CS2040C Data Structures and Algorithms

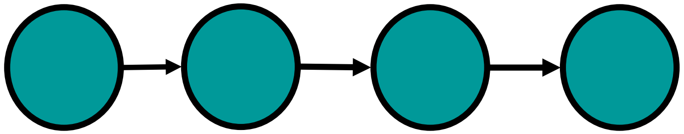
Graphs

Outline

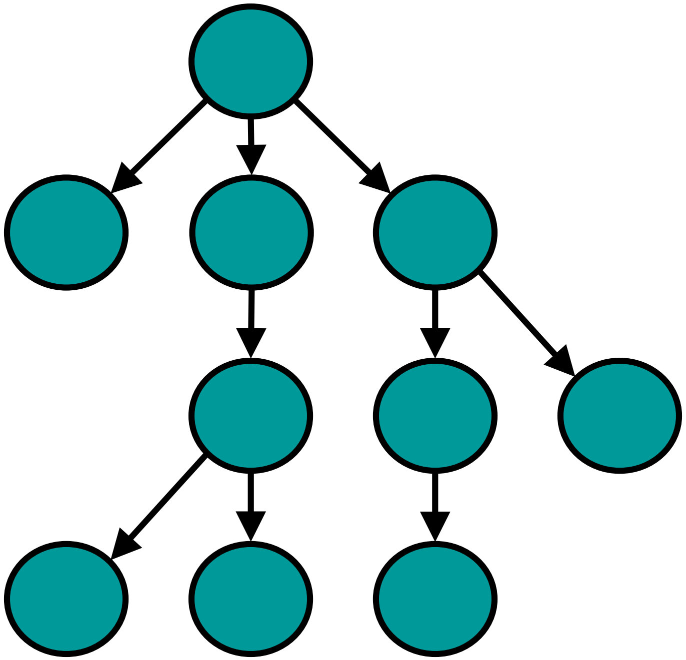
- Types of graphs
- Applications
- Implementation
  - Adjacency Matrix, Adjaceny List, Edge List
- Breadth First Search
- Depth First Search
- BFS/DFS Applications
- Directed Acyclic Graph
- Topological Sort

So far….

Linked list (linear data structure)

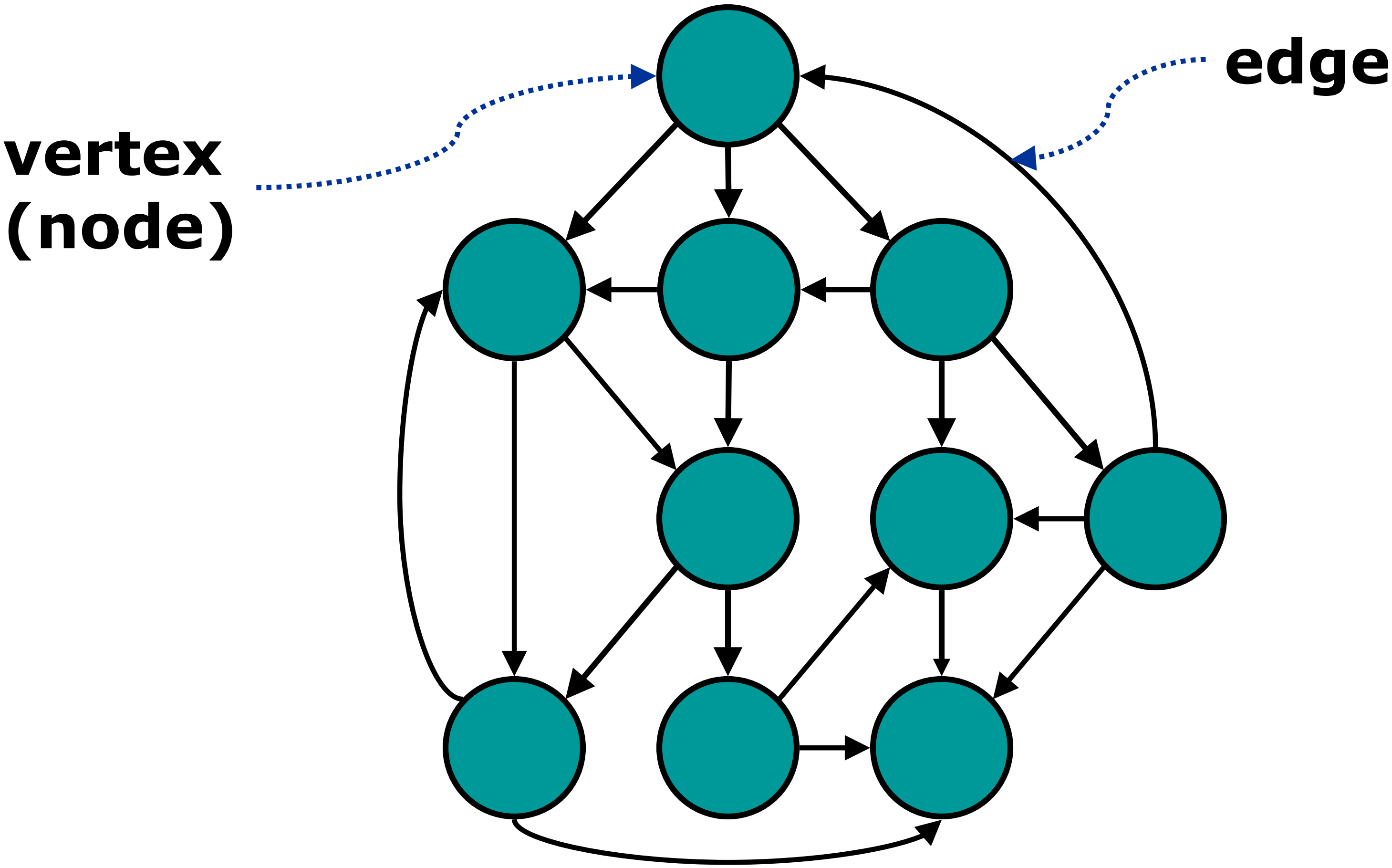


Tree (non-linear)



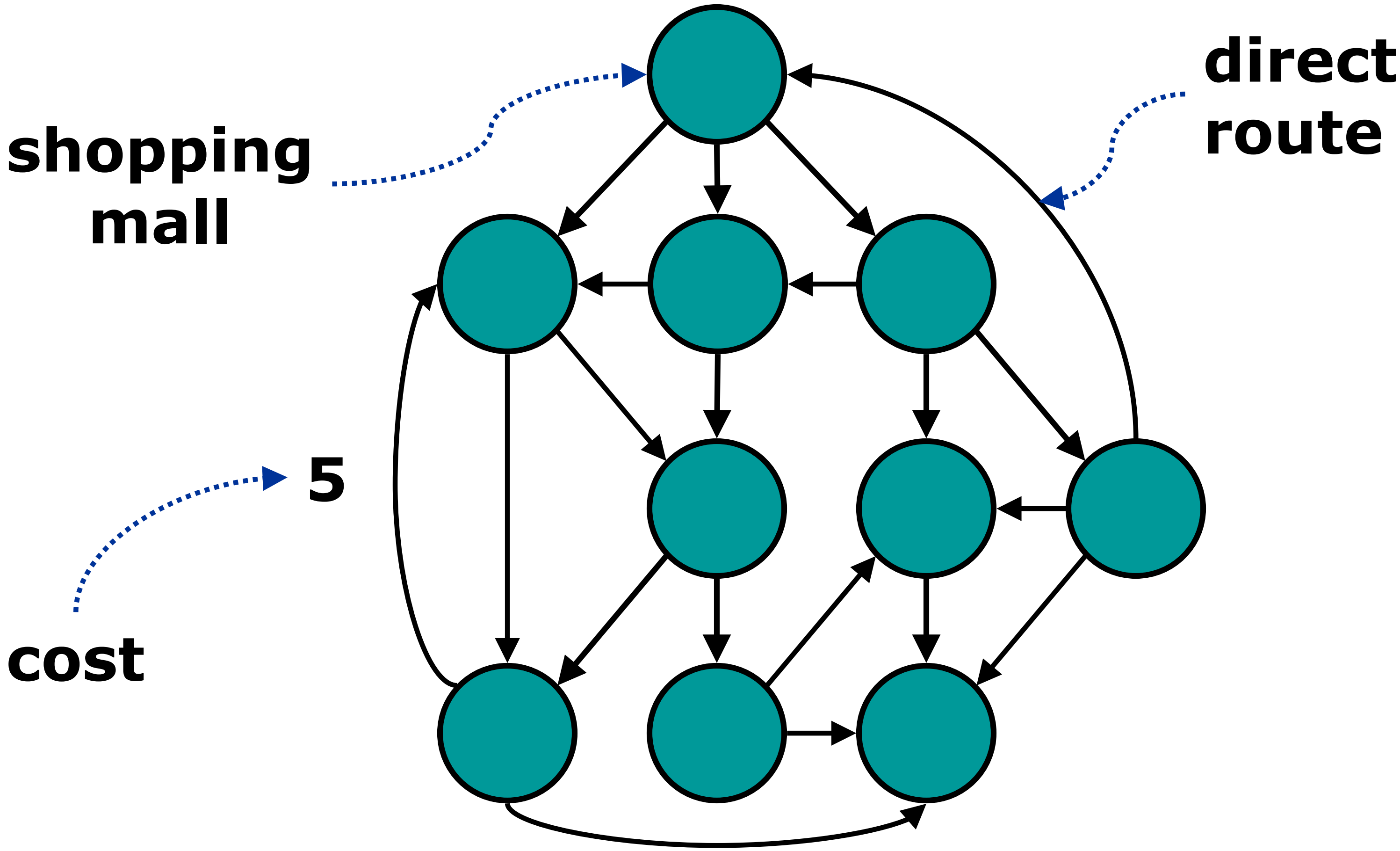
Directed graphs

A graph consists of a set of vertices and a set of edges between the vertices. In a tree, there is a unique path between any two nodes. In a graph, there may be more than one path between two nodes.



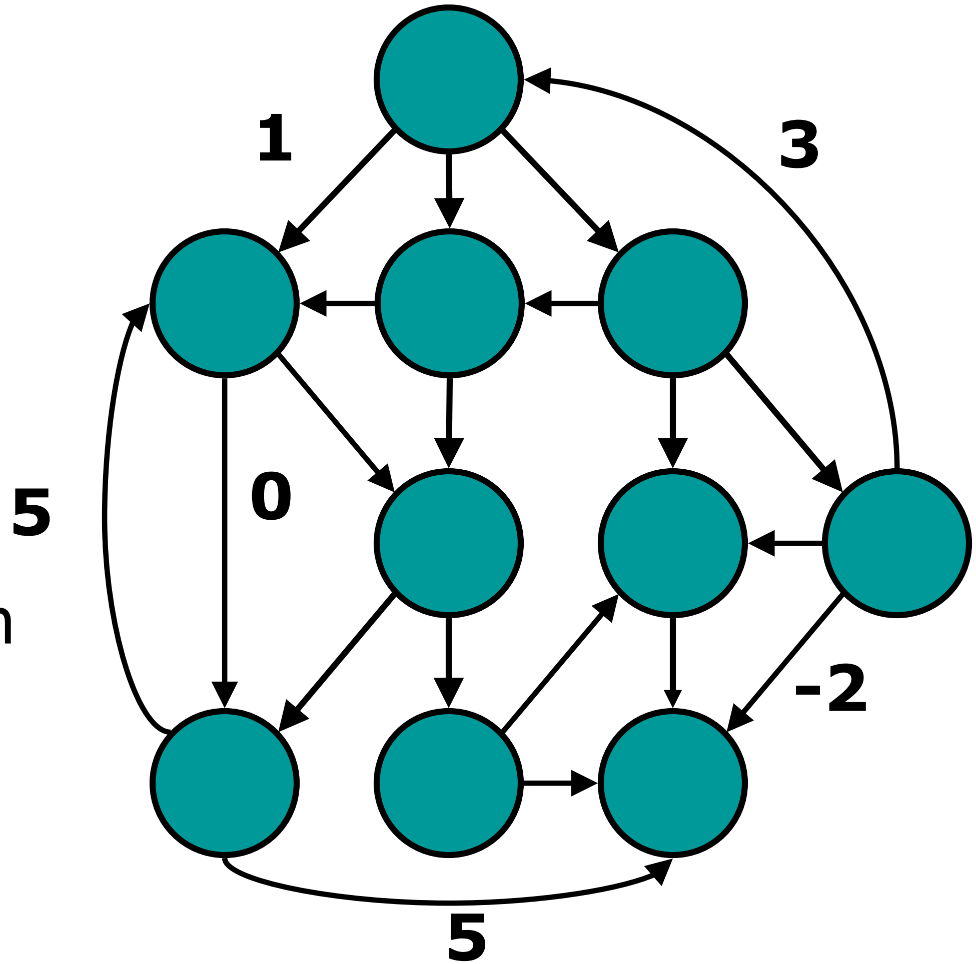
In a directed graph, edges are directed from one vertex to another

Example: travel planning



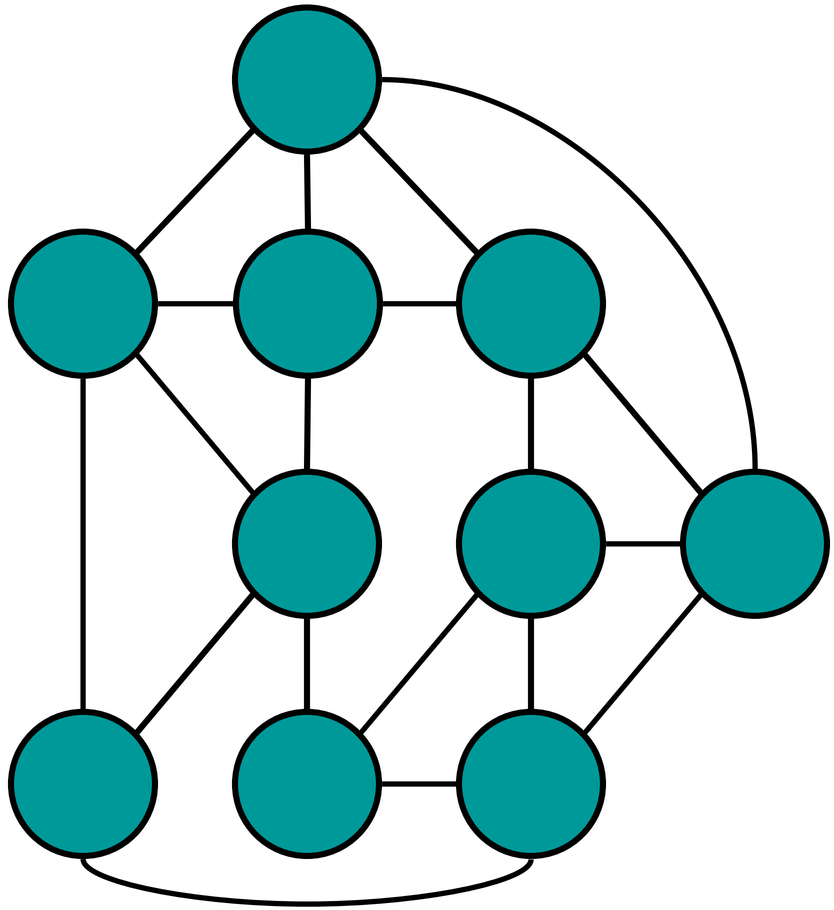
Weighted directed graph

In a weighted graph, edges have a weight (or cost) associated with it. Not all weights are labeled in this slide for simplicity.



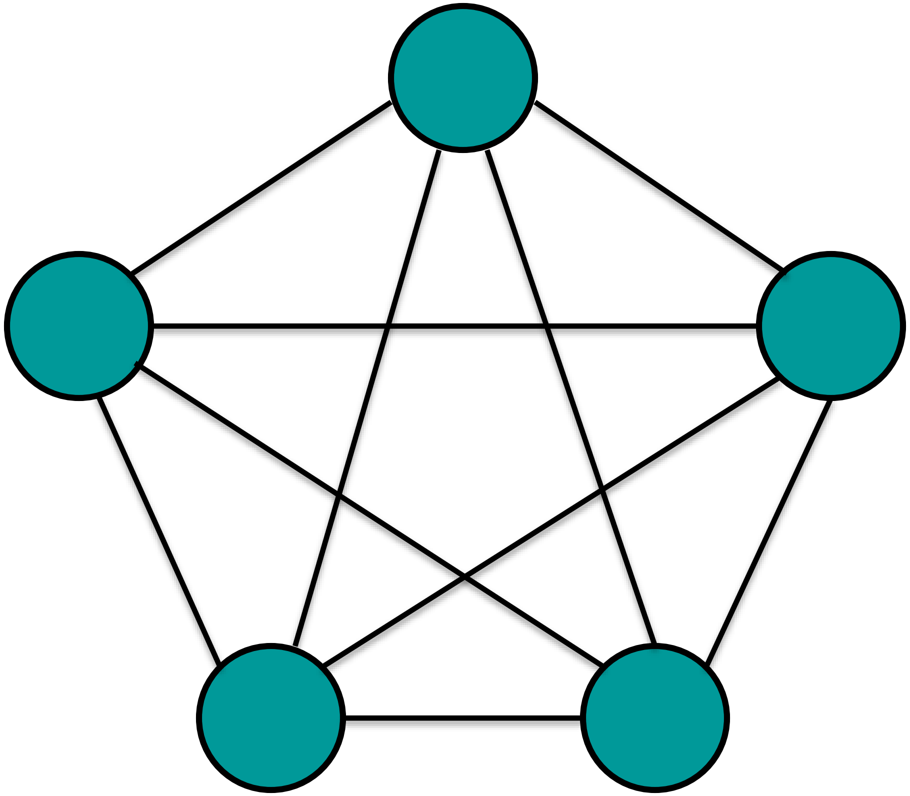
Undirected graph

- **edges are bidirectional**



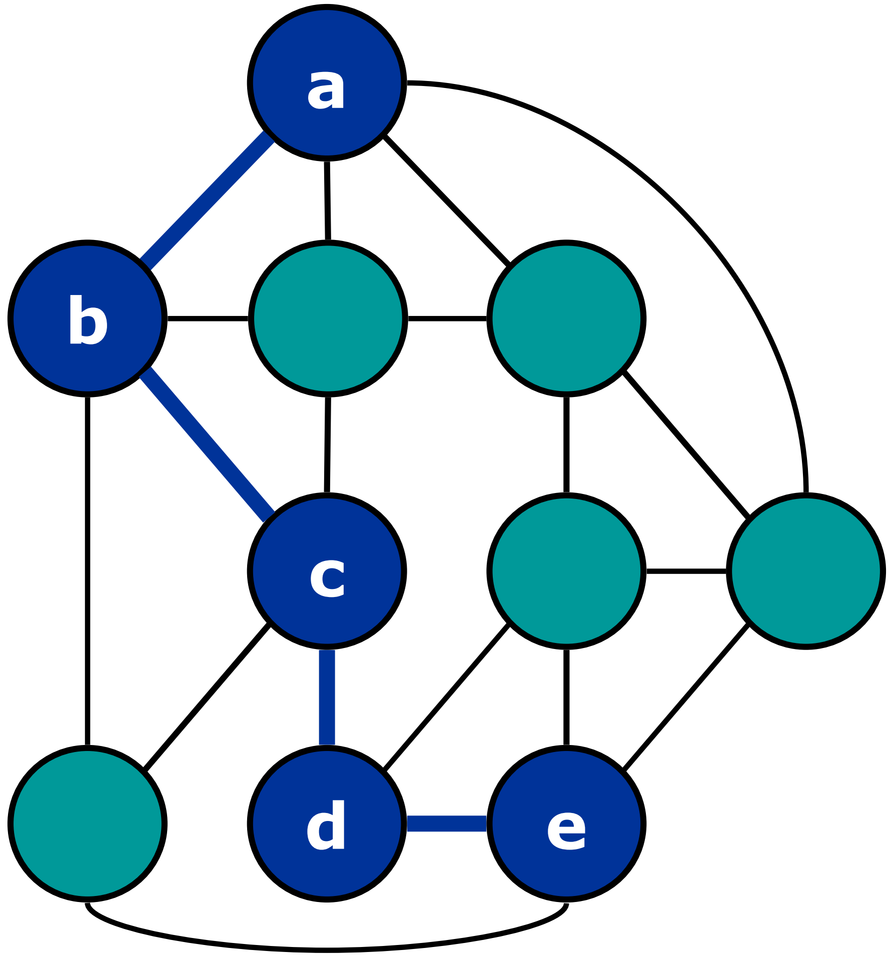
Complete graph

- A graph is complete if every pair of vertices has an edge between them.
- The number of edges in a complete graph is $V(V-1)/2$, where V is the number of vertices. Therefore, the number of edges is $O(V^2)$. A complete graph is also called a clique.



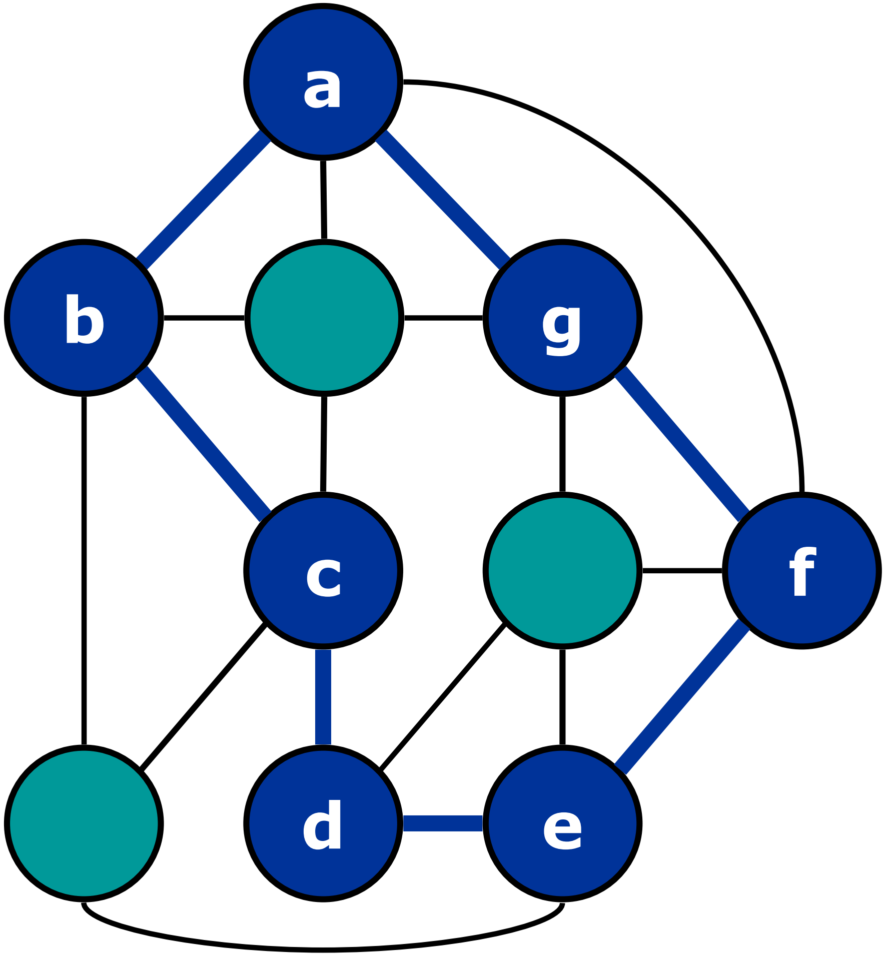
Path

- A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another
- The **length** of a path p is the number of edges in p.
- A **simple path** never visits the same vertex more than once



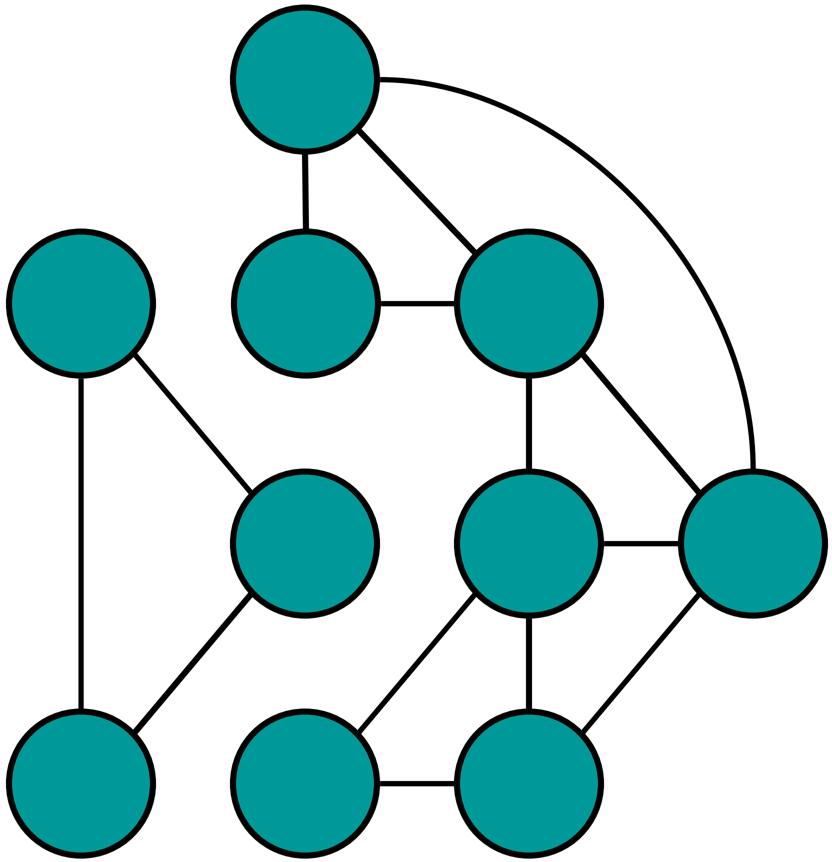
Cycle

- 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 definition of path and cycle applies to directed graph as well



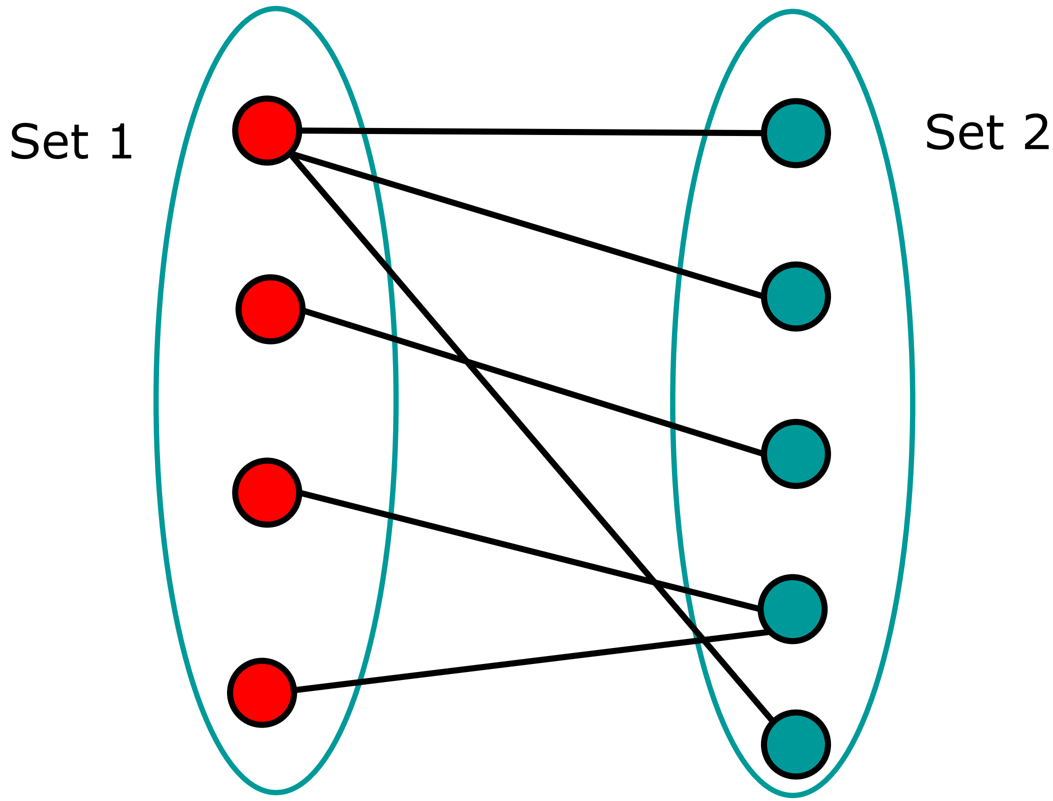
Disconnected graph

- A graph does not have to be connected
- The graph below has two connected components



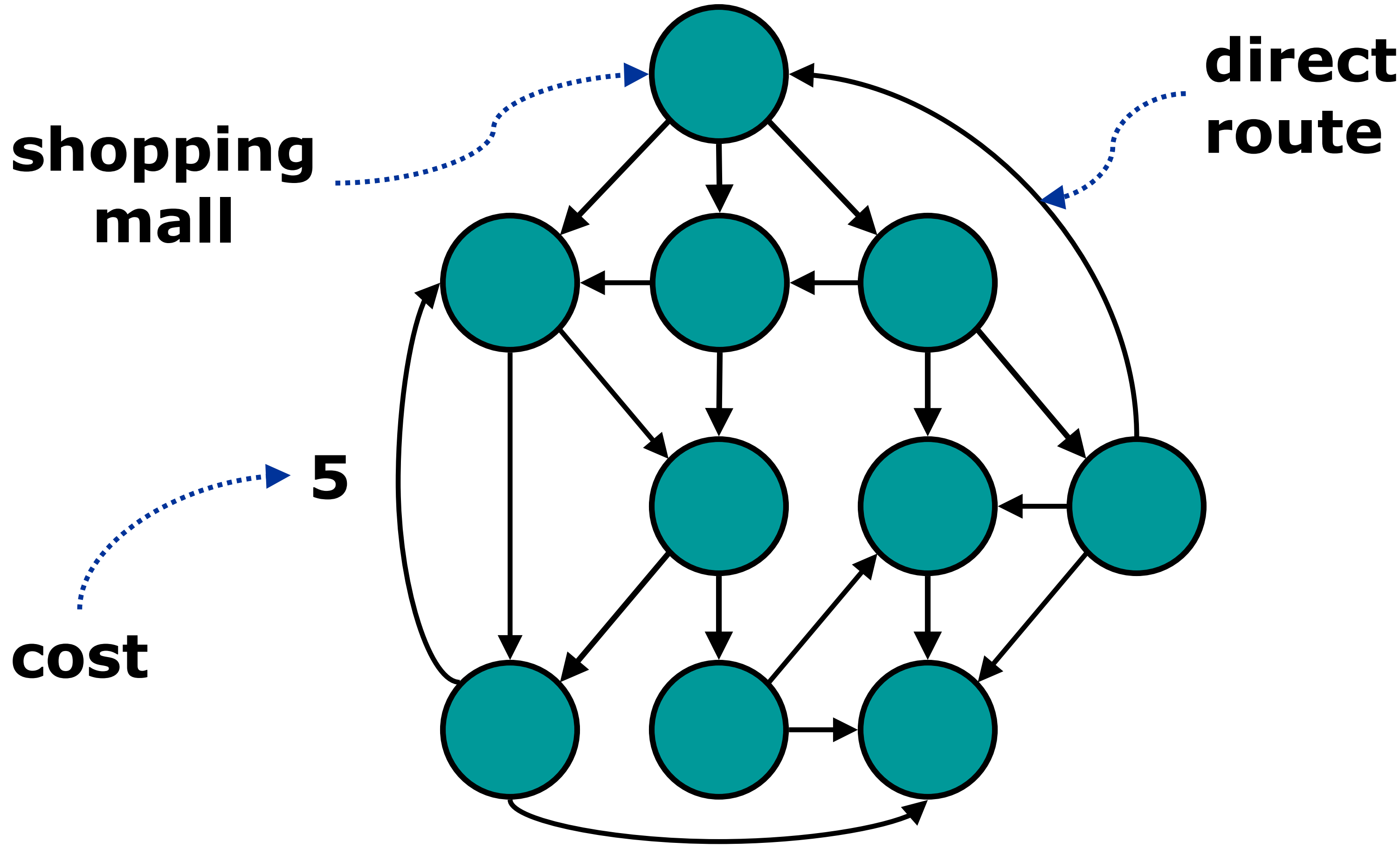
Bipartite Graph

- A **bipartite** graph, also called a *bigraph*, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.
- A bipartite graph is a special case of a k-partite graph with k=2.



Applications

Travel Planning



Questions

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

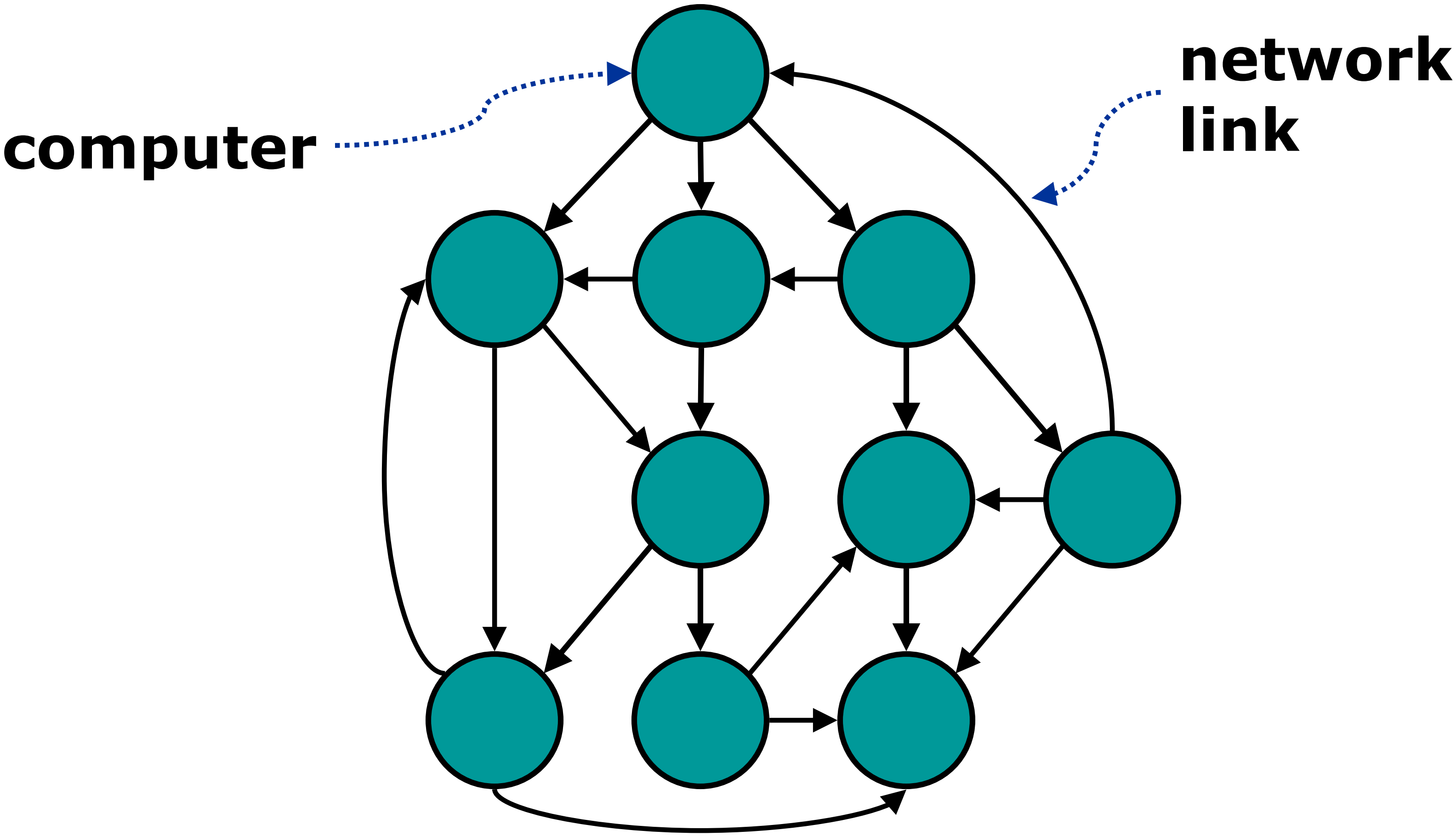
  "SHORTEST PATH PROBLEM"

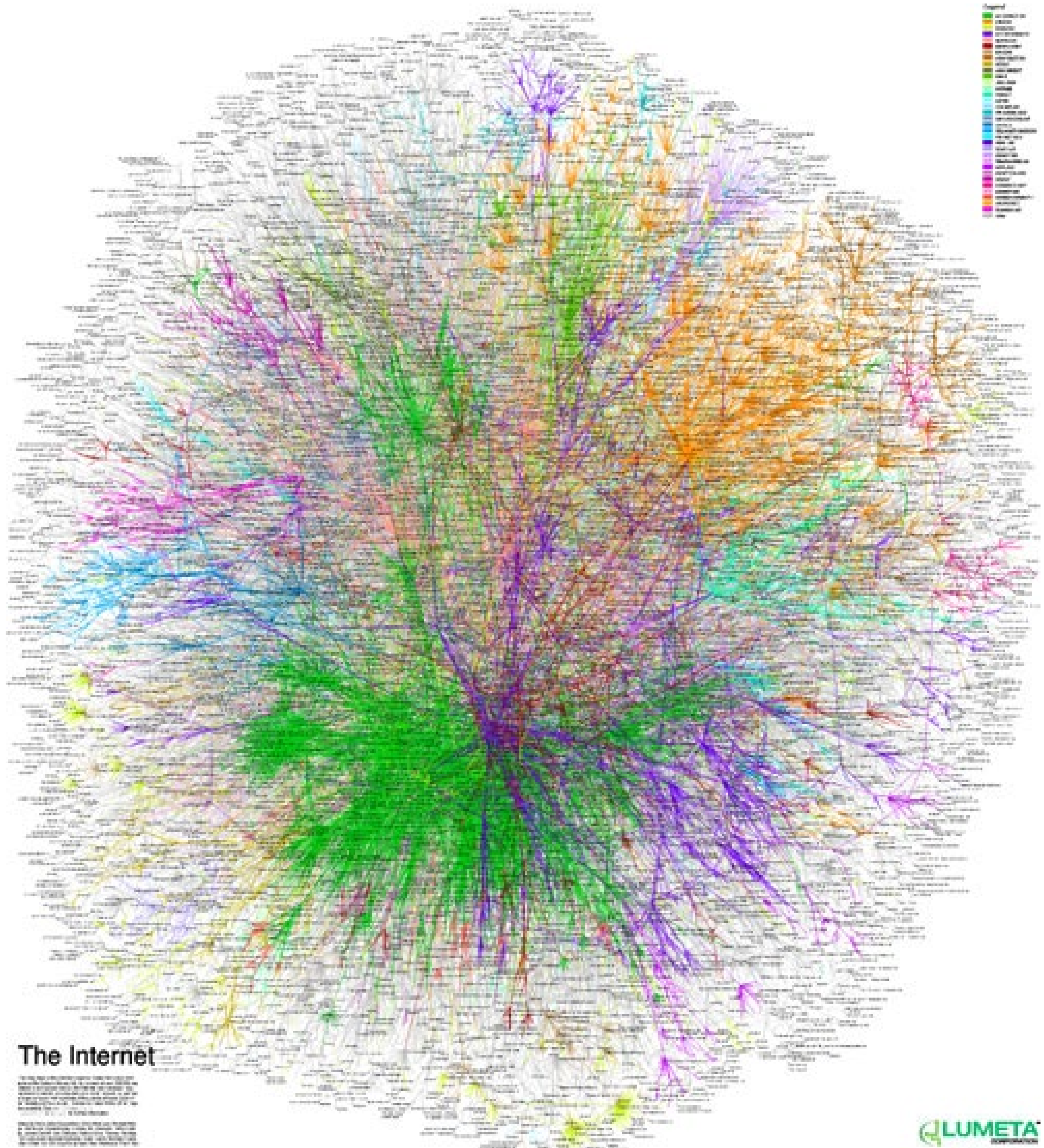
- How to minimize the cost of visiting n cities such that we visit each city exactly once, and finishing at the city where we start from?

"TRAVELING SALESMAN PROBLEM (TSP)"

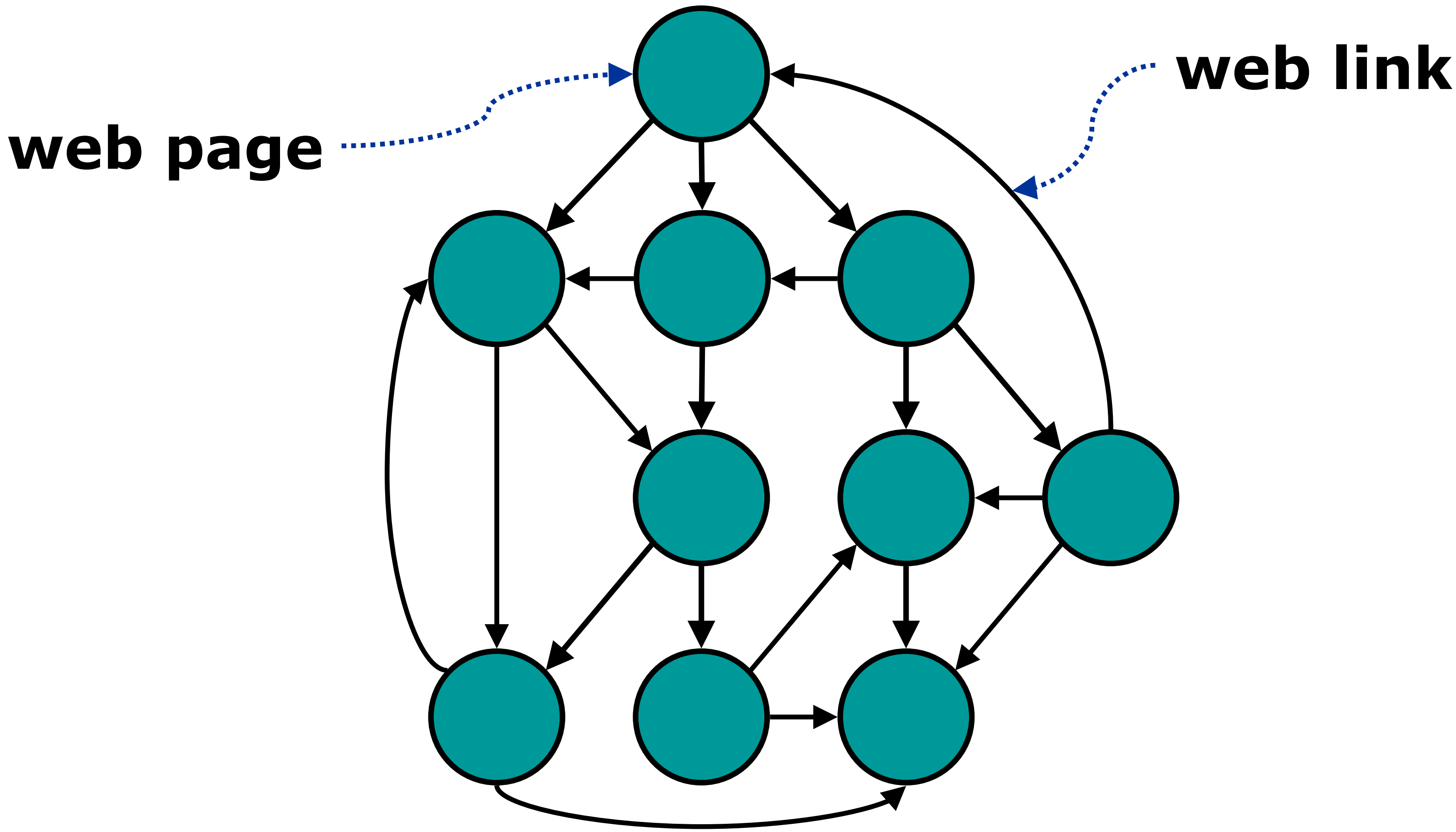
Internet

What is the shortest route to send a packet from A to B? (Shortest Path Problem)



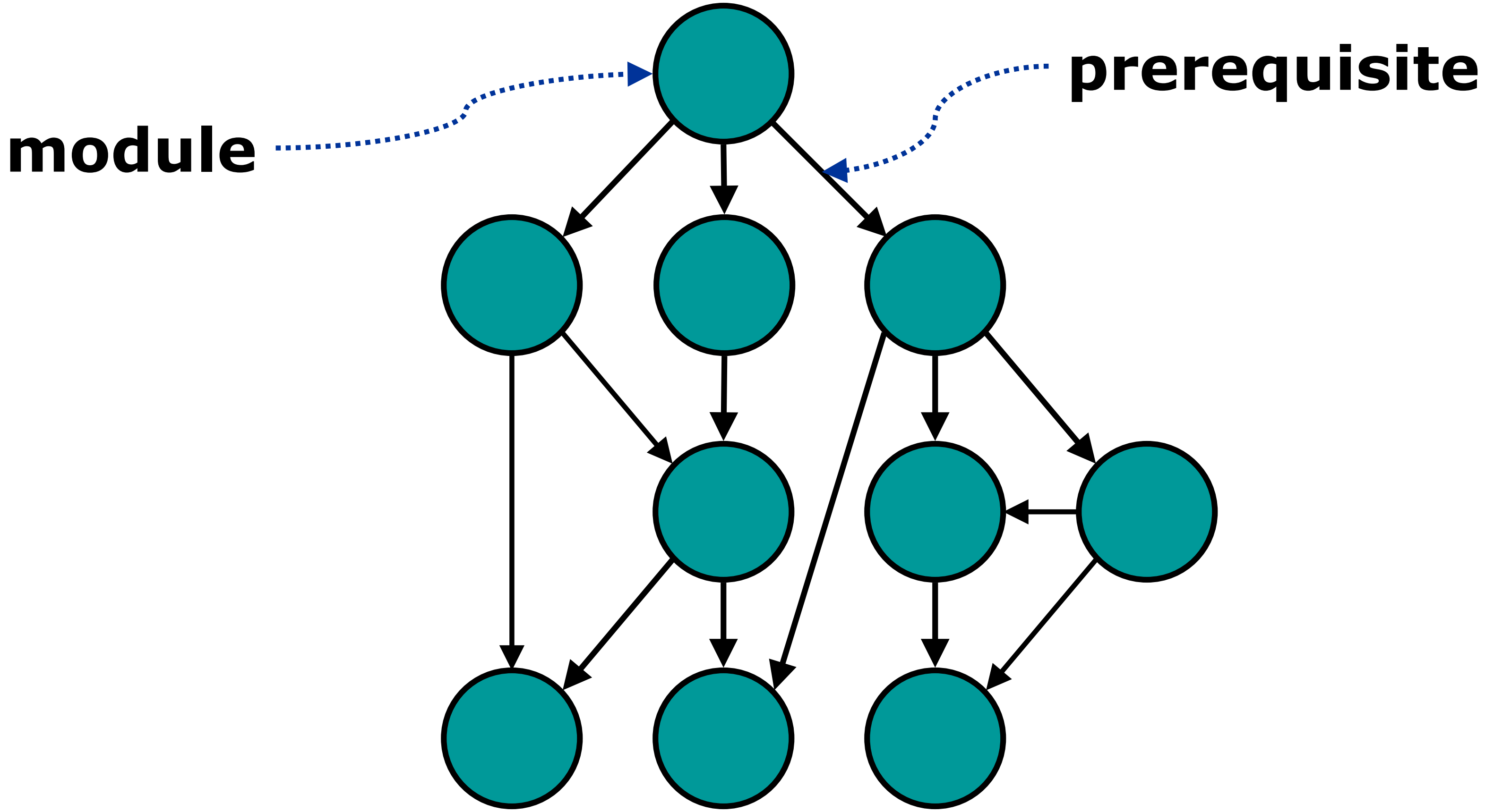

The Internet
LUMETA

The Web



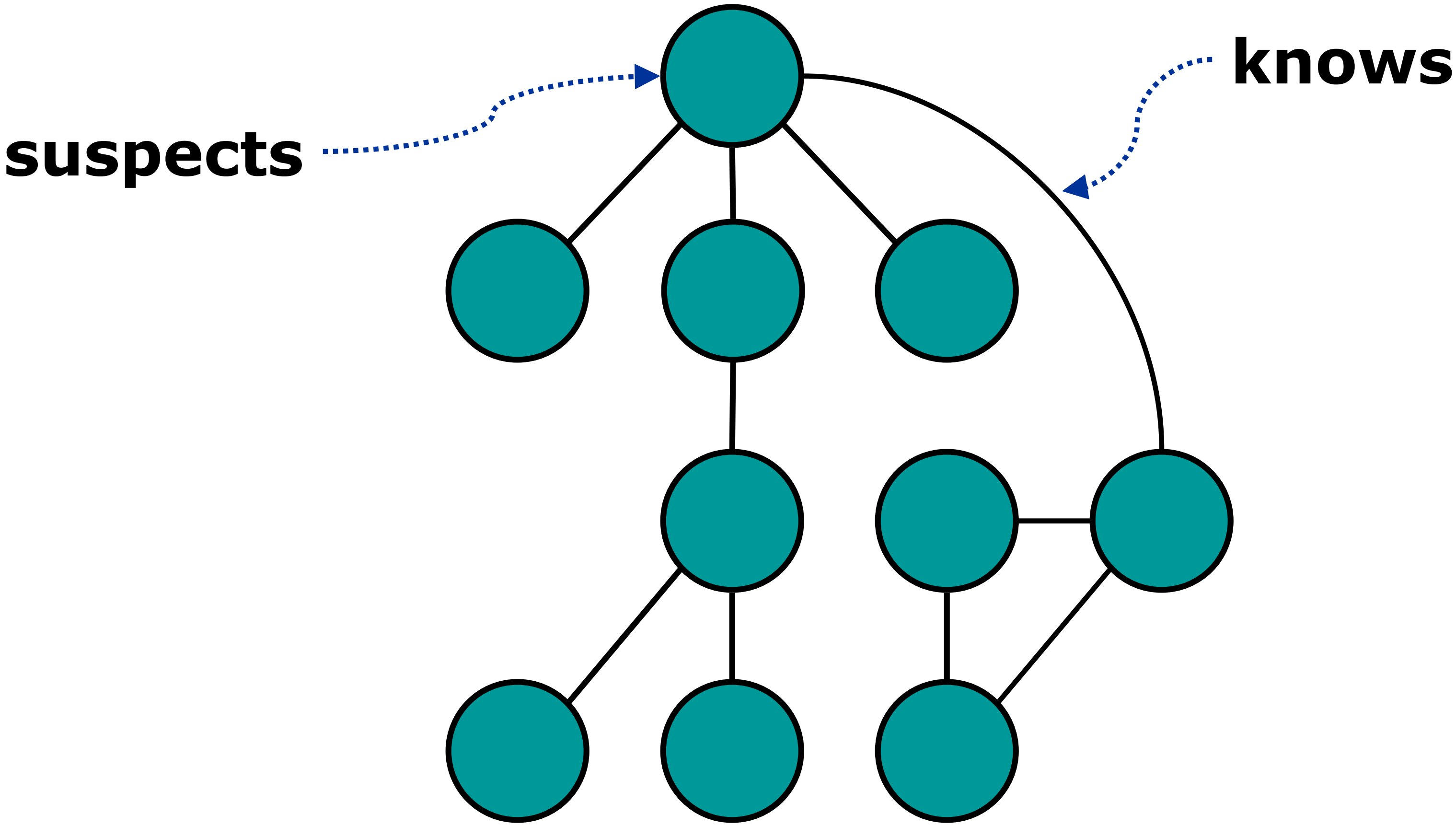
- Which web pages are important?
- Which set of web pages is likely to be of the same topic?

Module Selection

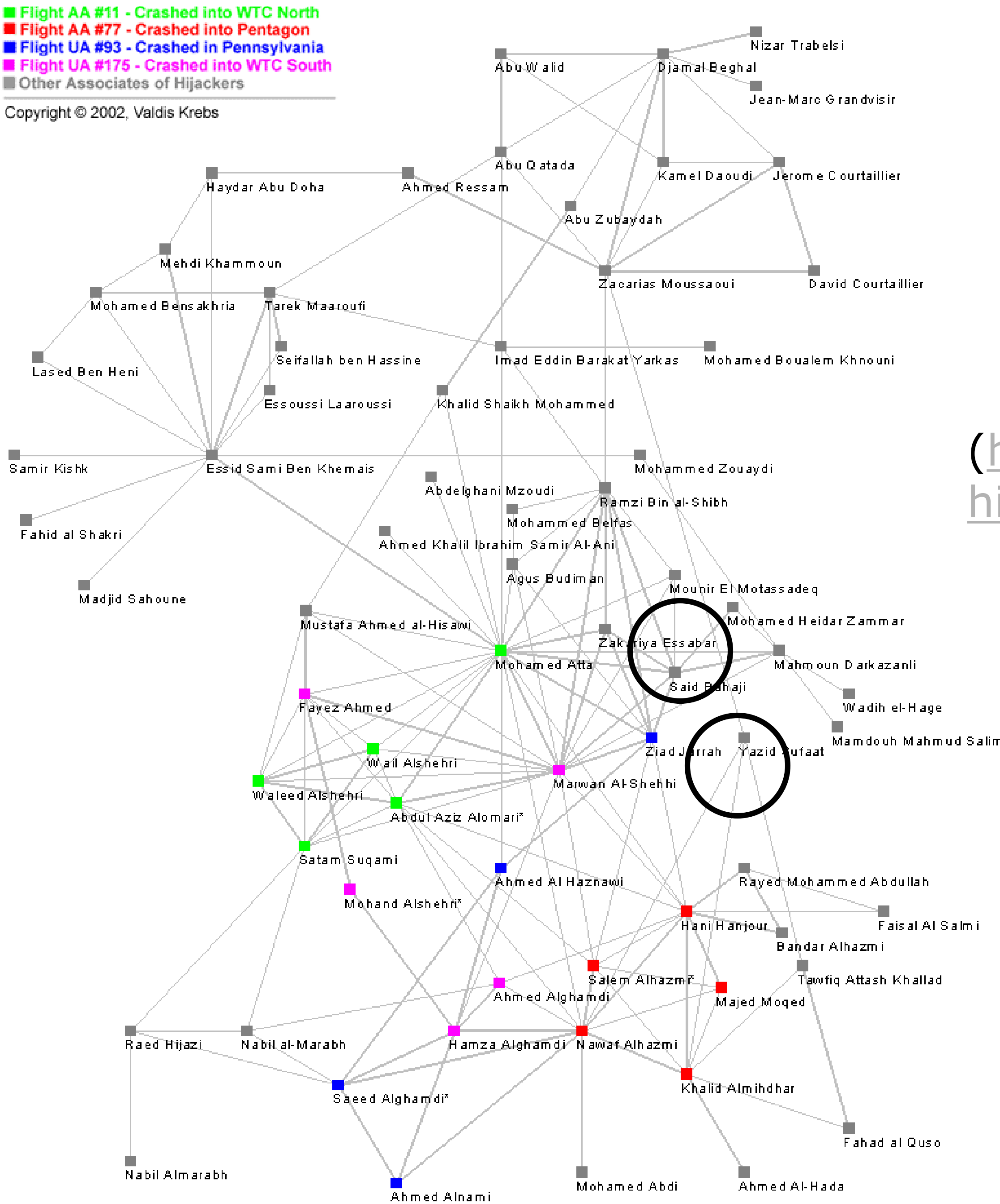


Find a sequence of modules to take that satisfy the prerequisite requirements (Topological sort)

Terrorist

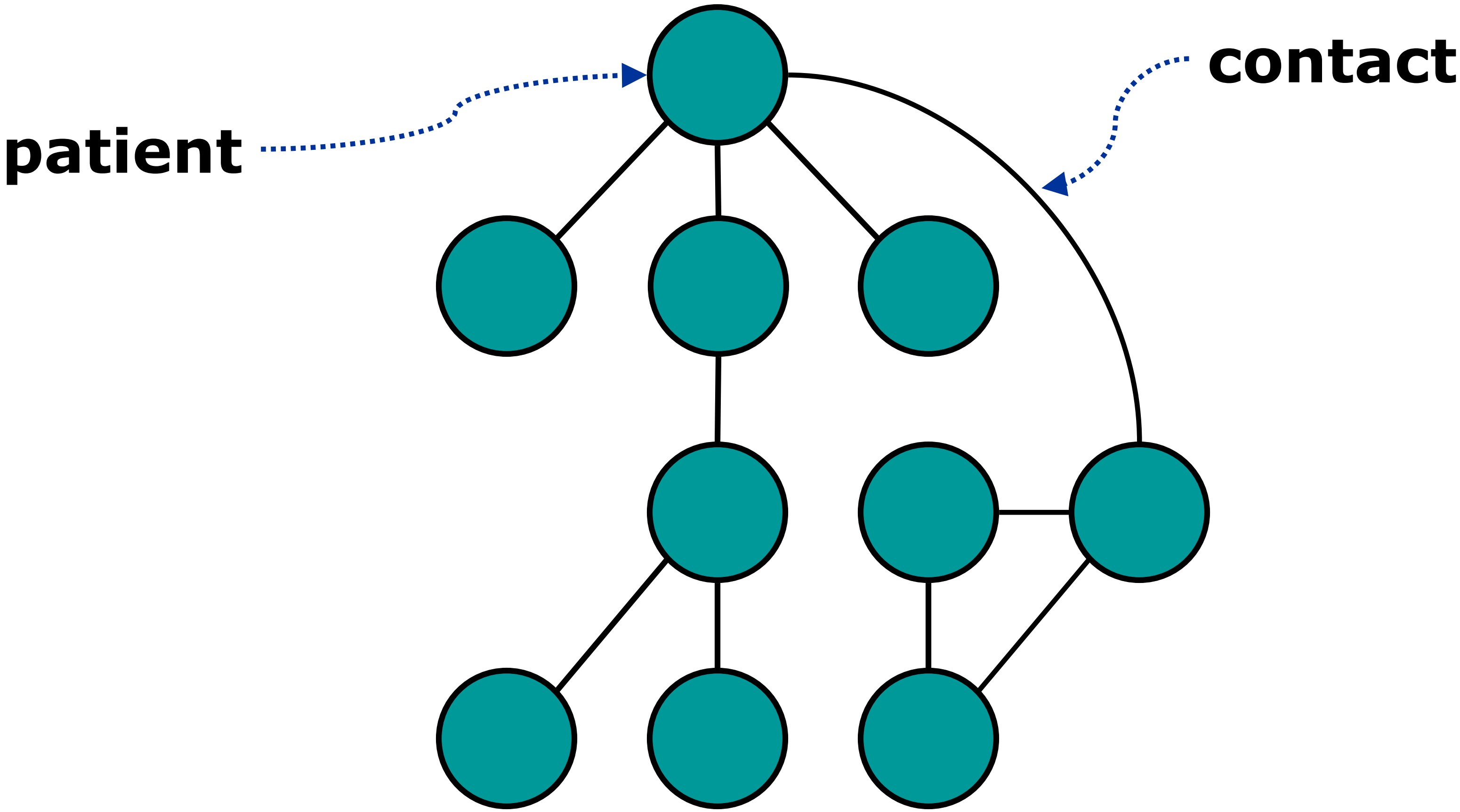


Who are the important figures in a terrorist network?



(http://www.orgnet.com/hijackers.html)

Epidemic Studies



Other applications

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

Implementation

Three implementations: Adjacency Matrix, Adjacency List and Edge List

Formally

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

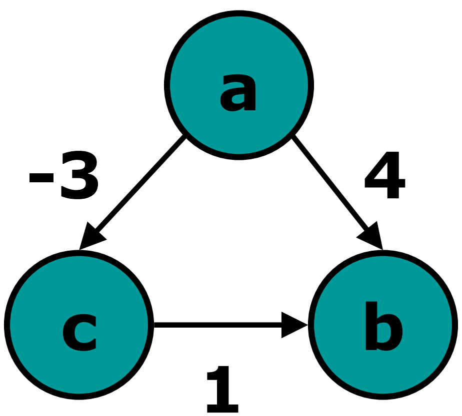
- V is the set of vertices
- E is the set of edges
- w is the weight function

Example

V = { a, b, c }

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

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



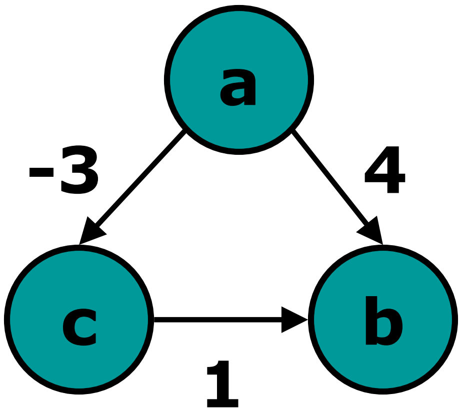
Adjacent vertices

- **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



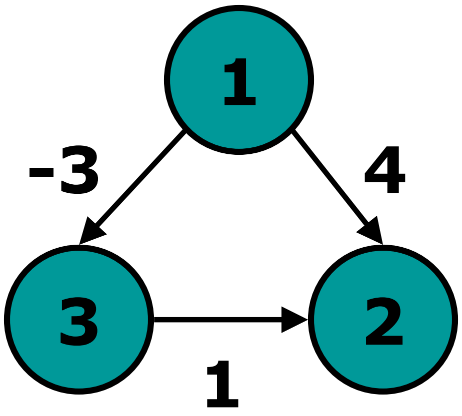
The vertices adjacent to v are called neighbours or successors of v.

Adjacency Matrix

Use 2-dimensional square matrix (array)

double AM[ ][ ];

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

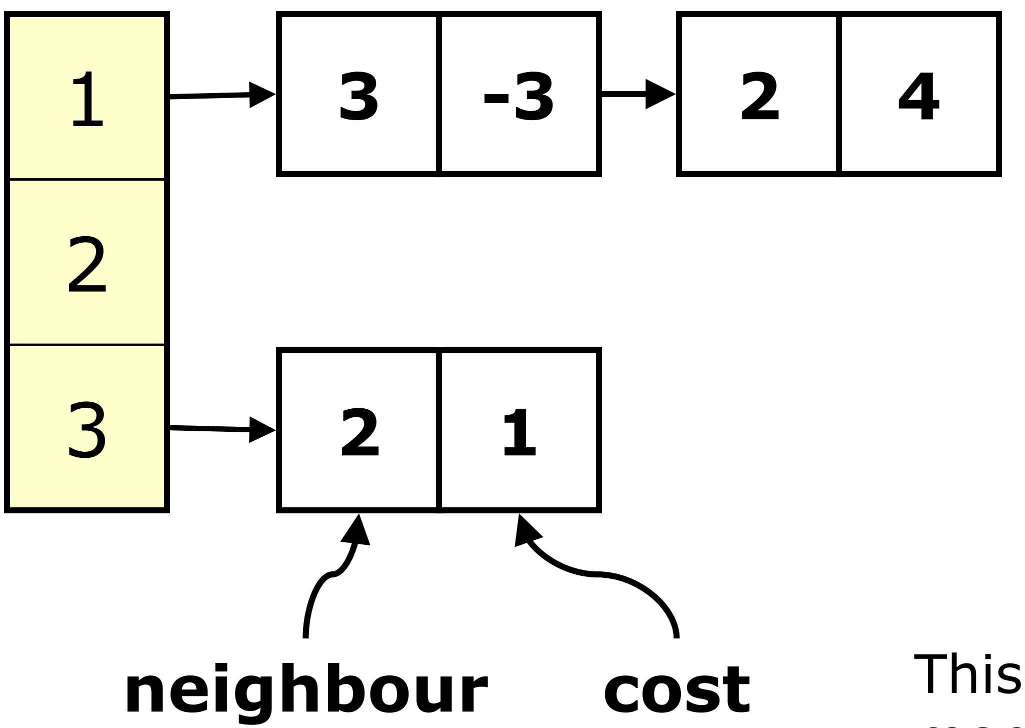


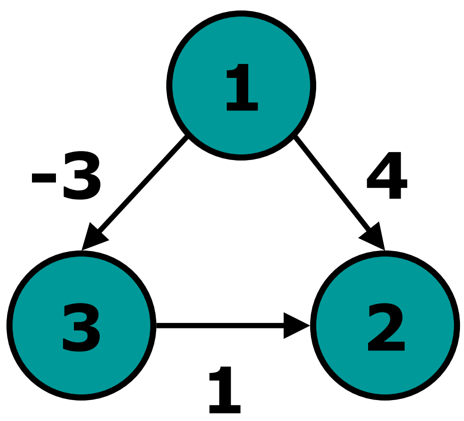
This requires $O(V^2)$ memory, and is not suitable for sparse graph. (Only 1/3 of the above matrix contains useful information).

Adjacency List

Array of Vertices

VertexList AL[ ]; // AL[i] stores list of i's neighbours





This requires only O(V + E) memory.

Adjacency List

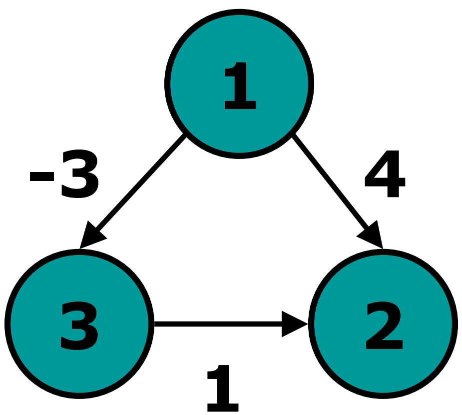
- In C++, can implement as vector of vector pairs

  `vector<vector<pair<int,int>>> AL;`
- use pairs as we need to store pairs of information for each edge: (neighbour vertex number, edge weight) where weight can be set to 0 or unused for unweighted graph.
- use Vector of Pairs due to Vector's auto-resize feature. If we have **k** neighbours of a vertex, we just add **k** times to an initially empty Vector of Pairs of this vertex (this Vector can be replaced with Linked List).
- We use Vector of Vectors of Pairs for Vector's indexing feature, e.g. if we want to enumerate neighbours of vertex **u**, we use AL[u] to access the correct Vector of Pairs.

Edge List

- Collection of edges with both connecting vertices and their weights
- In C++, can use vector of triples

  Vector<tuple<int, int, int>> EdgeList;
- Usually sorted by weight
- Example below shows sorted by 1st vertex, followed by 2nd vertex



vertex	vertex	weight
1	2	4
1	3	-3
3	2	1

Simple Applications

- Counting no. of vertices (V)
- Counting no. of edges (E)
- Enumerating neighbours of a vertex u
- Checking the existence of edge (u,v)
- etc

Counting V

- In an AM or AL, V is just the no. of rows in the array/vector
- Can be obtained in O(V)
- If graph is more or less static, use a variable to store this count, then O(1)
- Question: what if it was stored as EL?

Counting E

- In an EL, count no. of rows, O(E)
- AL: sum up the length of all V lists and divide final answer by 2 (for undirected graph), O(V+E)
- Again, can be stored as separate variable if graph is not dynamic
- Question: what if it was stored in AM?

Enumerating Neighbours of a Vertex u

- In AM, need to loop through all columns of AM[u][j] for every j and report pair of (j, AM[u][j]) if AM[u][j] is not zero, O(V)
- In AL, need to scan AL[u]. If there are only k neighbours of u, then just need O(k) to enumerate them
- Question: what if it was stored in EL?

Checking Existence of Edge (u,v)

- AM: simply check if AM[u][v] is non-zero, O(1)
- AL: have to check whether AL[u] contains vertex v or not, O(k)
- Question: what if it was stored as EL?

Summary of diff implementations

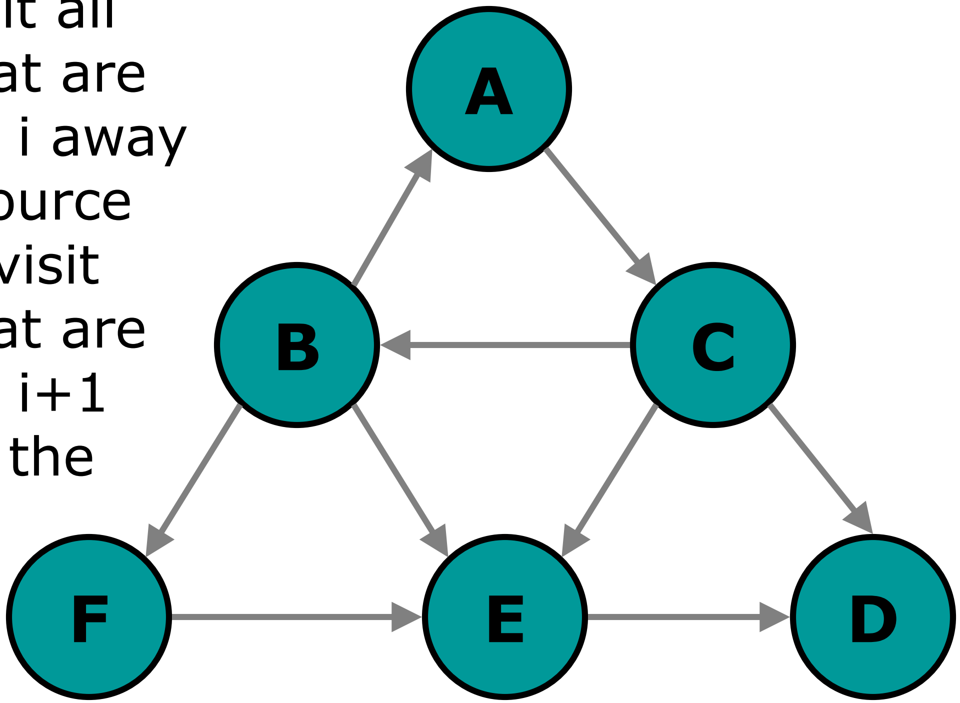
	Adjacency Matrix	Adjacency List	Edge List
Implementation	2-D array	Vector of Vector pairs	Vector of triples
Space Complexity	$O(V^2)$	$O(V+E)$	$O(E)$
Counting V	$O(V)$	$O(V)$	$O(E)$
Counting E	$O(V^2)$	$O(V+E)$	$O(E)$
Enumerating neighbours of u	$O(V)$	$O(k)$ (k neighbours)	$O(E)$
Checking existence of edge (u,v)	$O(1)$	$O(k)$	$O(E)$

Breadth-First Search (BFS)

Traversing a Graph

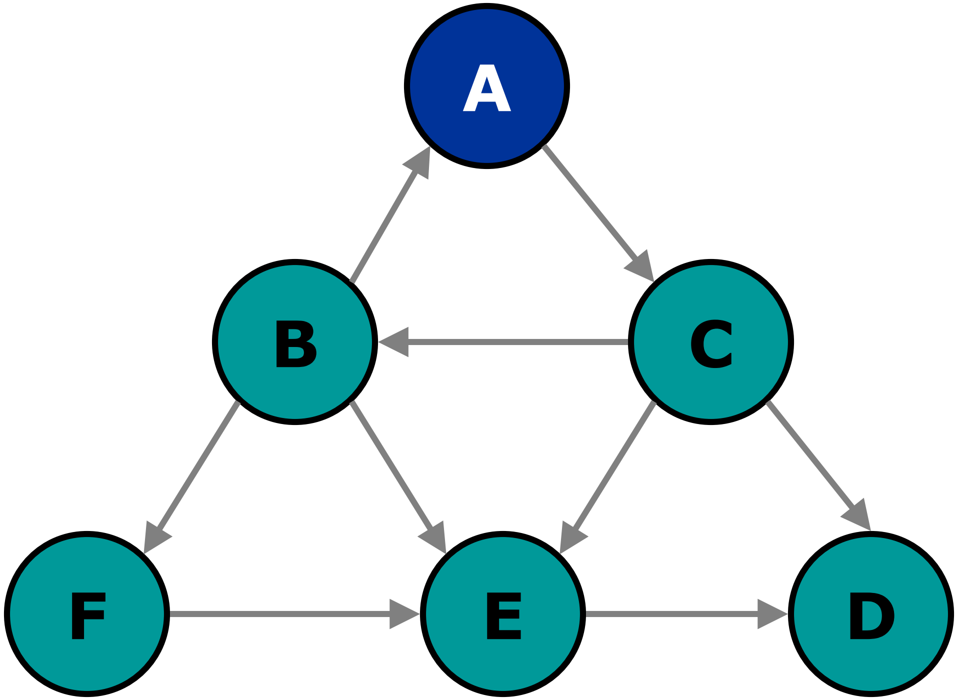
Breadth-first search

- Given a source vertex, we would 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

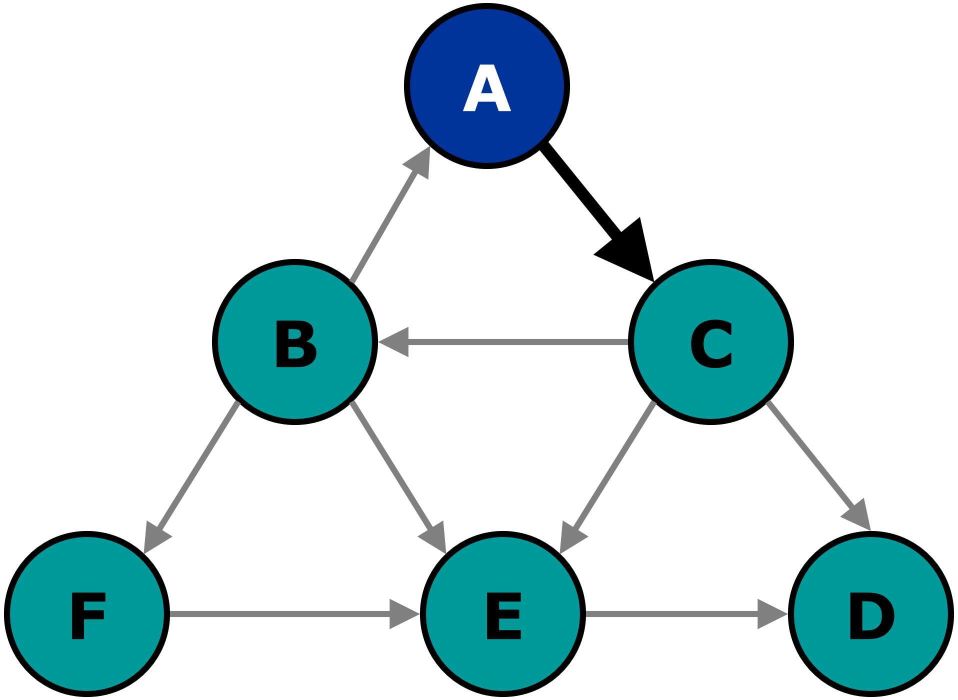


- The order of search is not unique and depends on the order of neighbours visited

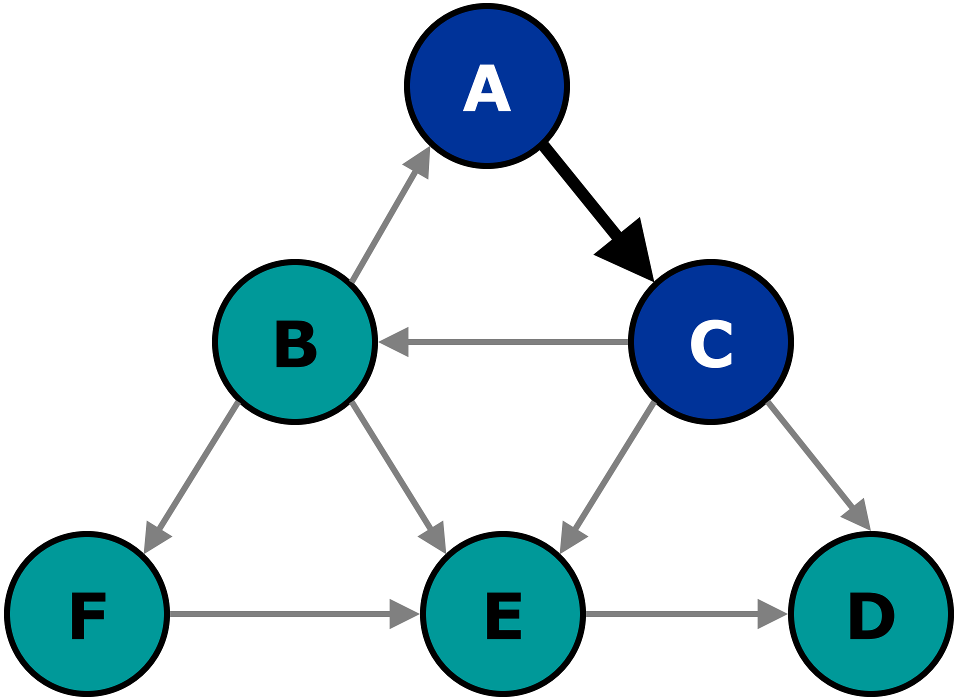
Breadth-first search



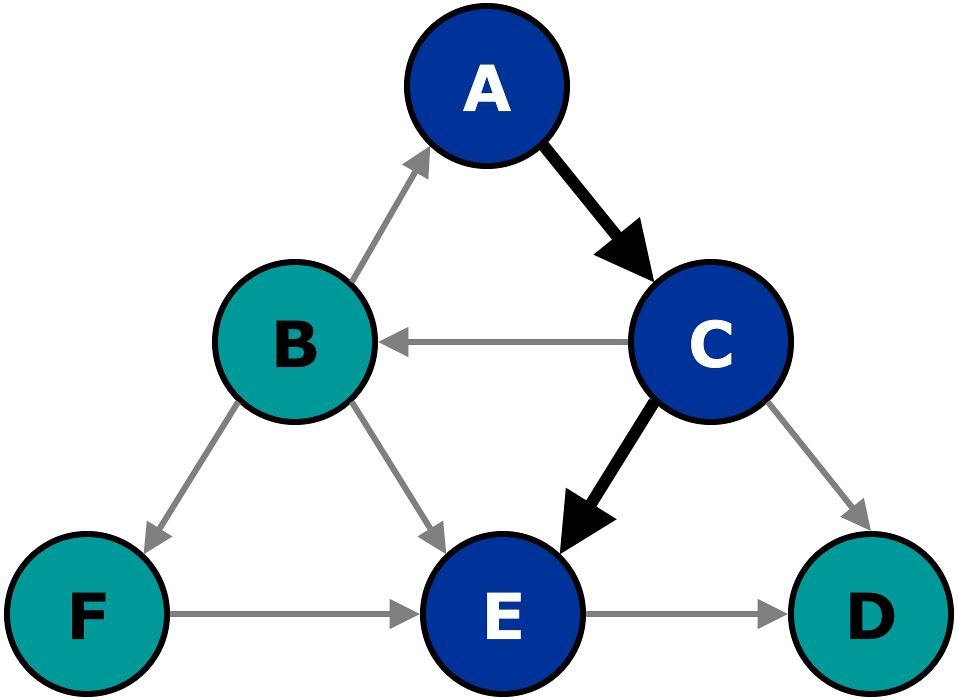
Breadth-first search



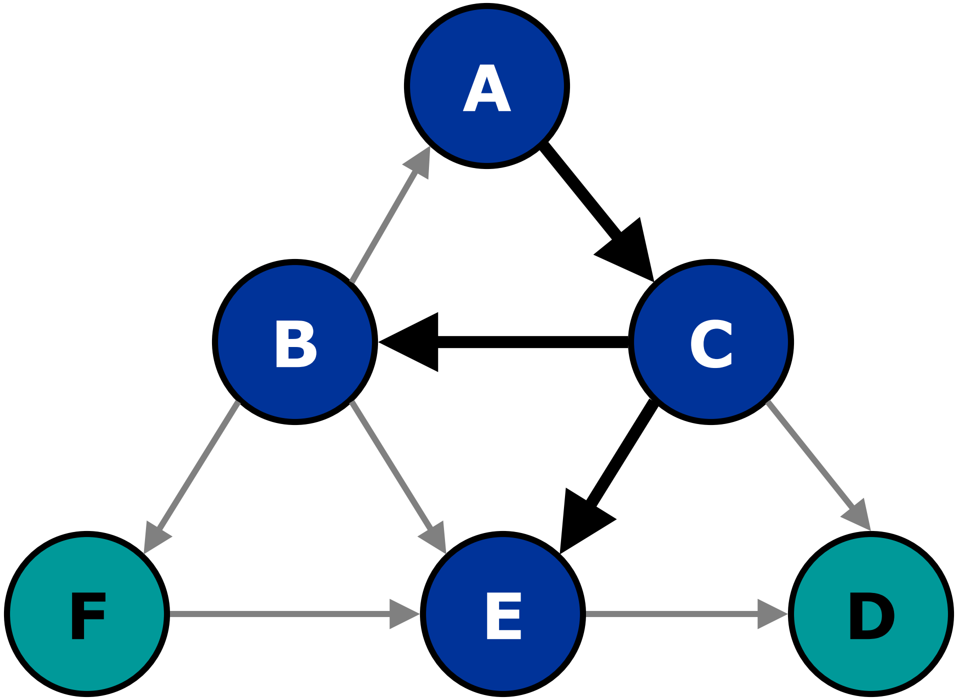
Breadth-first search



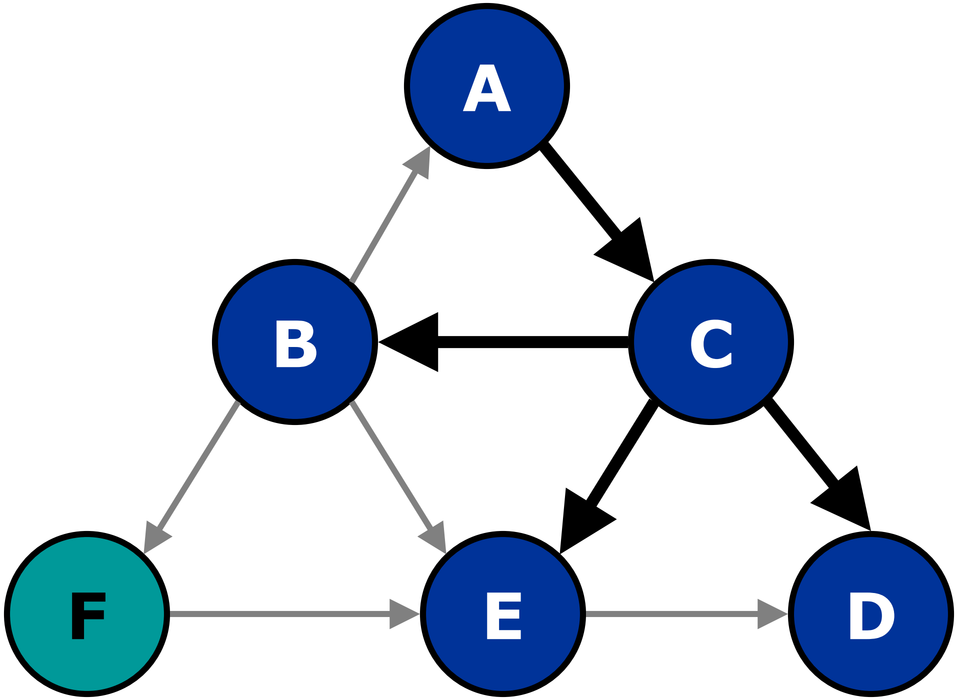
Breadth-first search



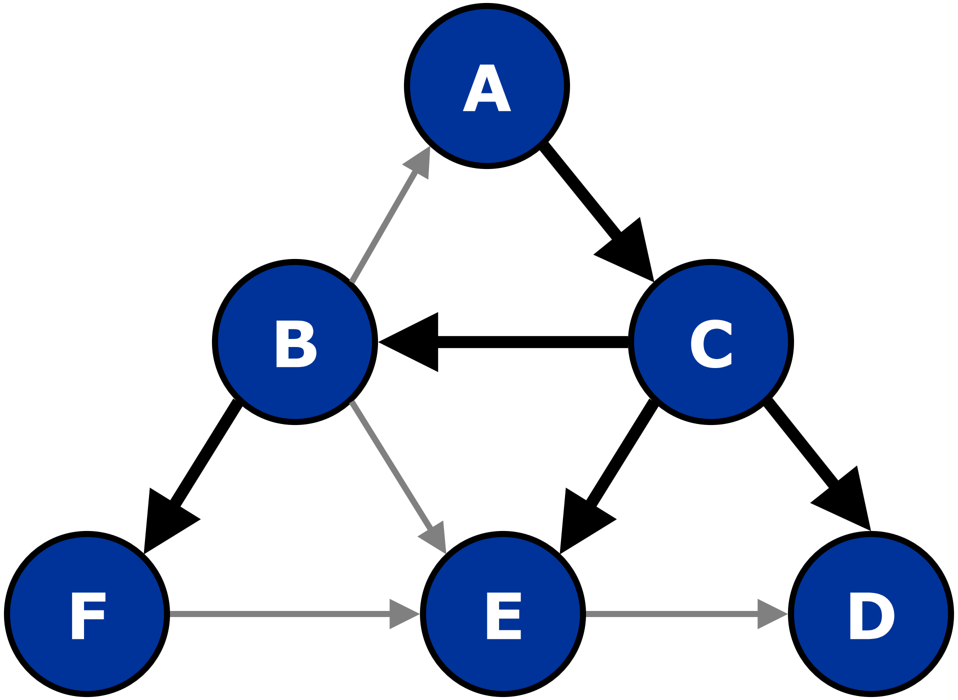
Breadth-first search



Breadth-first search

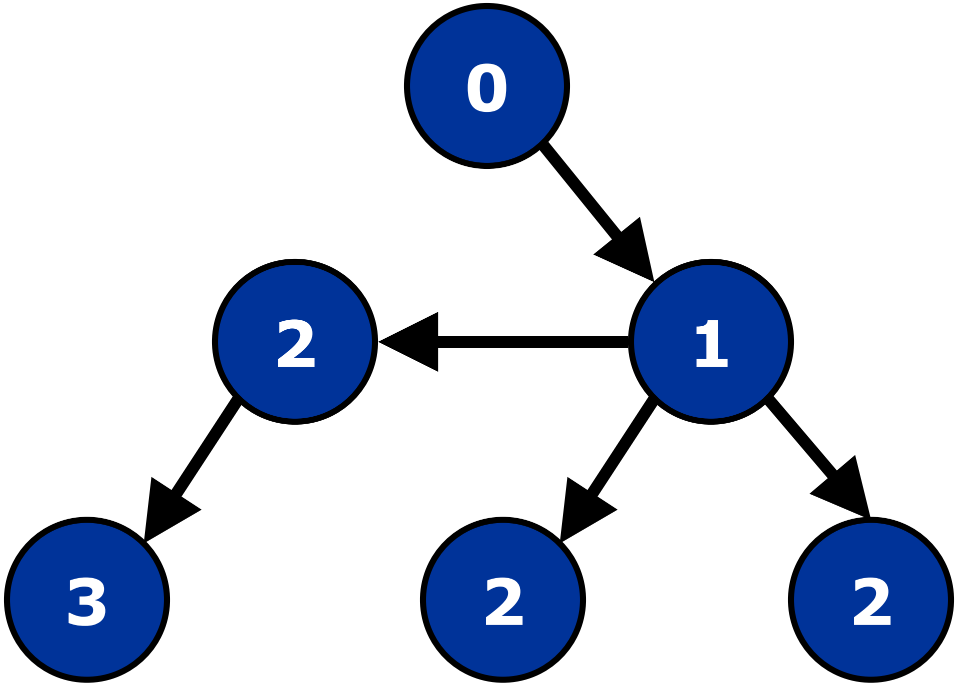


Breadth-first search



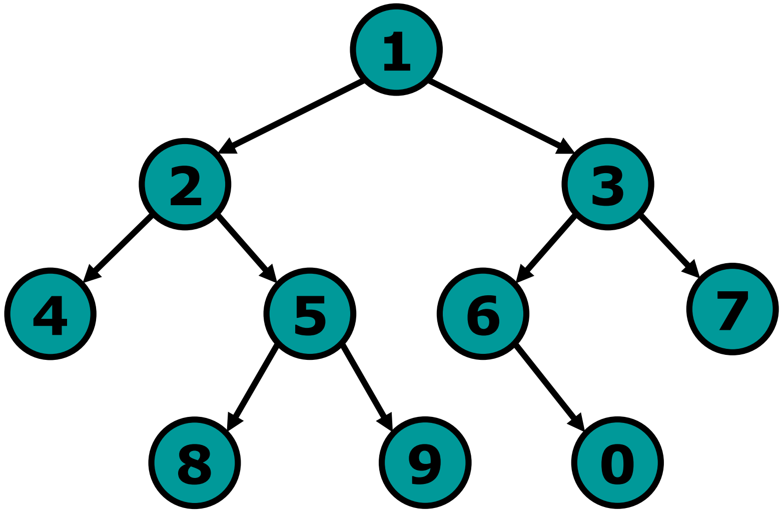
Breadth-first search

- 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 labelled with their distance from the source (or *level*).



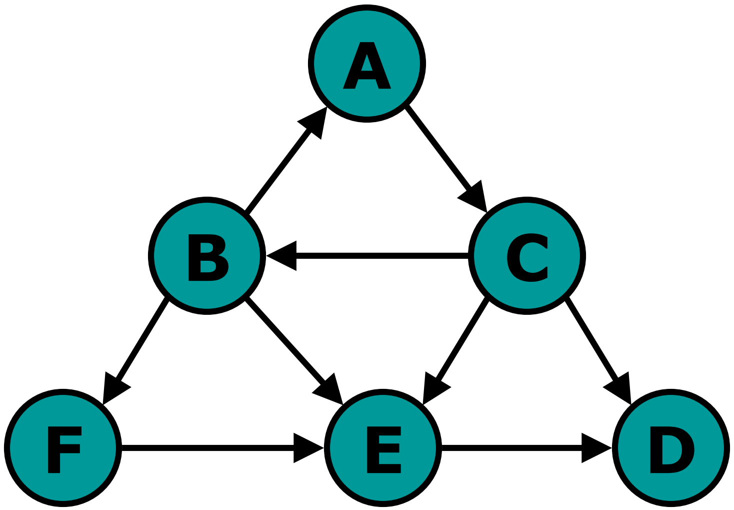
Recall: Level-Order on Tree

```
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)
```



BFS(v)

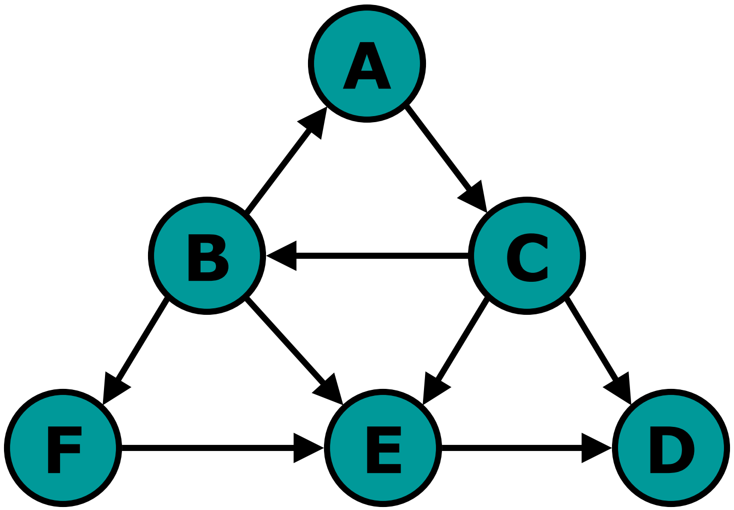
```
Q = new Queue
Q.enq (v)
mark v as visited
while Q is not empty
    curr = Q.deq()
    print curr
    foreach w in adj(curr)
        if w is not visited
            Q.enq(w)
            mark w as visited
```



The pseudocode for BFS is very similar to level-order traversal of trees. The major difference is that now we may visit a vertex twice (since unlike a tree, there may be more than one path between two vertices). Therefore, we need to remember which vertex we have visited before (how?)

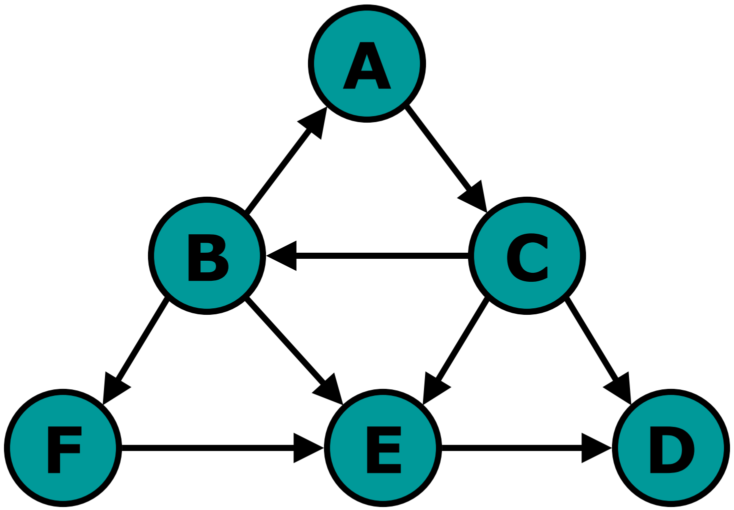
Building the BFS Tree

```
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
            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
```



Similarly, we can maintain the distance of a vertex from the source.

Search all vertices

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

BFS guarantees that if there is a path to a vertex v from the source, we can always visit v. But since some vertices may be unreachable from the source, we can call BFS multiple times from multiple source.

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
```

Main Loop

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

Initialization

$$O(V)$$

Total Running Time

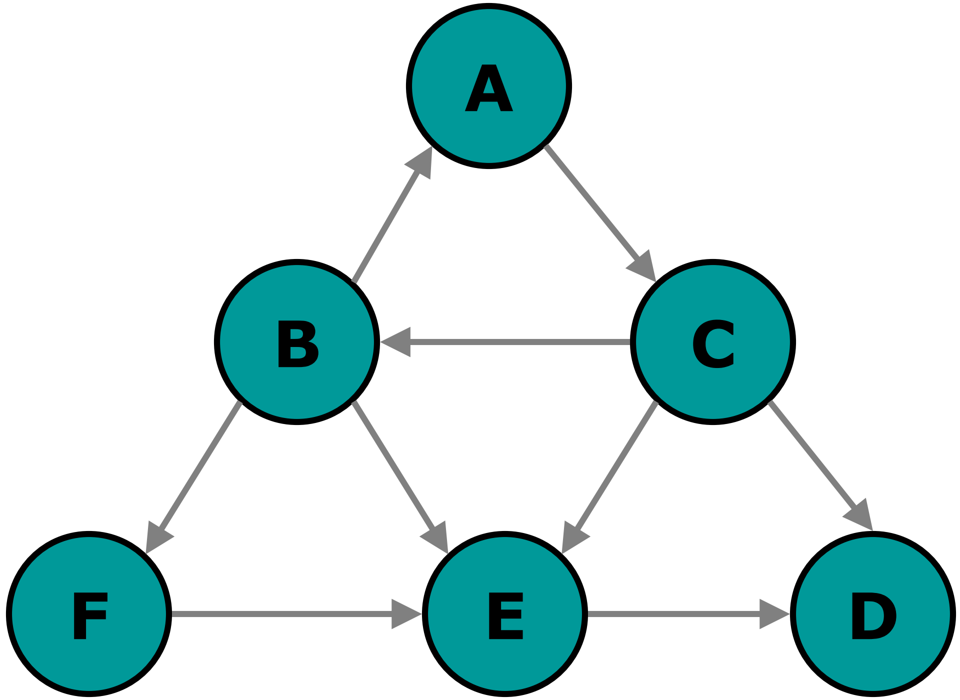
$$O(V + E)$$

Depth-First Search

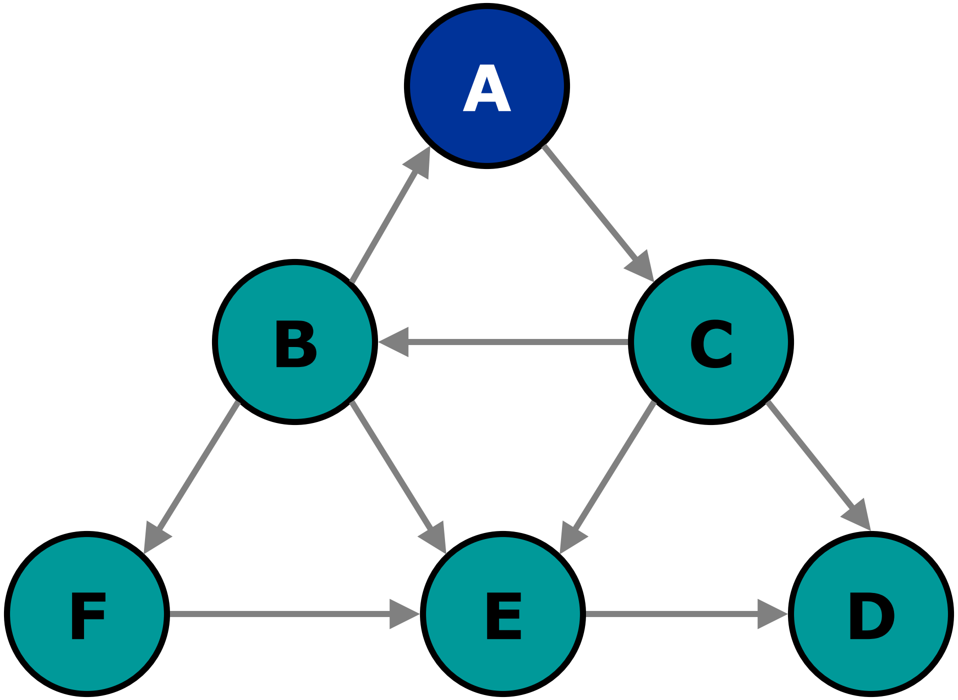
Traversing a Graph

Depth-first search

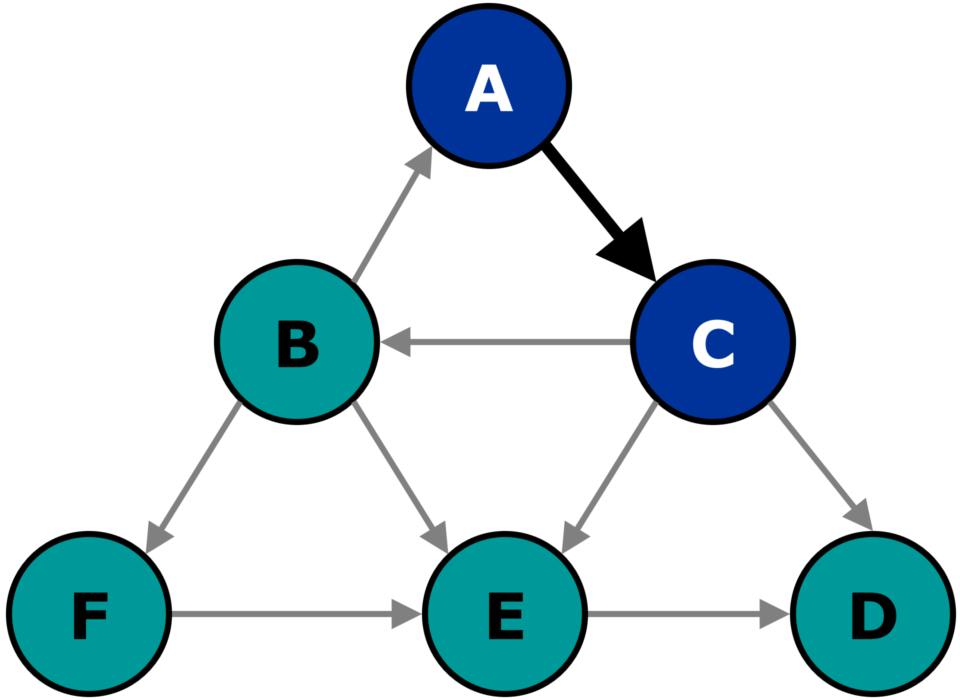
Idea for DFS is to go as deep as possible. Whenever there is an outgoing edge, we follow it.



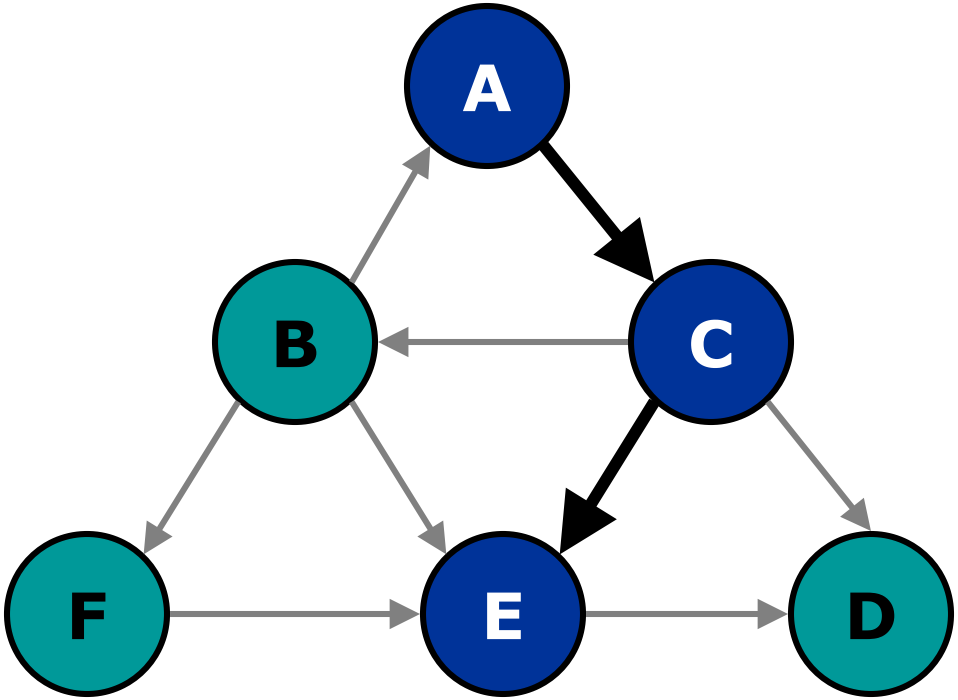
Depth-first search



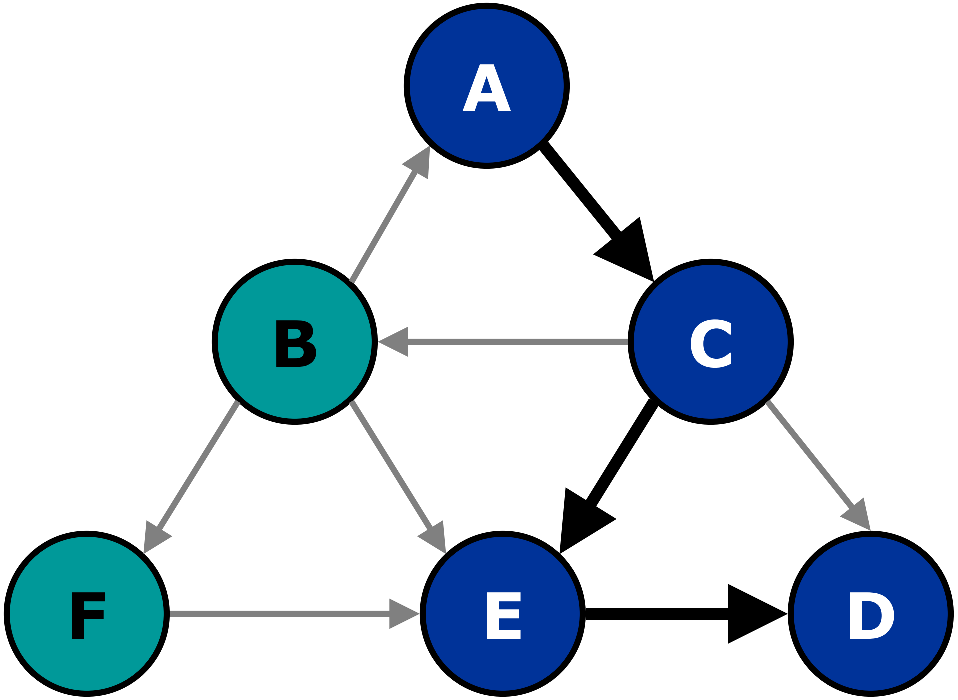
Depth-first search



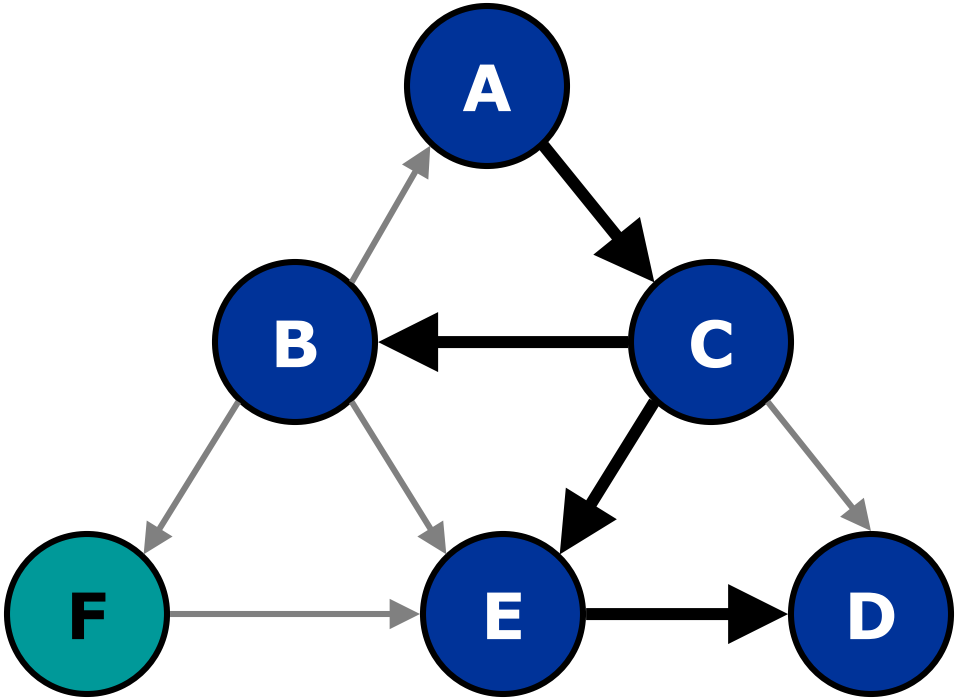
Depth-first search



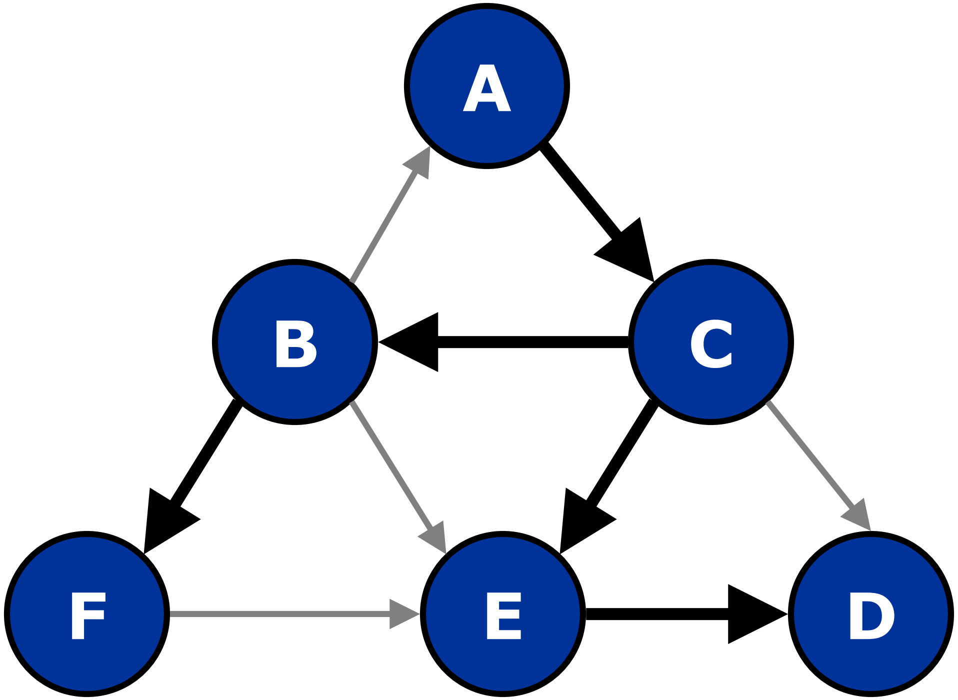
Depth-first search



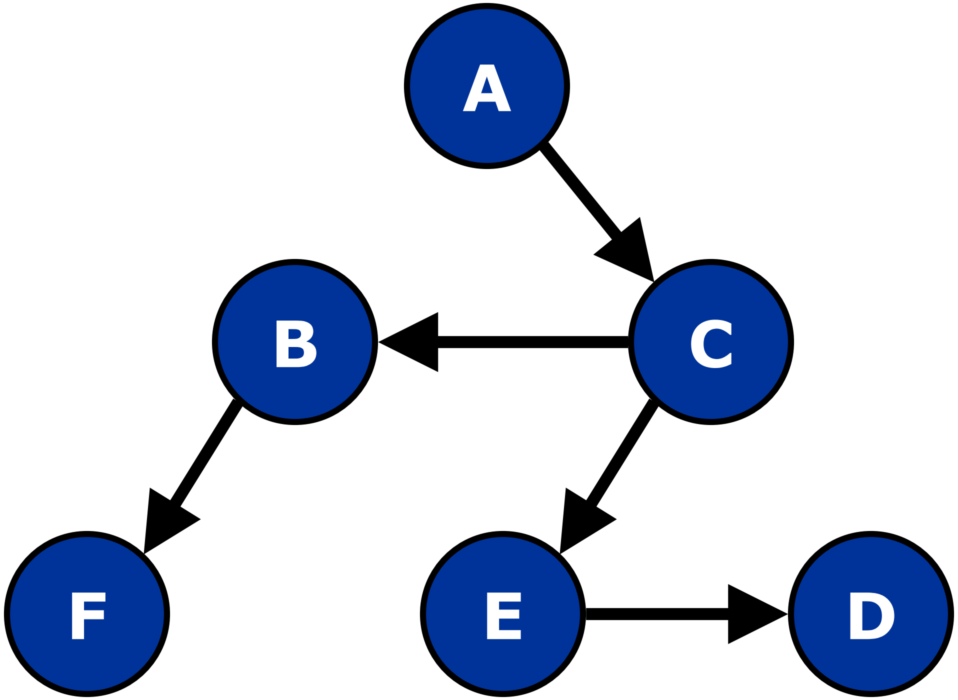
Depth-first search



Depth-first search

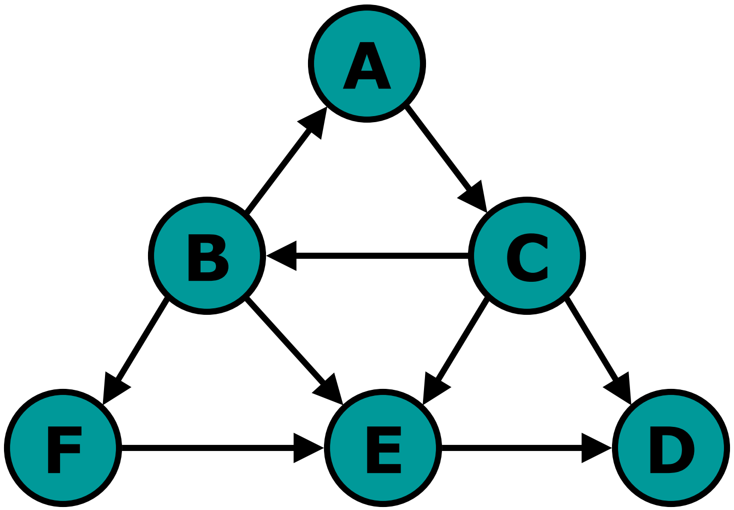


Depth-first search



DFS(v)

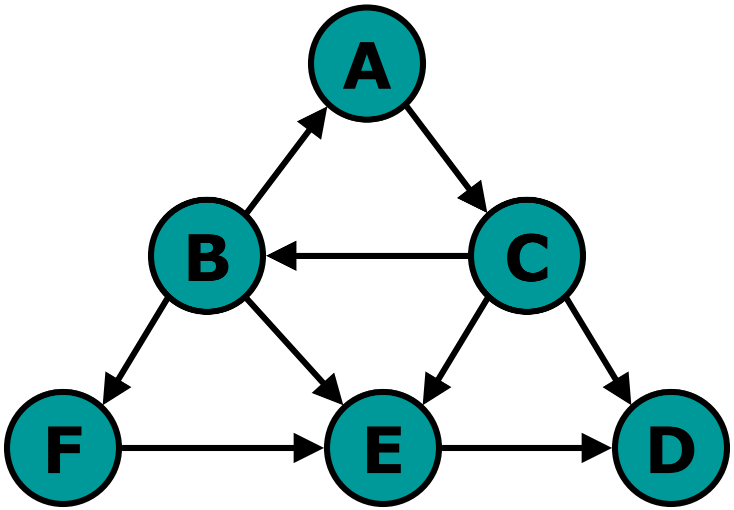
```
S = new Stack
S.push (v)
mark v as visited
while S is not empty
    curr = S.top()
    if every vertex in adj(curr) is visited
        S.pop()
    else
        let w be an unvisited vertex in adj(curr)
        S.push(w)
        print and mark w as visited
```



In DFS, we use a stack to “remember” where to backtrack to.

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

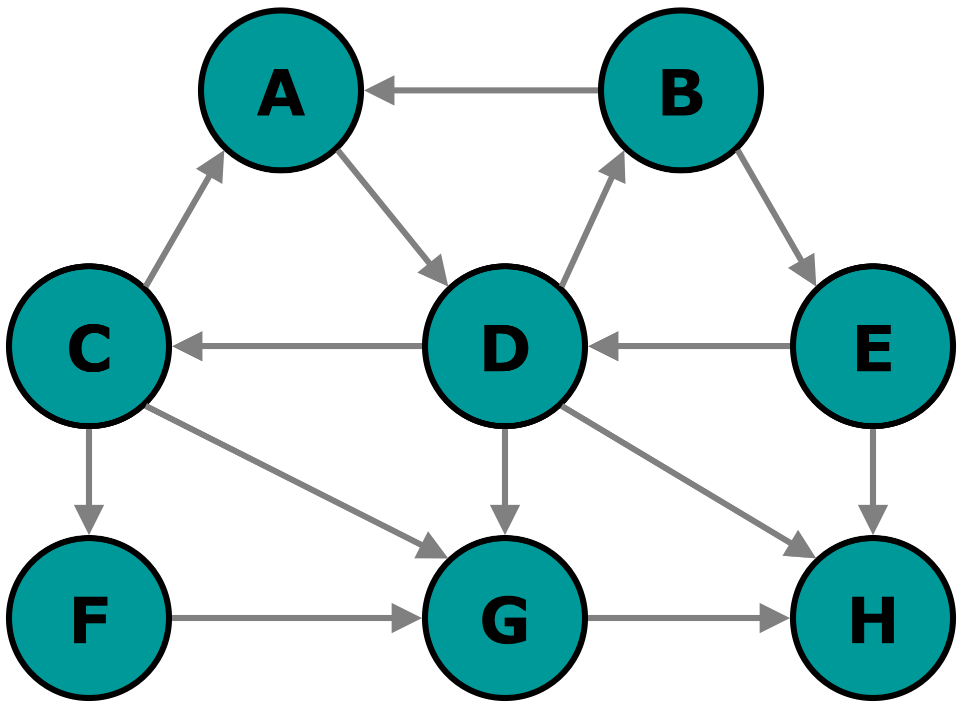
```
Search(G)
    foreach vertex v
        mark v as unvisited
    foreach vertex v
        if v is not visited
            DFS(v)
```

Just like BFS, we may want to call DFS() from multiple vertices to make sure that we visit every vertex in the graph.

Running time

- DFS: O(V + E)
- Each vertex is only visited once (DFS recursively explores vertices that are not visited (O(V))
- Every time a vertex is visited, all its k neighbours are explored. After all vertices are visited, we have examined all E edges (O(E), since total no. of neighbours of each vertex = E)

Exercise: trace the graph using BFS and DFS



BFS/DFS Applications

- Detecting if a graph is cyclic
- Printing the traversal path
- Reachability test
- Identifying/Counting connected components of undirected graphs
- Topological Sort (applicable to DAGs)
- *Please refer to visualgo for examples*

Detecting Cycles

- Augment DFS with additional data, include array status[u] with three enumerated values:
  - *Unvisited* (vertex u has not been reached before)
  - *Explored* (visited u before but at least one neighbour of u has not been visited yet)
  - *Visited*: (all neighbours of u visited, can backtrack)
- If DFS is traversing x -> y and status[y] is explored, then a cycle has been found, since we have visited y before (refer to vertices A, B, C in slide 55)

Printing Traversal Path

- Define array p[u] to remember parent/predecessor of vertex u along the BFS or DFS traversal path
- p[source] = -1   (source has no parent)

```
backtrack(u)
    if (u == -1) stop
    backtrack(p[u]);
    output vertex u
```

- To print path from source u to target t, call DFS or BFS and then call backtrack(t)

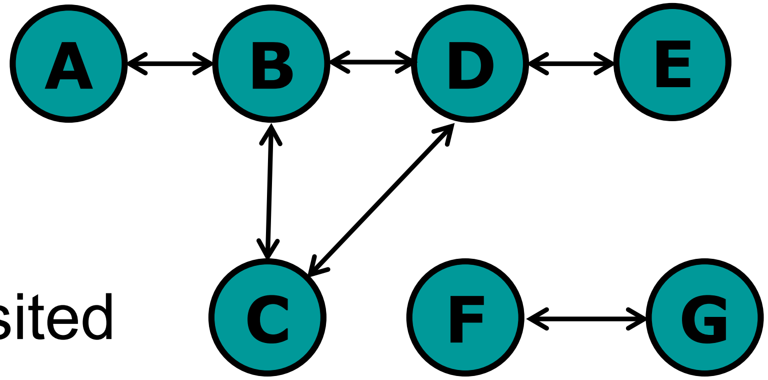
Reachability Test

- To test if vertex s and vertex t are reachable (directly connected or indirectly via a simple, non-cyclic path) call DFS/BFS and check if status[t] = visited

Identifying a Connected Component

- Enumerate all vertices that are reachable from vertex s in an undirected graph
- Call DFS(s)/BFS(s) and enumerate all vertices v that have status[v] = visited
- These vertices form a *Connected Component (CC)*

Counting No. of CCs



Pseudocode:

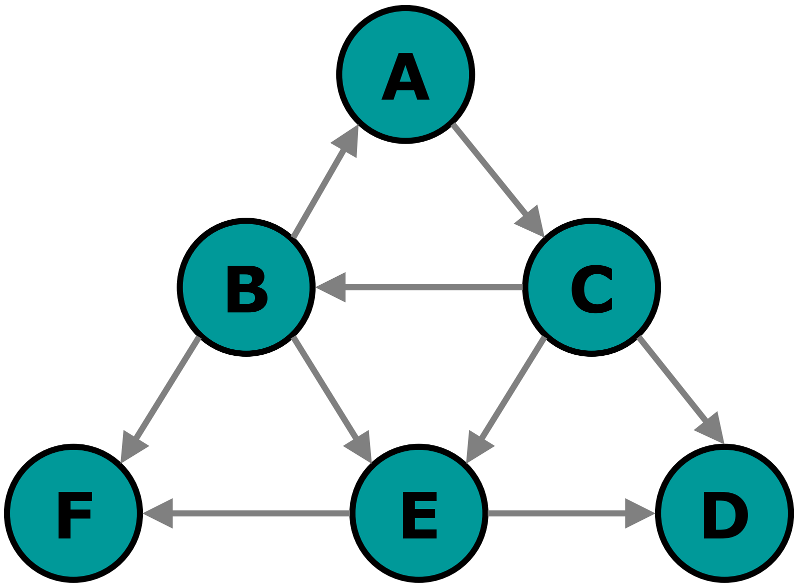
```
CC = 0
for all u in V, set status[u] = unvisited
for all u in V
    if (status[u] == unvisited)
        CC++ // we can use CC count number as the CC label
        DFS(u) // or BFS(u), that will flag its members as visited
output CC // the answer is 2 for the example graph above,
            // CC 0 = {A,B,C,D,E}, CC 1 = {F, G}
```

Definition

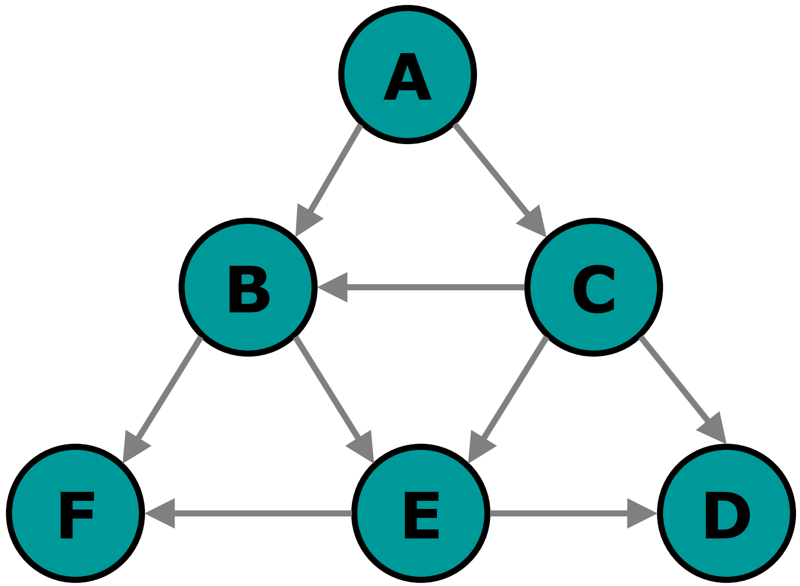
- An acyclic graph is a graph without a cycle
- An undirected graph is a tree
- in-degree of a vertex is the number of incoming edges
- out-degree of a vertex is the number of outgoing edges

Definition

- Directed Acyclic Graph (DAG): A directed graph with no cycle.

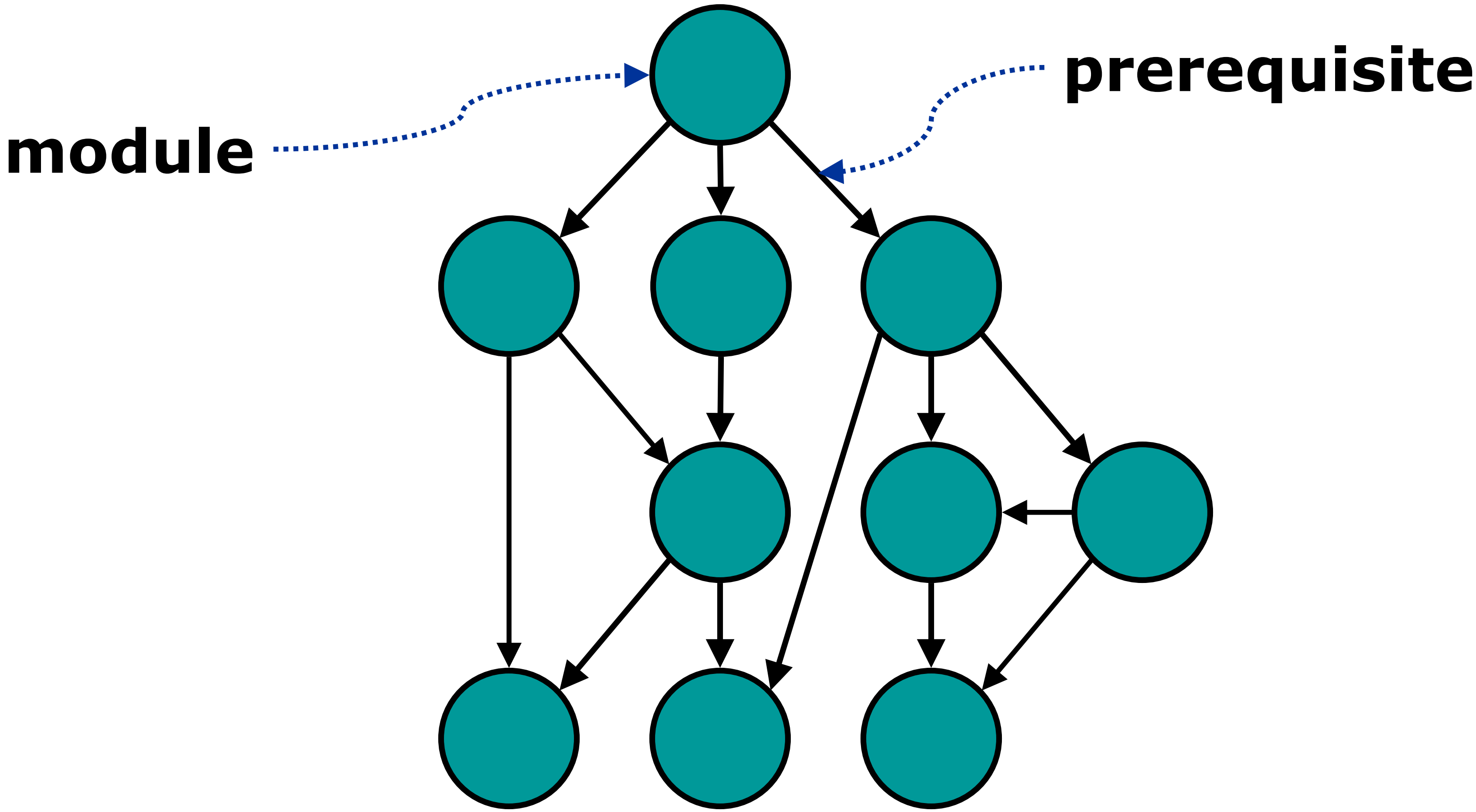


Not a DAG



DAG

Module selection

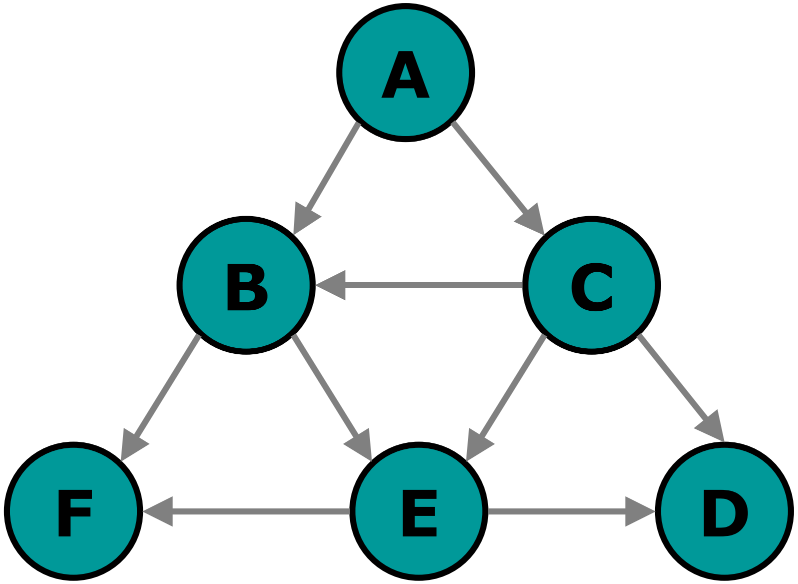


Topological Sort

- Goal: Give a DAG, order the vertices, such that if there is a path from u to v, u appears before v in the output.
- This is useful when vertices represents items with dependencies (such as course prerequisite) and we want to order the items without violating the dependencies.

Topological Sort

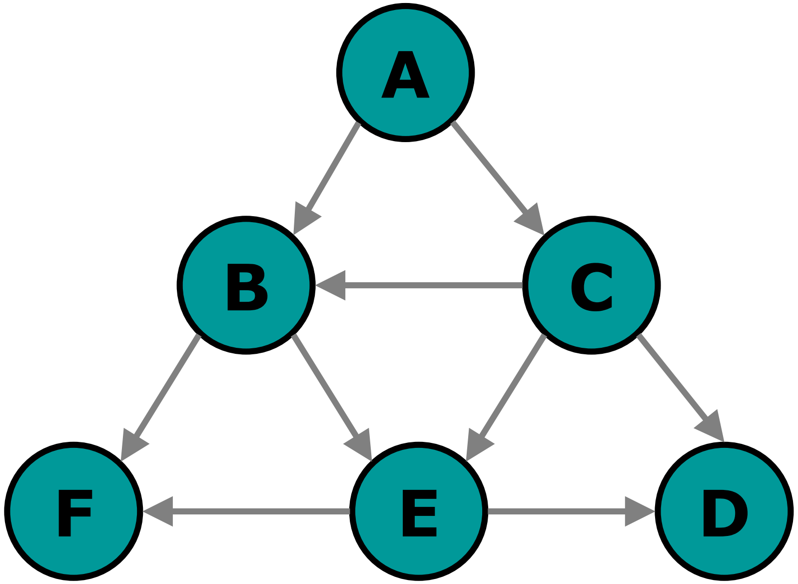
- Topological sort is not unique. In the graph above, ACBEFD and ACBEDF are both valid topological sorted orders. ACDBEF is NOT topologically sorted because D appears before B and there is a path from B to D.



We perform topological sort by repeatedly en-queueing 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 en-queued vertices with 0 in-degree into the queue is not unique, the output is not unique.

Topological Sort

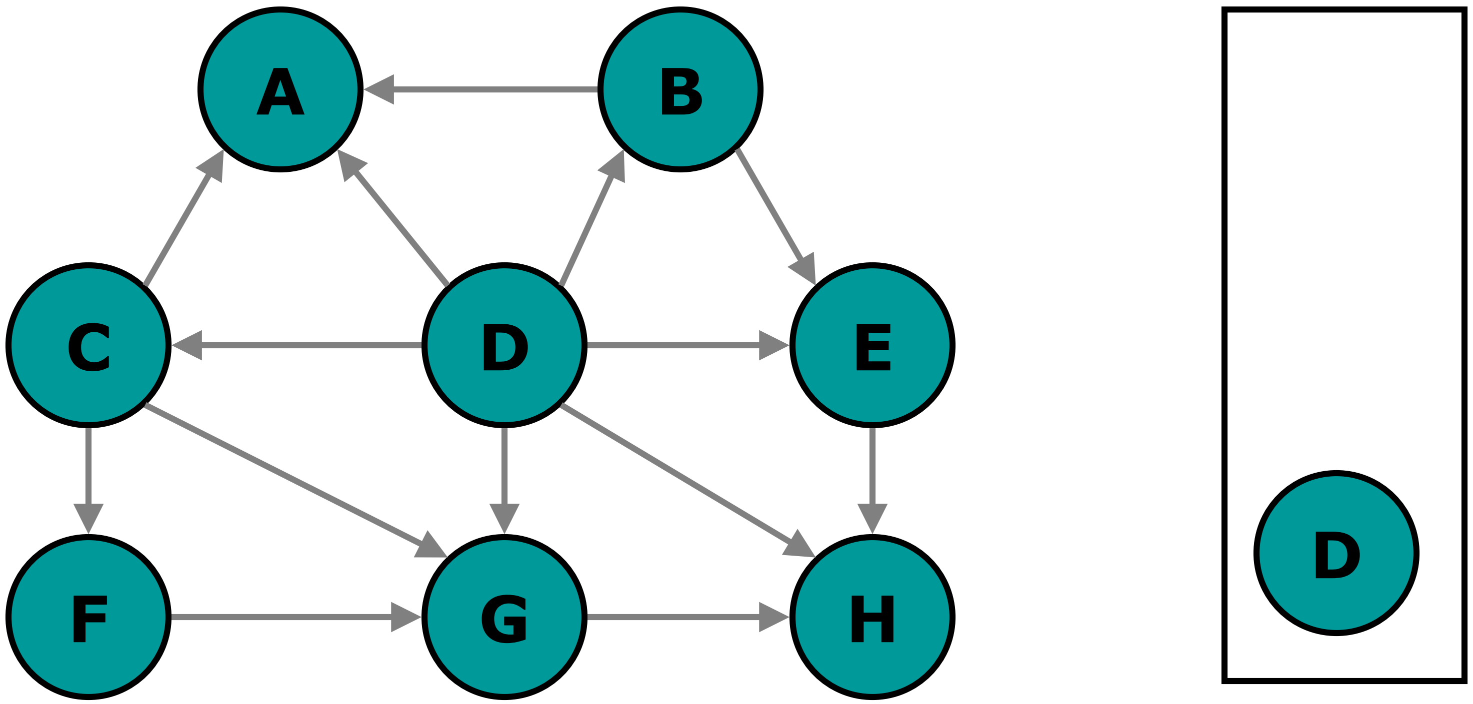
- ACBEFD yes
- ACBEDF yes
- ACDBEF no



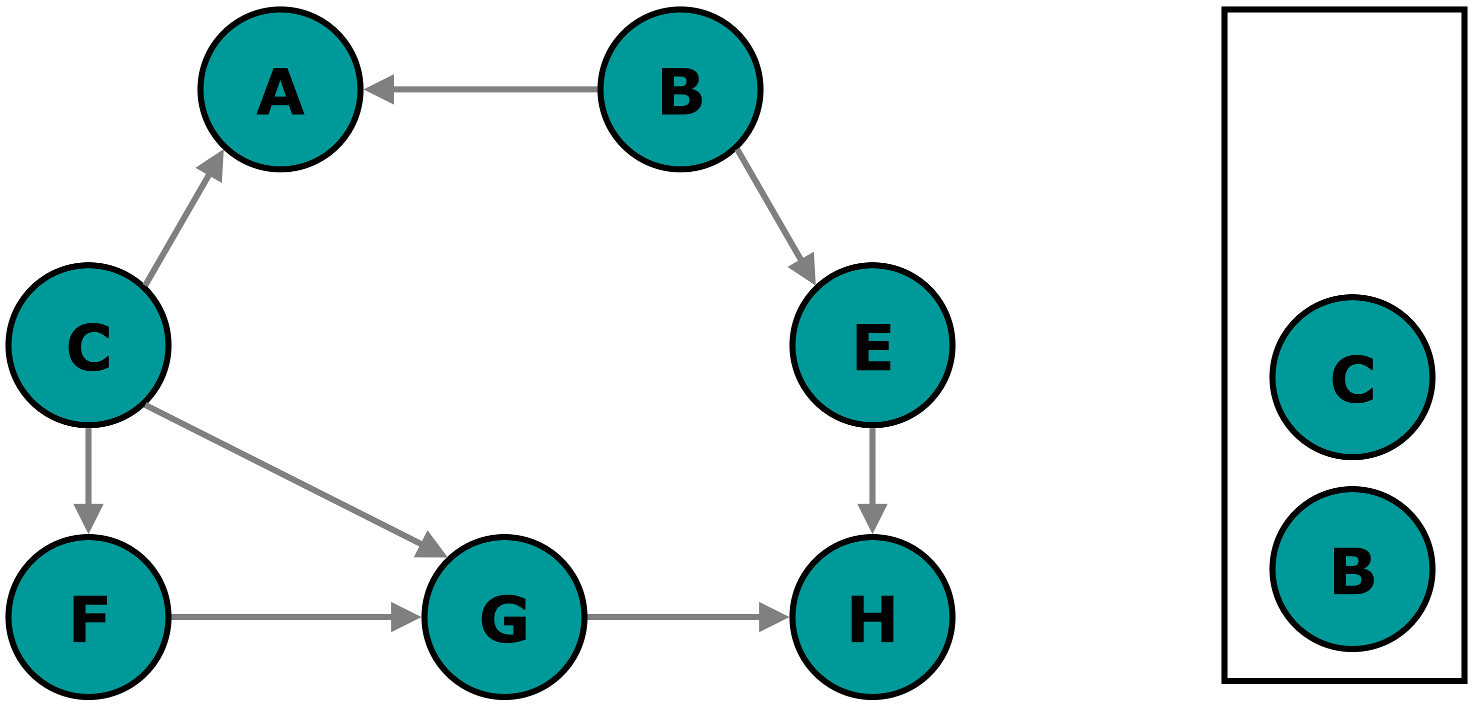
Pseudocode for Toposort

```
q = new Queue()
put all vertices with in-degree 0 into q
while q is not empty
    v = q.deq()
    print v
    remove v from G
    enqueue neighbours of v with in-degree 0
```

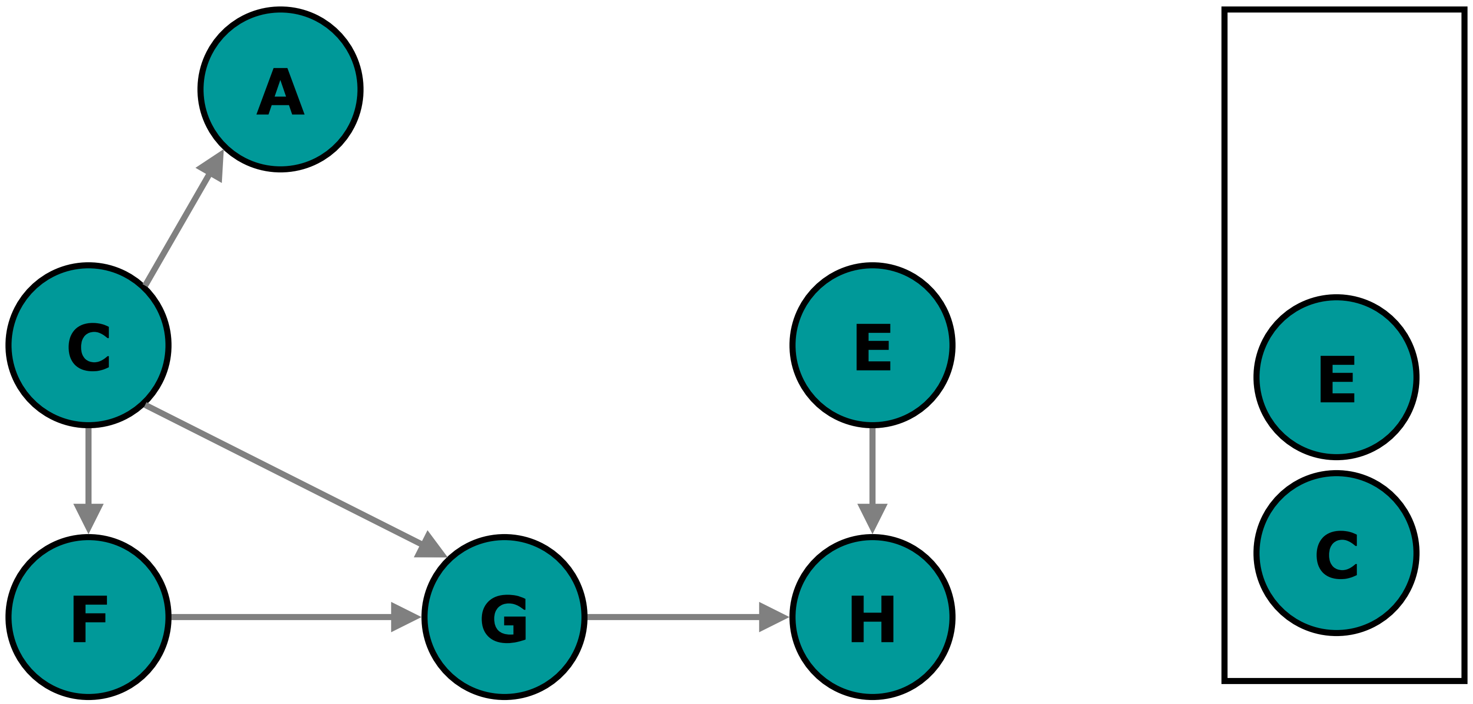
Example



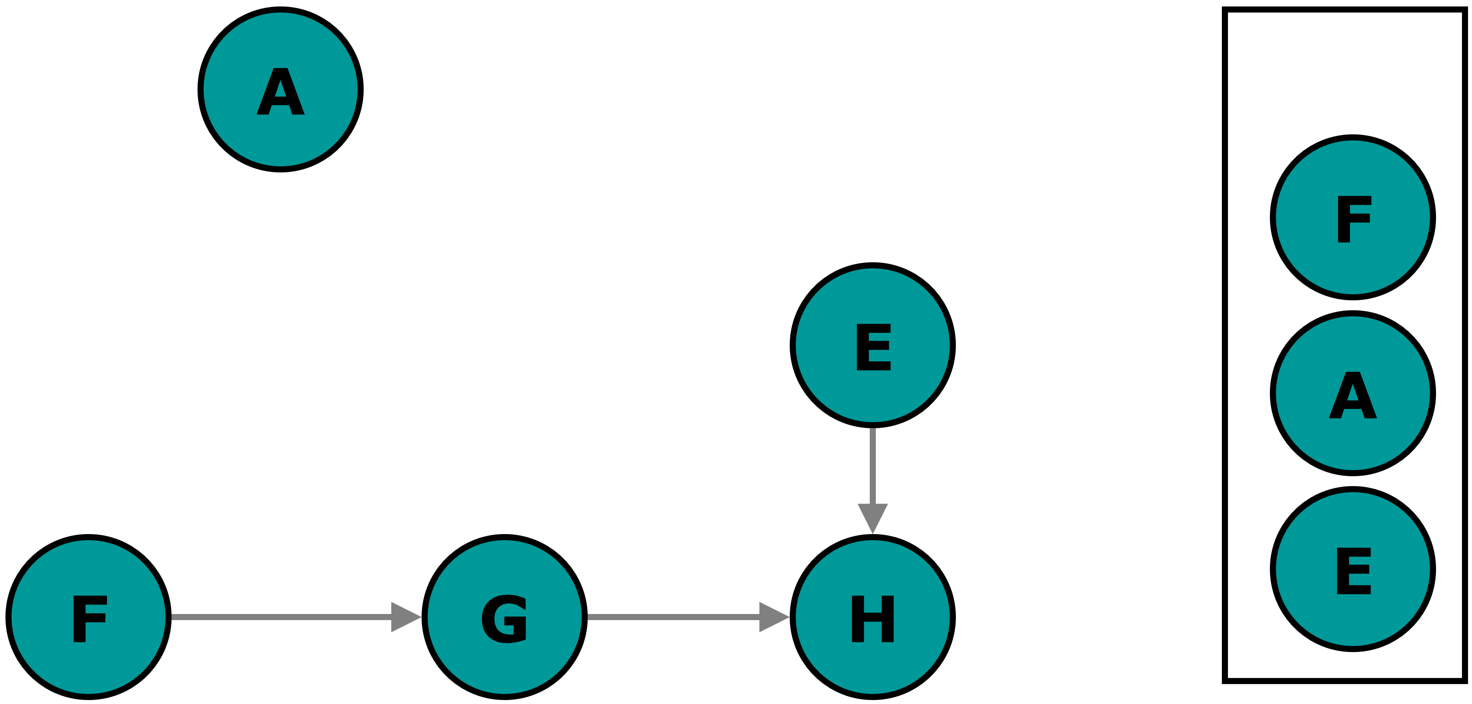
Output: D



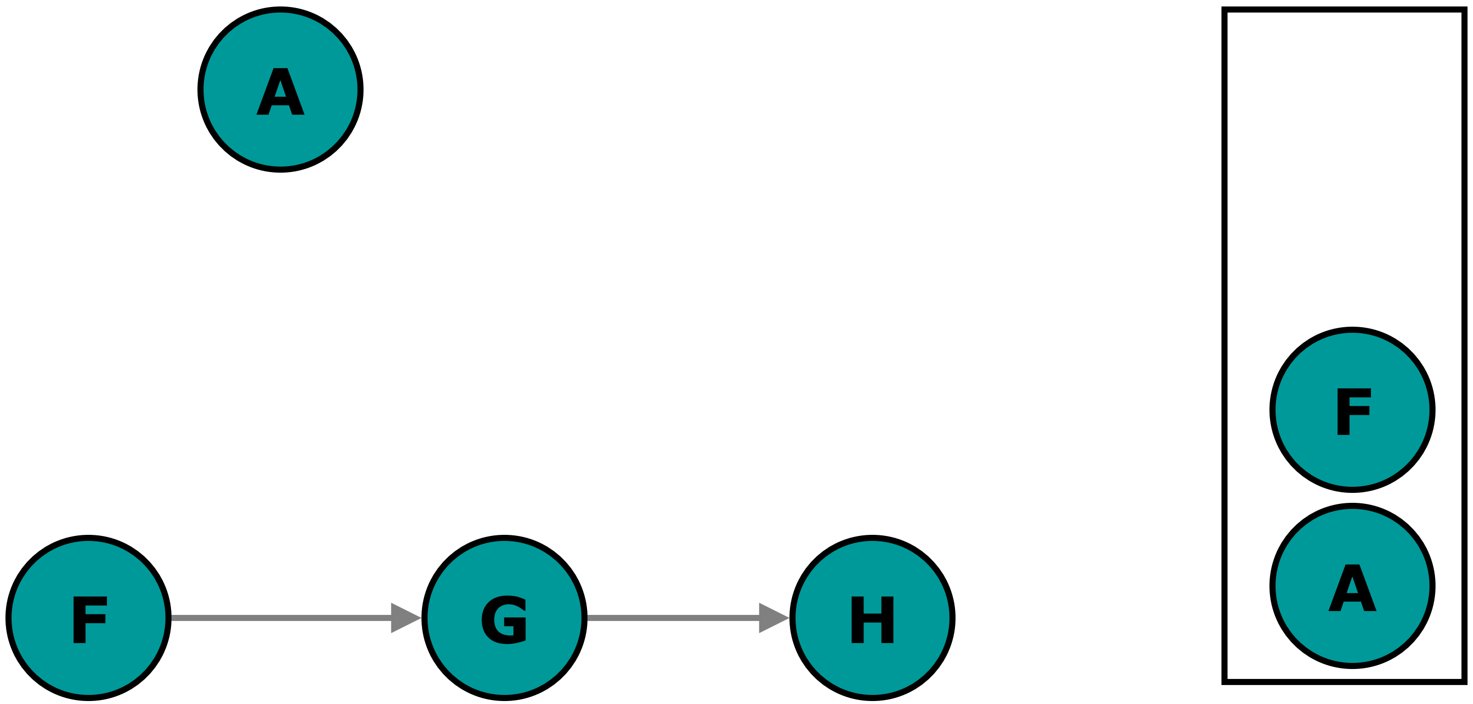
Output: DB



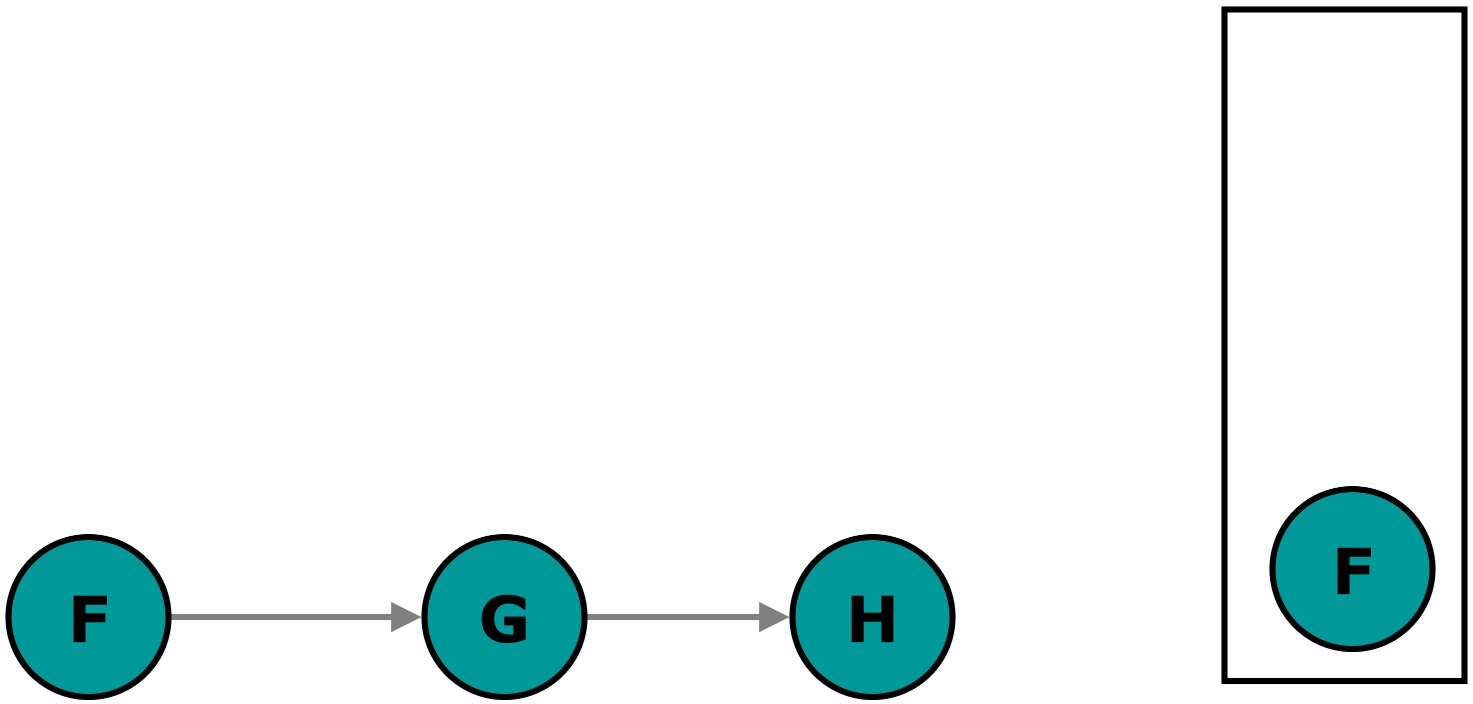
Output: DBC



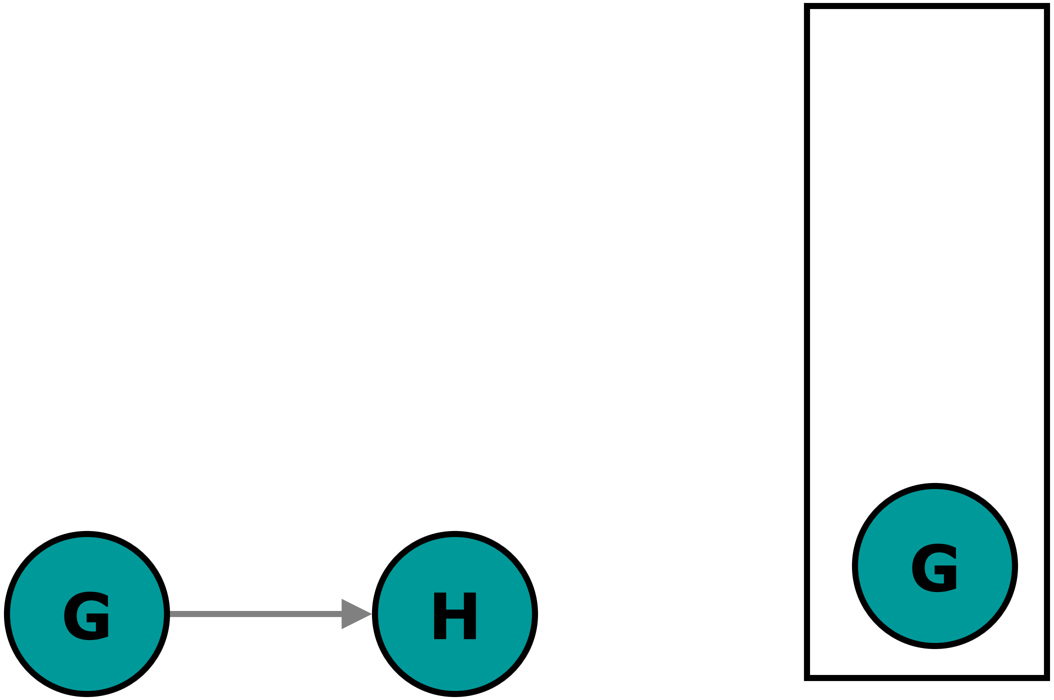
Output: DBCE



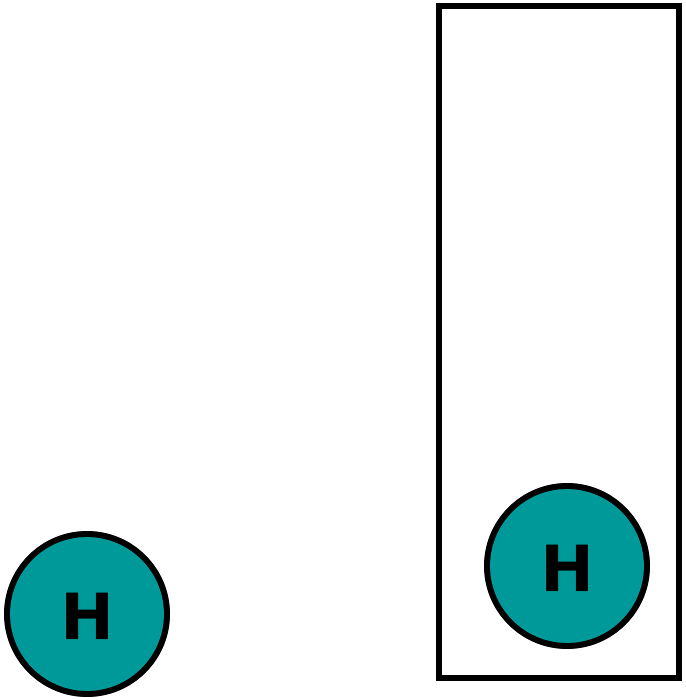
Output: DBCEA



Output: DBCEAF



Output: DBCEAFG



Output: DBCEAFGH

Summary

- terminology of graphs
- Many applications using graphs
- Implemented using Adjacency Matrix, Adjacency List, Edge List
- Applications using AM, AL, EL
- Traversing graphs using BFS/DFS
- Topological sort