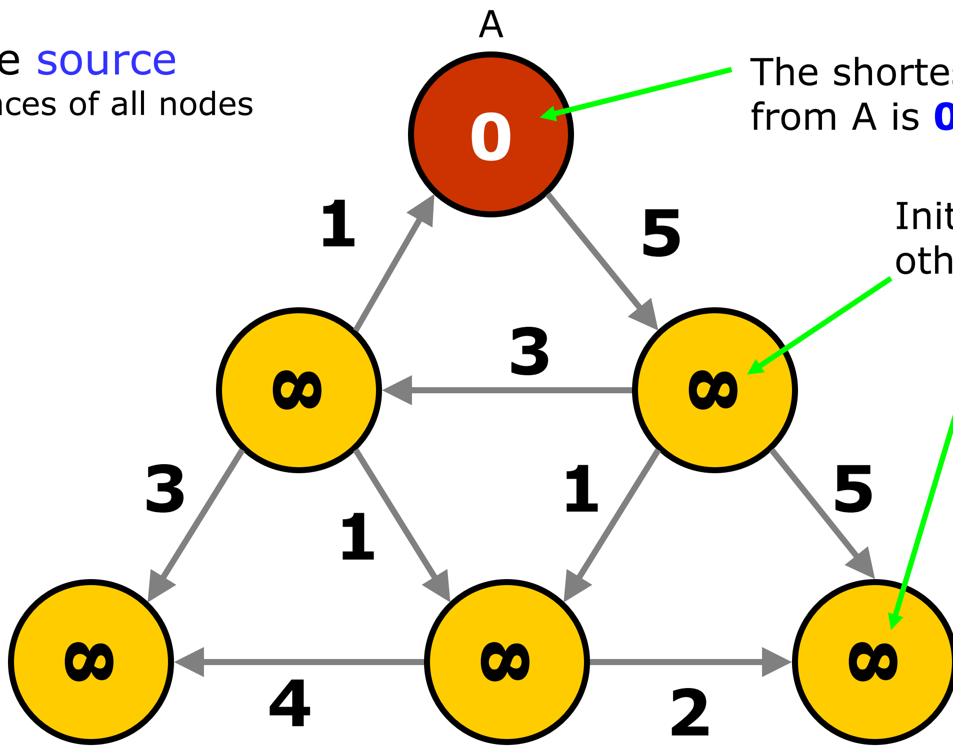
Dijkstra's algorithm

- **shortest path algorithm** for graphs with **positive weights**



Dijkstra’s algorithm

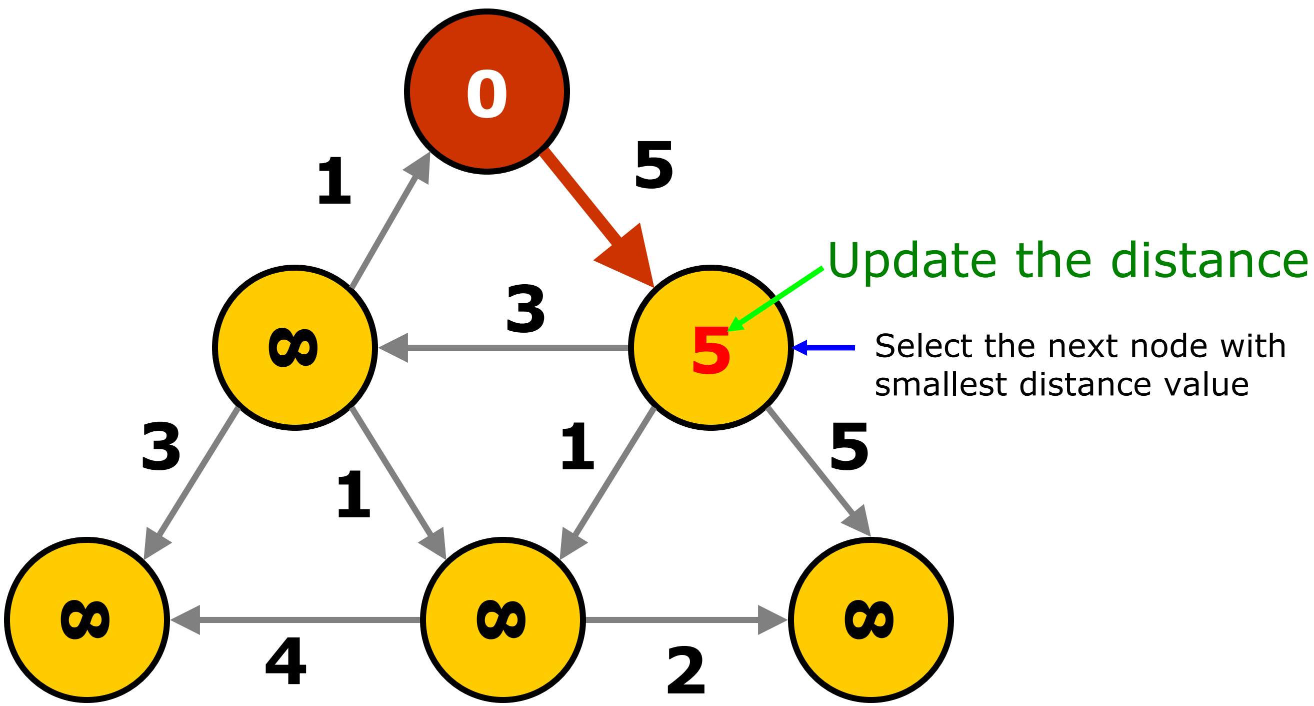
Use node A as the source
i.e. find shortest distances of all nodes from source node A



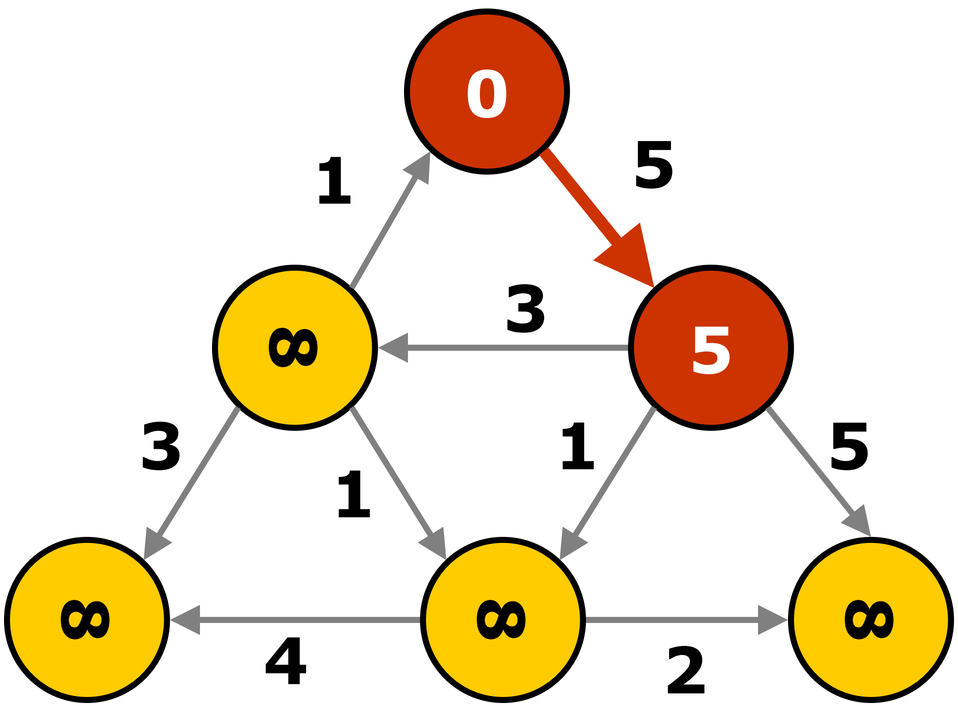
The shortest distance of A from A is **0**

Initialize the distances of others nodes as **infinite**

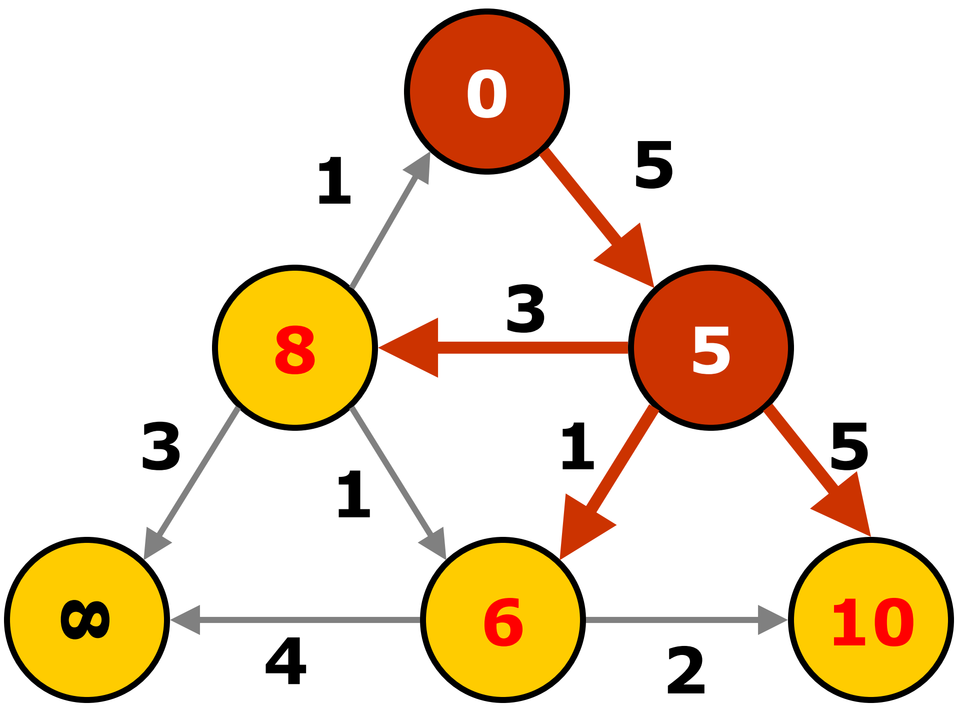
Dijkstra’s algorithm



Dijkstra’s algorithm

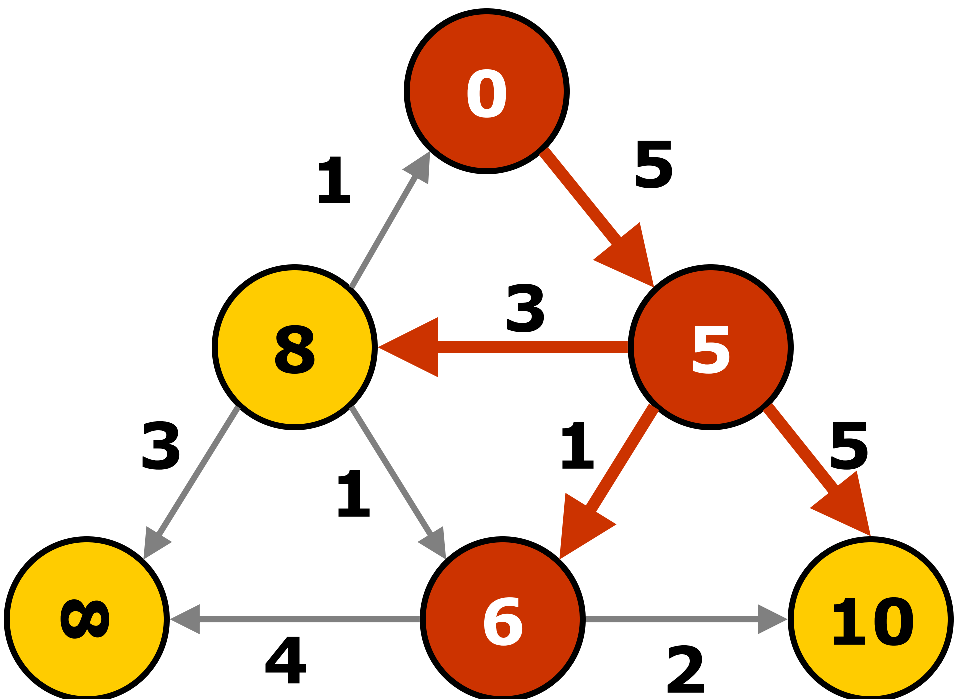


Dijkstra's algorithm



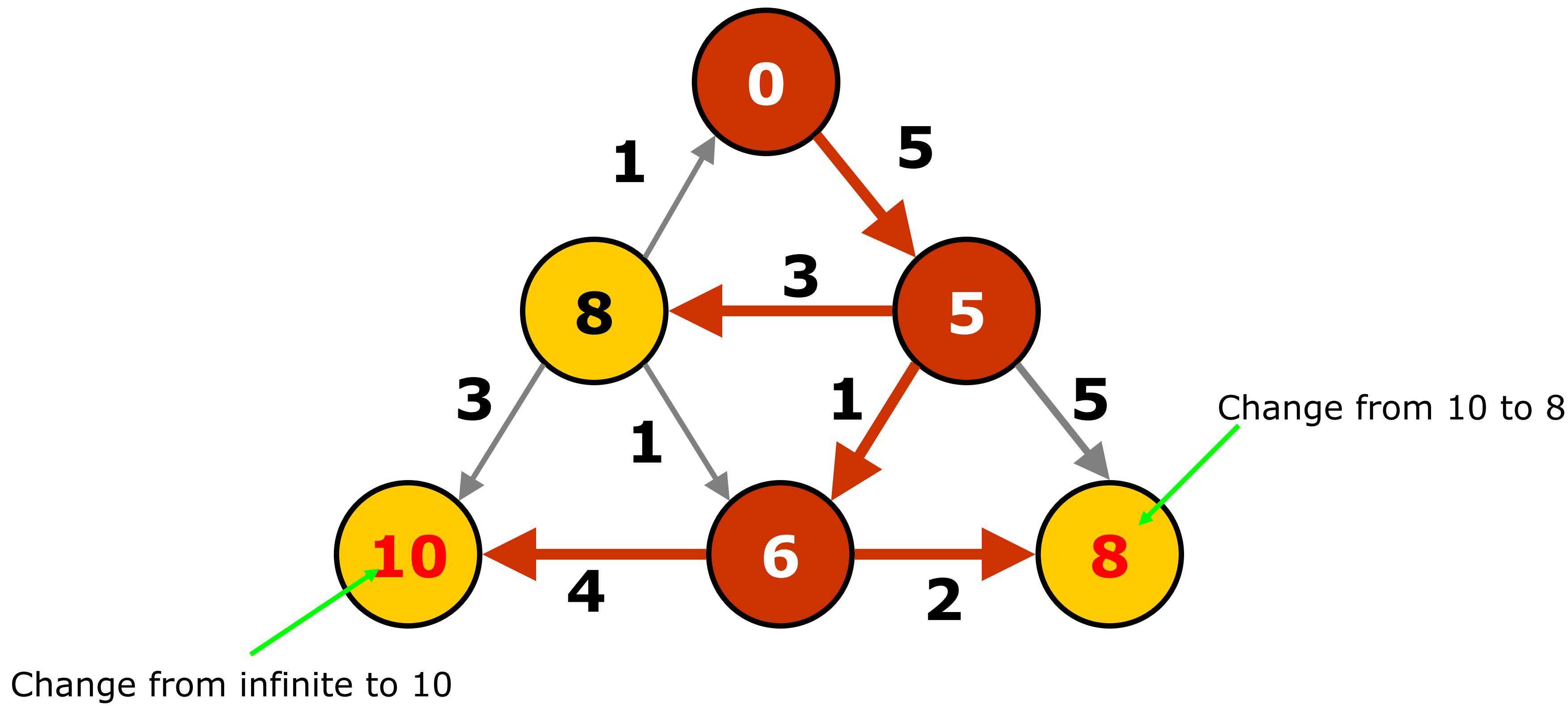
(1) Update distance for 3 nodes

Dijkstra’s algorithm

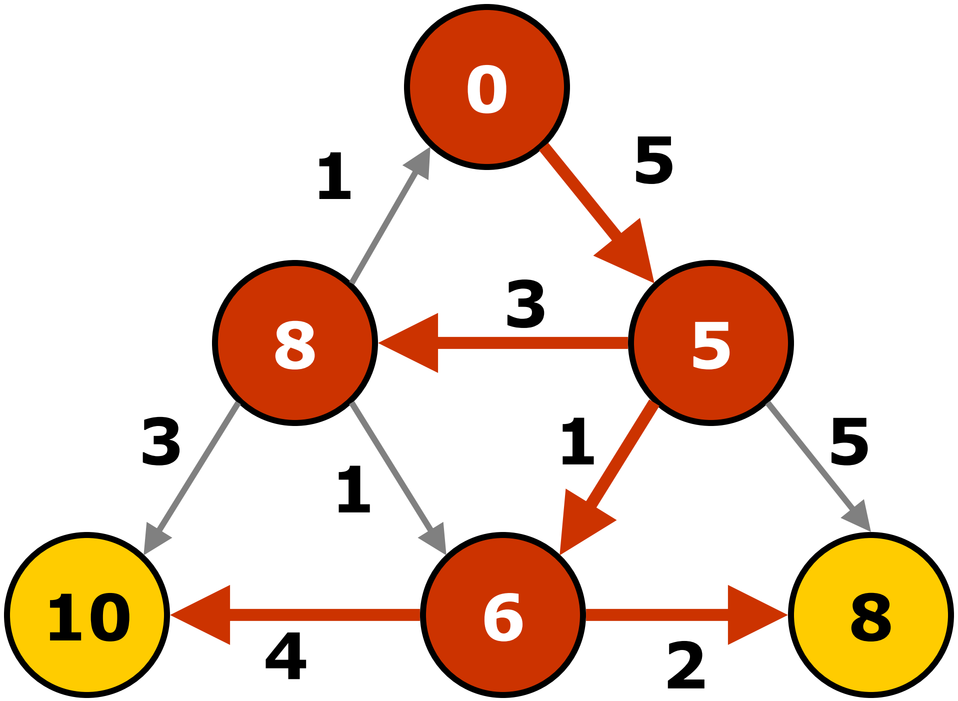


(2) Select the next node with smallest distance value

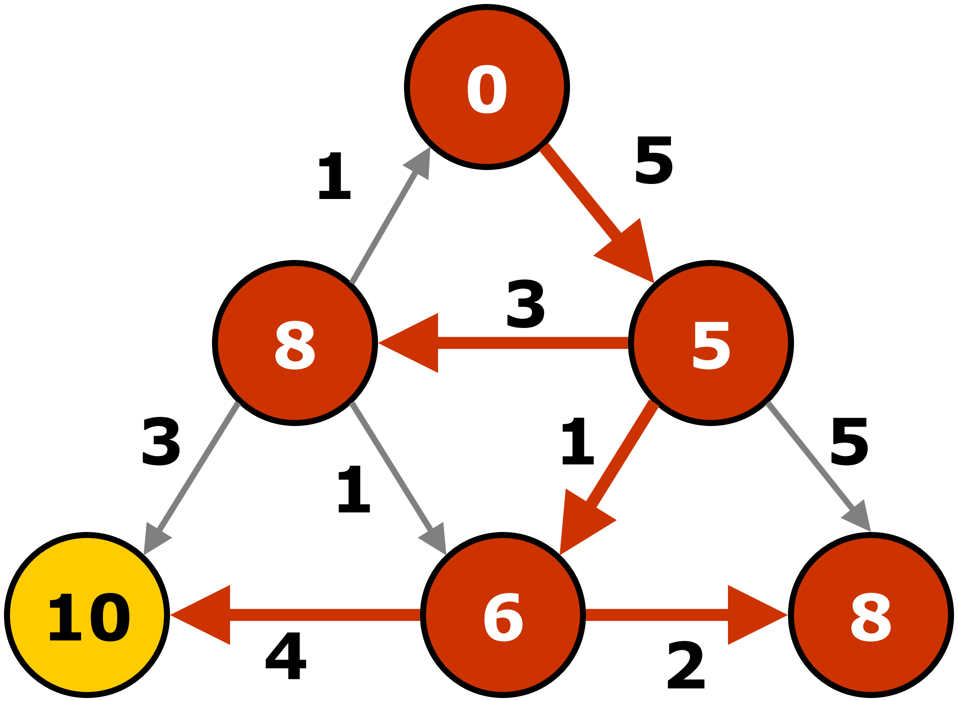
Dijkstra's algorithm



Dijkstra's algorithm

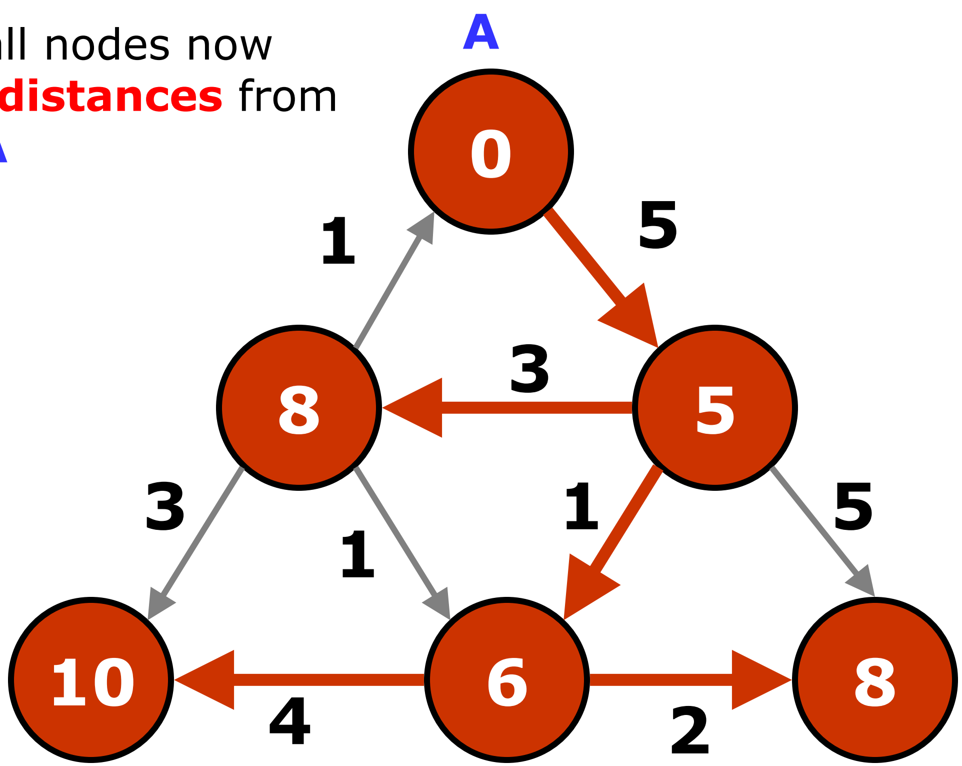


Dijkstra's algorithm



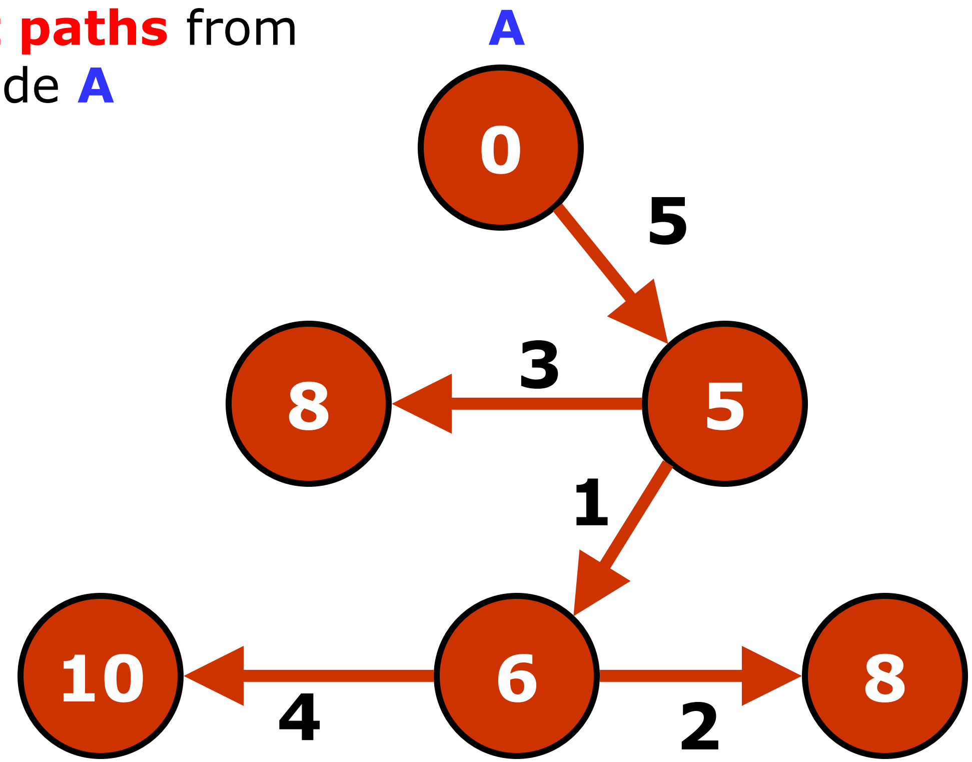
Dijkstra's algorithm

The distances of all nodes now are the **shortest distances** from the source node **A**



Dijkstra's algorithm

The **shortest paths** from the source node **A**



Dijkstra's algorithm

color all vertices yellow

// yellow nodes are those not yet processed

foreach vertex w

distance(w) = INFINITY

distance(s) = 0 //source node distance is 0

Dijkstra's algorithm

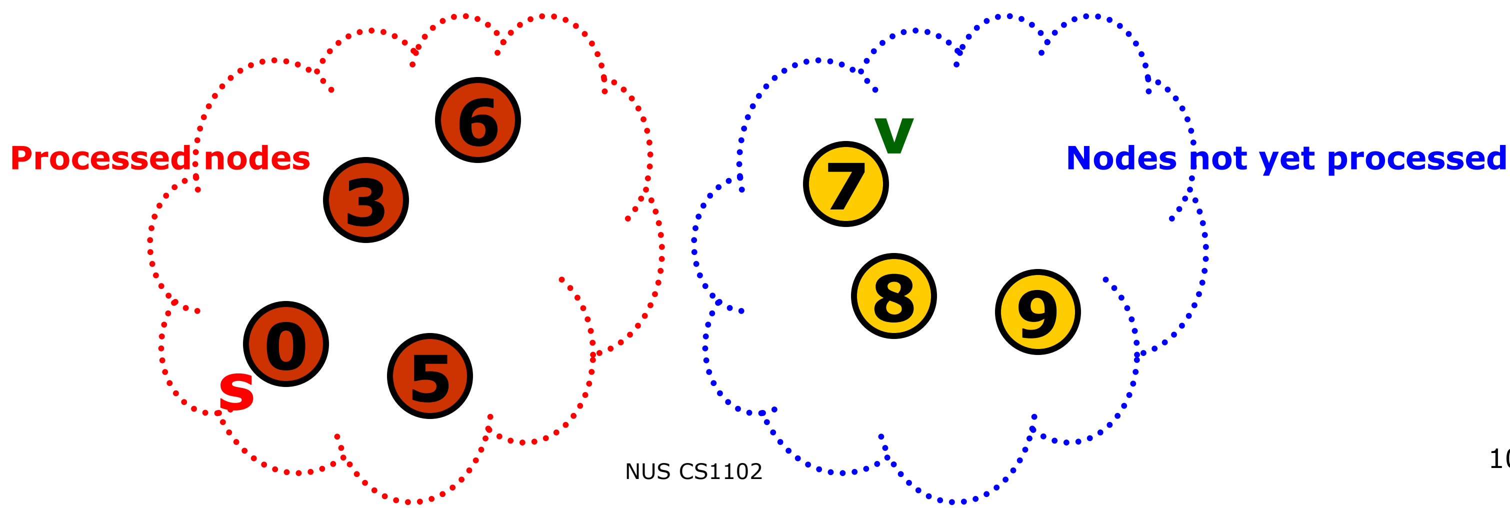
while there are yellow vertices //unprocessed nodes are yellow

v = yellow vertex with min distance(v)

color v red // red vertices are vertices with shortest distances from s found

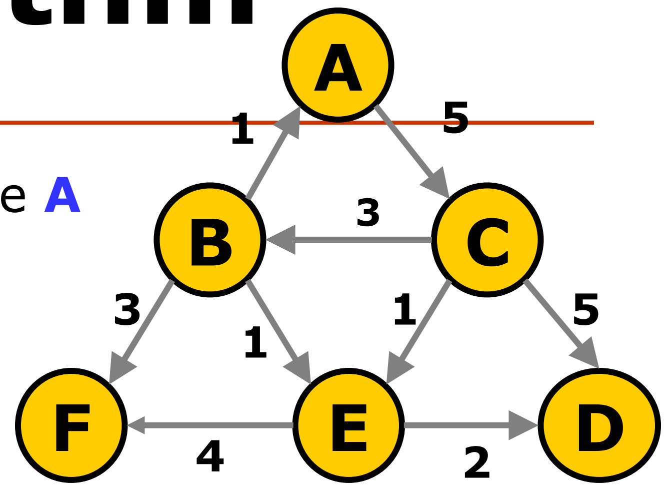
foreach yellow neighbour w of v

relax(v,w) // relax algorithm from **observation 1** slides



Dijkstra's algorithm

E.g. The **shortest paths** from the source node **A**



step	**node**	**S (nodes processed)**	A	B	C	D	E	F
1	A	A	0	∞	5	∞	∞	∞
2	C	AC	0	8	5	10	6	∞
3	E	ACE	0	8	5	8	6	10
4	B	ACEB	0	8	5	8	6	10
5	D	ACEBD	0	8	5	8	6	10
6	F	ACEBDF	0	8	5	8	6	10

Running time $O(V^2 + E)$

```
color all vertices yellow
foreach vertex w
  distance(w) = INFINITY
distance(s) = 0
while there are yellow vertices
  v = yellow vertex with min distance(v)
  color v red
  foreach yellow neighbour w of v
      relax(v,w)
```

Running time $O(V^2 + E)$

- Initialization takes O(V) time. // V = no of nodes
- Picking the vertex with minimum distance(v) can take O(V) time, and relaxing the neighbours take O(adj(v)) time. // adj(v) = adjacent nodes of v
- The sum of these over all vertices is $O(V^2+E)$.
  // Because sum adj(v) = E where E is the no of edges
- We can improve this if we can improve the running time for picking the minimum distance(). Yes, use priority queue to pick the minimum.

Using priority queue

```
foreach vertex w
    distance(w) = INFINITY
distance(s) = 0
pq = new PriorityQueue(V)        // minimum heap
                  //with all vertices and their distances (as keys)
while pq is not empty
   v = pq.deleteMin()     // O(log V)
   foreach neighbour w of v
       relax(v,w)
```

Initialization O(V)

```
foreach vertex w
   distance(w) = INFINITY
distance(s) = 0
pq = new PriorityQueue(V)
                    // with all vertices and their distances (as keys)
```

Main loop

```
while pq is not empty
   v = pq.deleteMin()
   foreach neighbour w of v
        relax(v,w)
```

Note: Need to expand the relax(v,w) using priority queue

Main loop - O((V+E) log V)

```
while pq is not empty
   v = pq.deleteMin()                      // O(log V)
   foreach neighbour w of v                // adj(v)
      d = distance(v) + cost(v,w)
      if distance(w) > d then
         distance(w) = d
         pq.decreaseKey(w, d)              // O(log V)
         parent(w) = v
```

Note: complexity = sum (O(log V) + adj(v) * O(log V)) over all vertices
= O(V log V + E * log V) = O((V+E) log V)
since the total number of adjacent nodes of all nodes is the total number of edges.

Main loop – O((V+E) log V) (cont.)

- If we expand the code for relax(), we will see that we cannot simply update distance(v), since distance(v) is a key in the priority queue.
- Here, we use an operation called decreaseKey() that updates the key value of distance(v) in the priority queue.
- decreaseKey() can be done in O(log V) time. How?
- The running time for this new version of Dijkstra algorithm takes O((V+E)log V) time.