Algorithms – I (CS29003/203)

Autumn 2022, IIT
Kharagpur

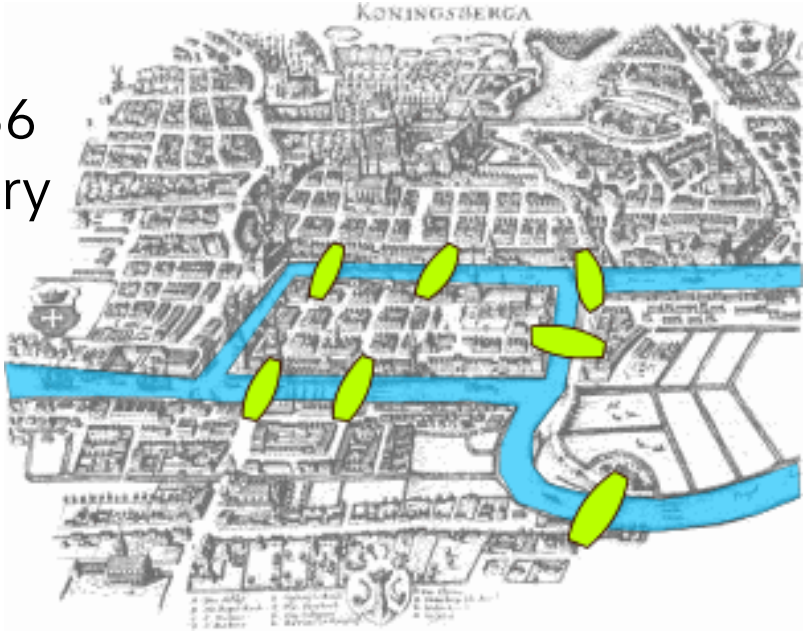
Graphs

Resources

- Apart from the book
- UC Davis ECS 36C Course by Prof. Joël Porquet-Lupine
- UC Riverside, CS 141 course, Fall 2021 by Prof. Yan Gu and Prof. Yihan Sun

Introduction - Seven Bridges of Königsberg

- Context
  - Paper published by Leonhard Euler in 1736
  - Regarded as first in history of graph theory



- Problem
  - City of Königsberg in old Prussia
    - Split in four pieces: 2 mainland portions and 2 islands
    - Interconnected by 7 bridges
  - Problem: is it possible to walk through the city, crossing each bridge once and only once?

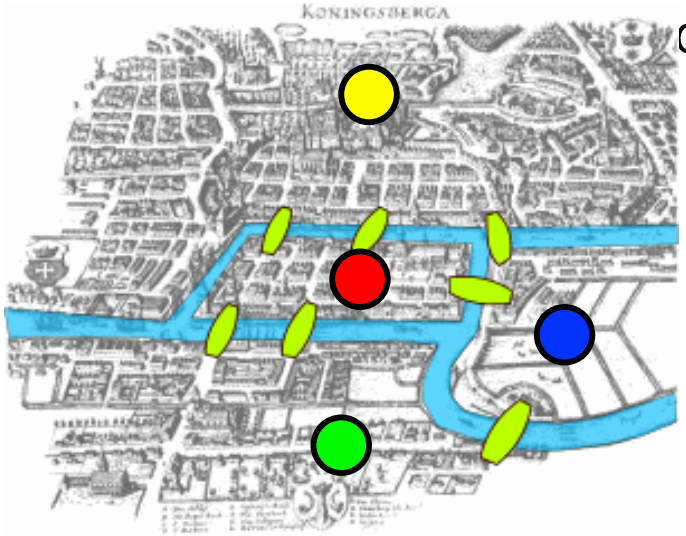
- Euler proved that the problem had no solution
- Developed new technique of analysis to support his proof
- Know Euler - Very interesting talk (must see if you ask me) - Link

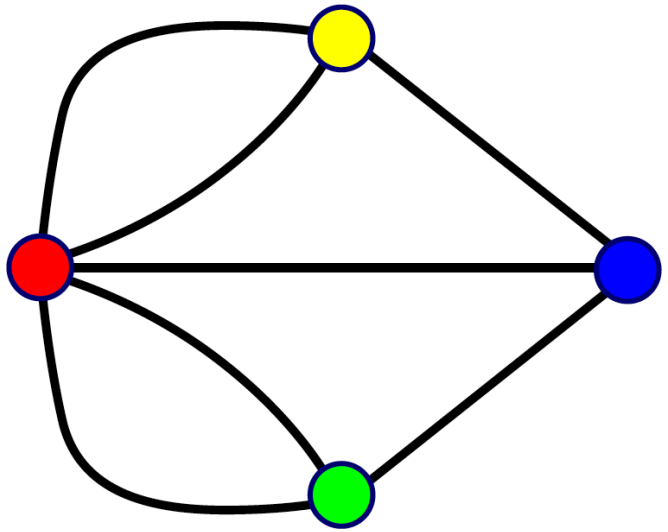
Source: UC Davis, ECS 36C course, Spring 2020

Image is taken from wikipedia

Introduction - Seven Bridges of Königsberg

- Representation
  - Reformulated the problem in abstract termsRegarded as first in history of
    - Land mass: vertex (or node)
    - Bridge: edge
  - a graph



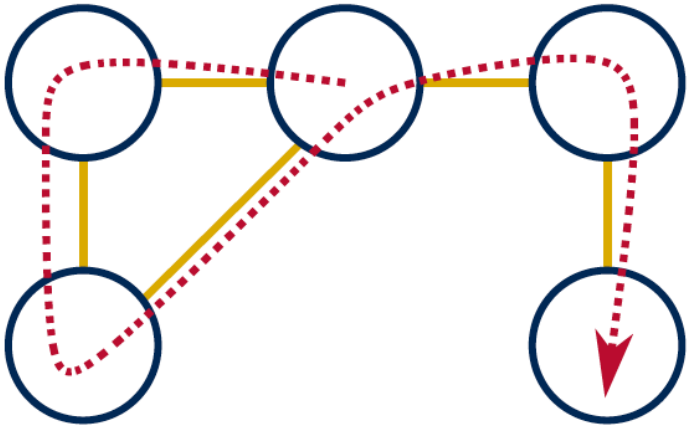


- Analysis
  - Need graph to be connected
  - Possibility of a walk depends on the degrees of the nodes

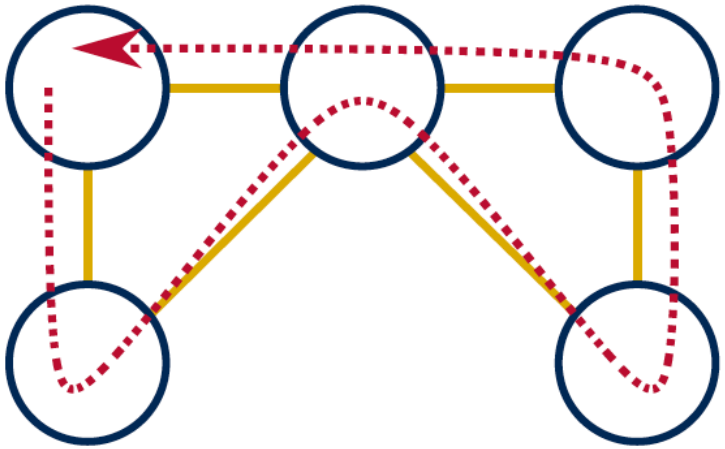
Source: UC Davis, ECS 36C course, Spring 2020

Introduction - Seven Bridges of Königsberg

- Euler's proposals
- Eulerian path
  - Aka Euler walk
  - Path traversing all edges exactly once
  - Only possible if graph has exactly zero or two nodes of odd degree
    - If two nodes of odd degree: start path at one and end at the other



- Eulerian circuit
  - Aka Euler tour
  - Path traversing all edges with same start and end point
  - Only possible if all nodes are of even degree

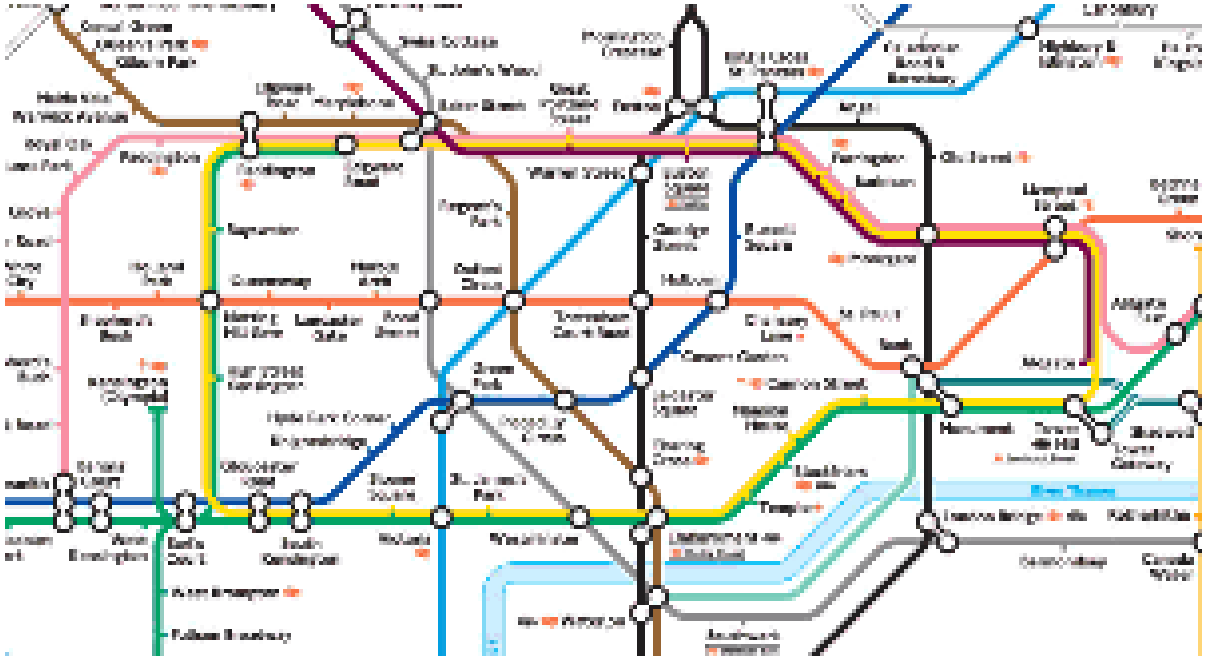


Source: UC Davis, ECS 36C course, Spring 2020

Introduction - Graphs

- Set of vertices connected pairwise by edges
- Extremely versatile, apply to thousands of practical applications
- Many known algorithms to manipulate graphs

- Examples - Subway map



London tube map

Source: UC Davis, ECS 36C course, Spring 2020

Introduction - Graphs

- Examples – Facebook World Friendship Map

Social graph of 500 million people

Source: UC Davis, ECS 36C course, Spring 2020

Introduction - Graphs

- Other application examples

Graph	Vertex	Edge
LAN	Computer	Fiber optic cable
Circuit	Gate, register, processor	Wire
Finance	Stock, currency	Transactions
Transportation	Street intersection, airport	Roads, airway routes
Neural network	Neuron	Synapse
Molecule	Atom	Bond

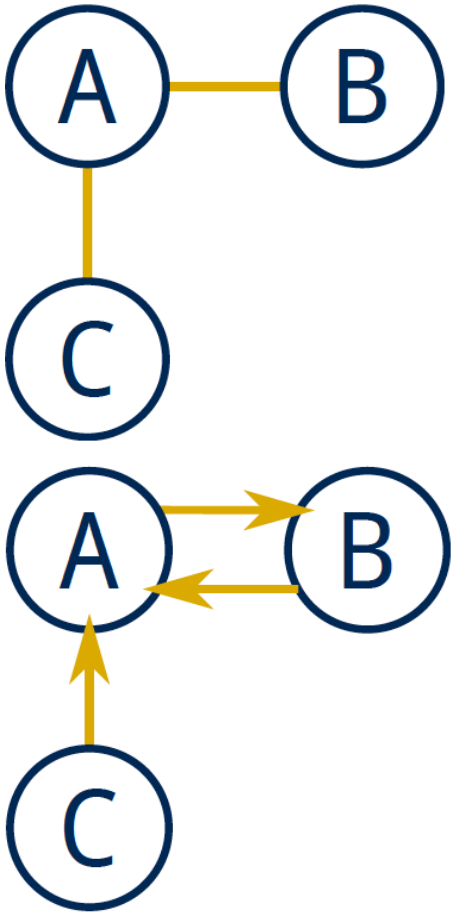
Source: UC Davis, ECS 36C course, Spring 2020

Graphs - Definitions

- $G = (V, E)$
  - V: Set of vertices or nodes
  - E: set of edges (or links, arcs) --pair of two vertices

- Undirected graph
  - Edges are all bidirectional
  - $V = \{A, B, C\}$
  - $E = \{\{A, B\}, \{A, C\}\}$
  - E.g., Facebook friends, most roads, most flights

- Directed graph
  - Edges are uni-directional
  - Aka digraphs
  - $V = \{A, B, C\}$
  - $E = \{(A, B), (B, A), (C, A)\}$
  - E.g., Twitter follows, code analysis

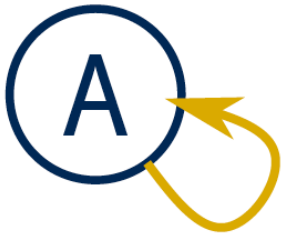


Source: UC Davis, ECS 36C course, Spring 2020

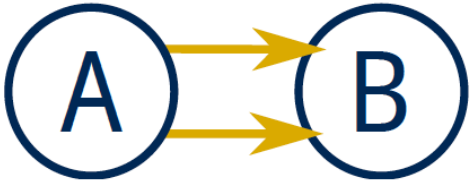
Graphs - Definitions

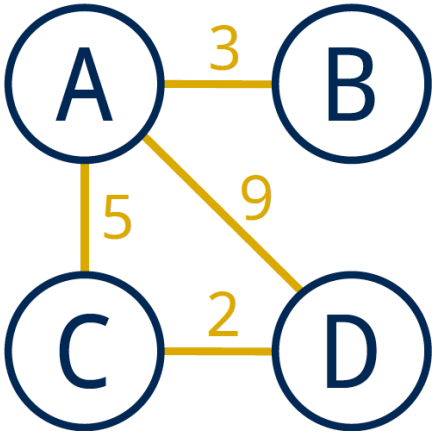
- Multigraph
  - A multigraph is a graph (directed or not) in which multiple edges and loops are allowed
- Simple graph
  - A simple graph is an undirected graph which has neither multiple edges nor loops
  - Default definition of a graph
- Weighted graph
  - In a weighted graph, each edge is associated with a weight
  - Weights usually represent costs, lengths, capacities, etc.
  - Depends on the problem at hand

Loops



Multiple edges

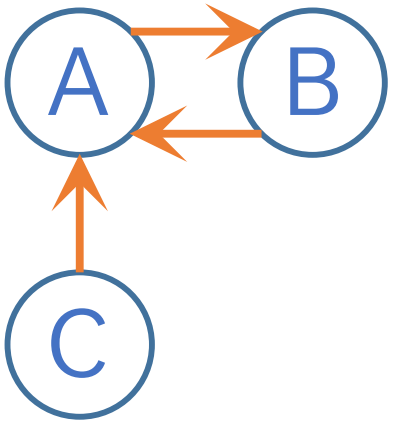




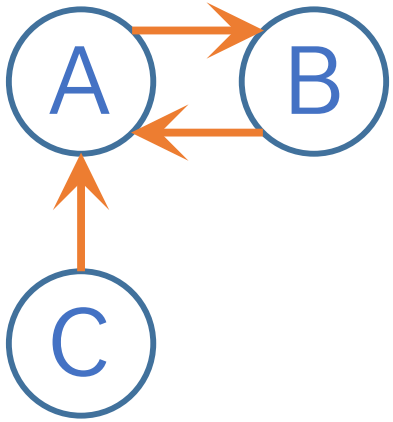
Source: UC Davis, ECS 36C course, Spring 2020

Graphs - Definitions

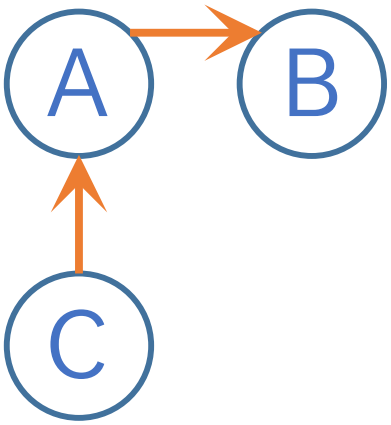
- Degree of a vertex
  - Number of edges incident on vertex
- Only relevant for digraphs:
  - In-degree: number of edges incident to vertex
  - Out-degree: number of edges incident from vertex



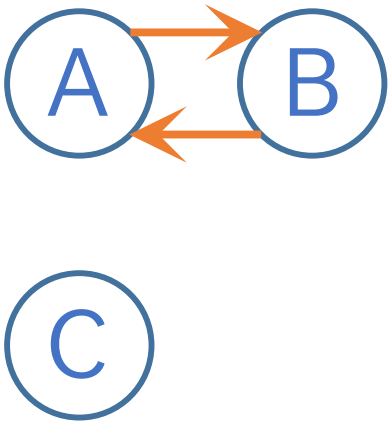
- 0-degree vertices



C: in-degree of 0



B: out-degree of 0



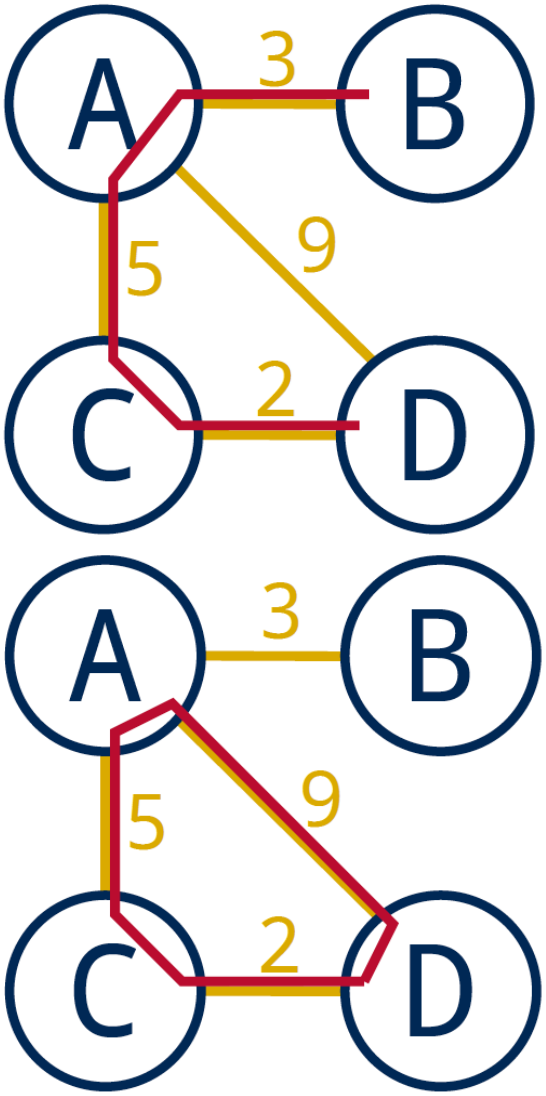
C: degree of 0

- All these graphs are valid

Source: UC Davis, ECS 36C course, Spring 2020

Graphs - Definitions

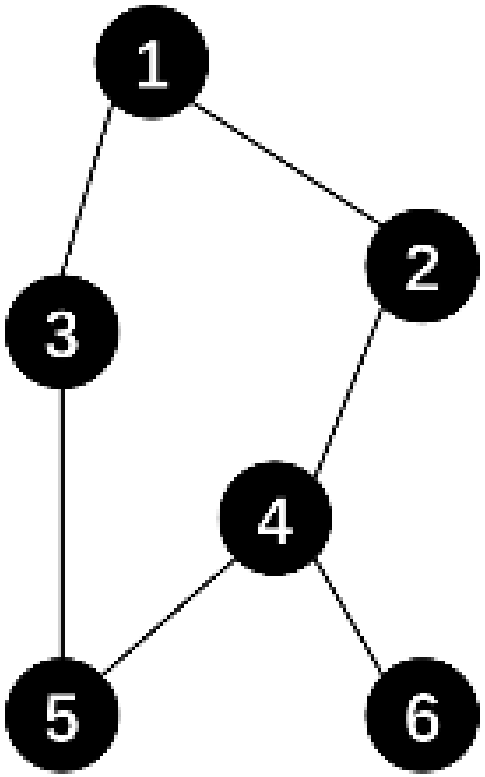
- Path
  - Sequence of vertices connected by edges
  - Length: number of edges along the path
  - Weight: sum of all edge weights along the path



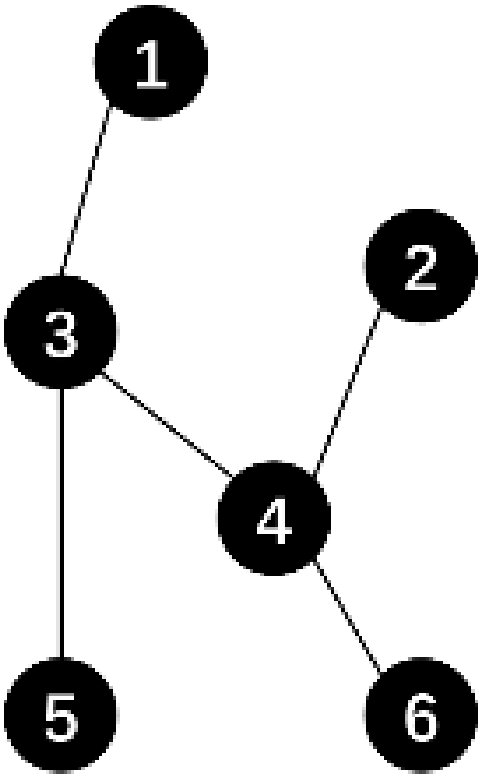
- Cycle
  - Path whose first and last vertices are the same
  - A graph containing no cycles is said to be acyclic

Source: UC Davis, ECS 36C course, Spring 2020

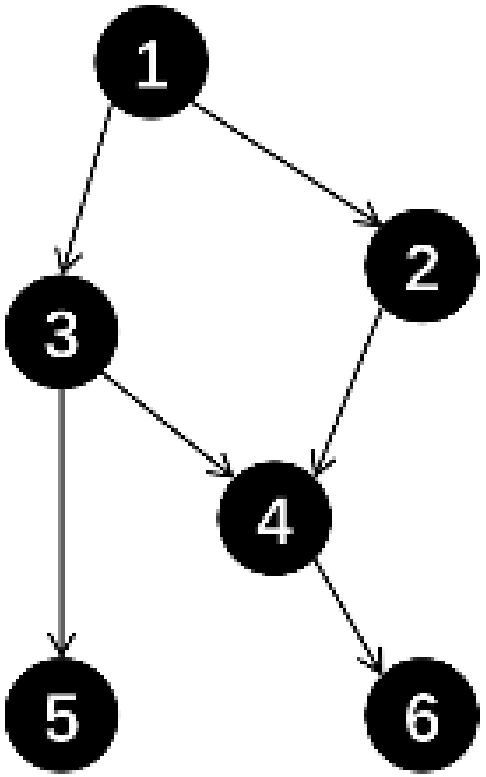
Graphs - Definitions



Cyclic Graph



Acyclic Graph



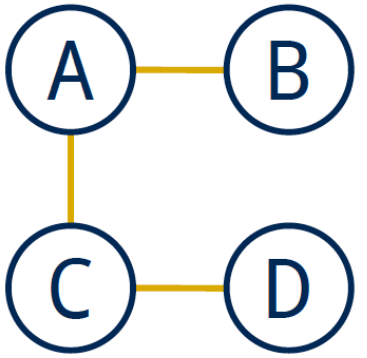
Directed Acyclic Graph (DAG)

Source: UC Riverside, CS141 course, Fall 2021

Graphs - Definitions

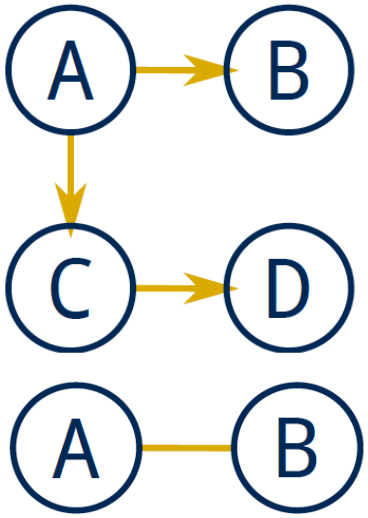
Connected graph

- Vertices are connected if there is a path between them
- For directed graphs
  - Strongly connected if directed path between two vertices
  - Weakly connected otherwise
- A connected graph is a graph in which every pair of vertices are connected
- If every pair of vertices is joined by an edge, the graph is complete or fully-connected



Disconnected graph

- A disconnected graph is composed of two or more connected components



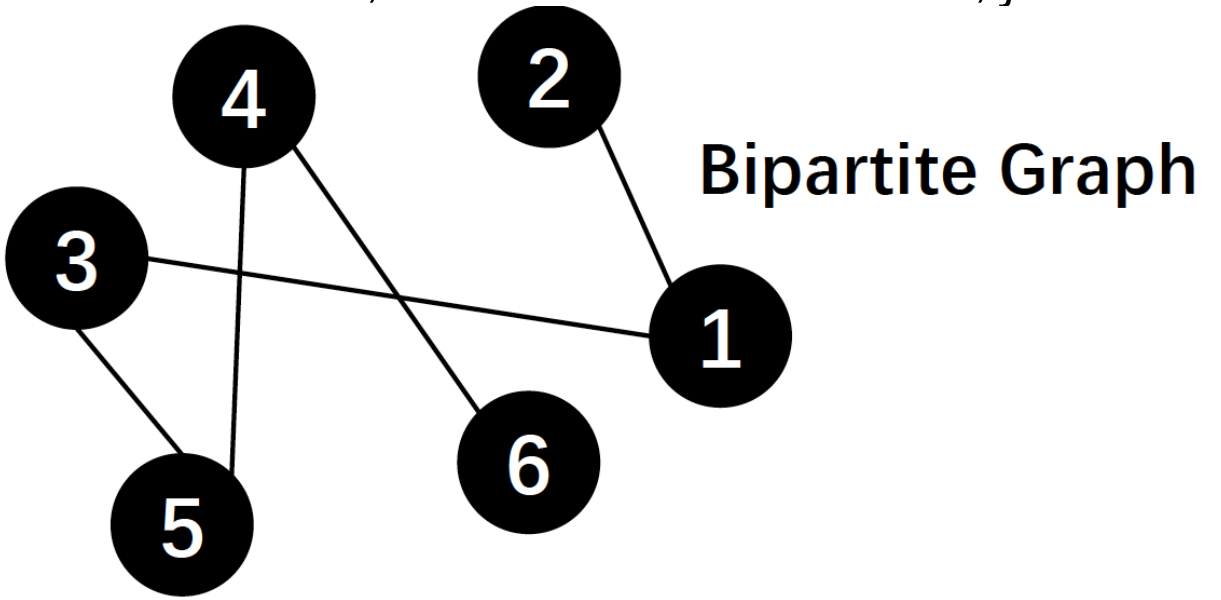
Source: UC Davis, ECS 36C course, Spring 2020

Graphs - Definitions

Bipartite graph

- A graph where the vertices can be partitioned into two subsets:
- no edges within a subset and all the edges are between two subsets

- Usually, vertices in two subsets have different meanings
  - E.g., students and courses, courses and classrooms, jobs and applicants

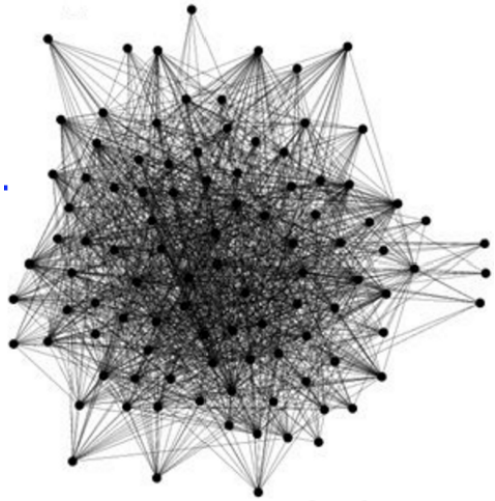


Source: UC Riverside, CS141 course, Fall 2021

Graphs - Definitions

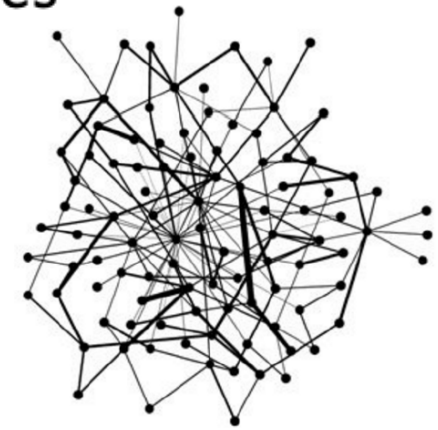
Dense graph

- A graph is dense if it contains a large number of edges
  - $E = \Theta(V^2)$



Sparse graph

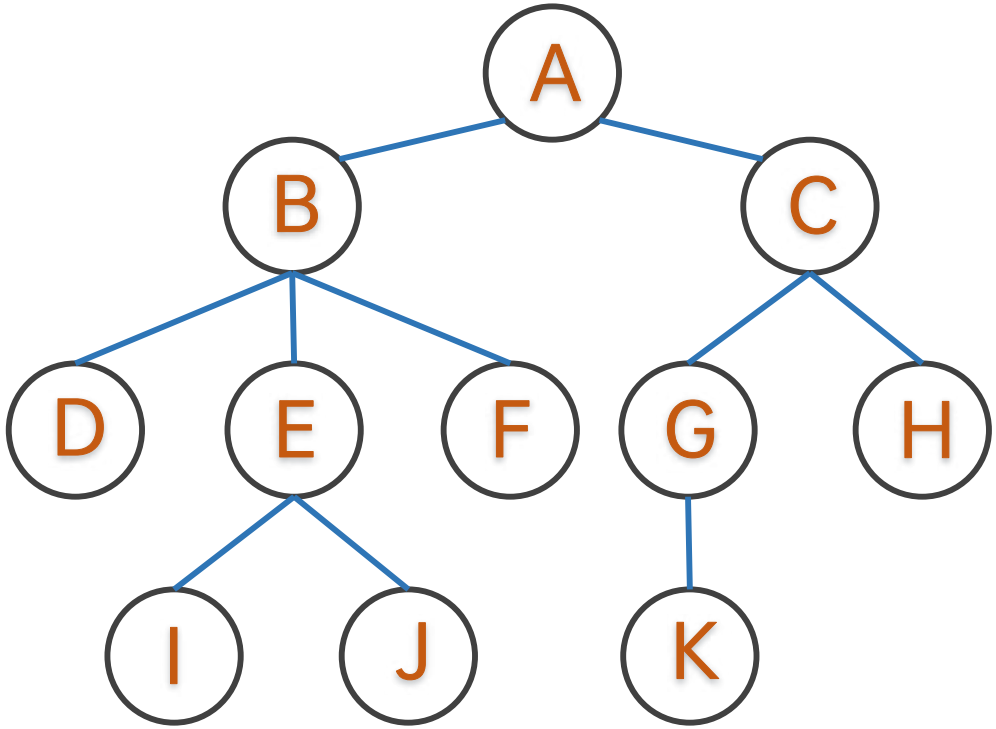
- A graph is sparse if it contains much less than the maximum number of possible edges
  - $E = \Theta(V)$



Source: UC Davis, ECS 36C course, Spring 2020

Trees

- Trees are a specific kind of graph too!

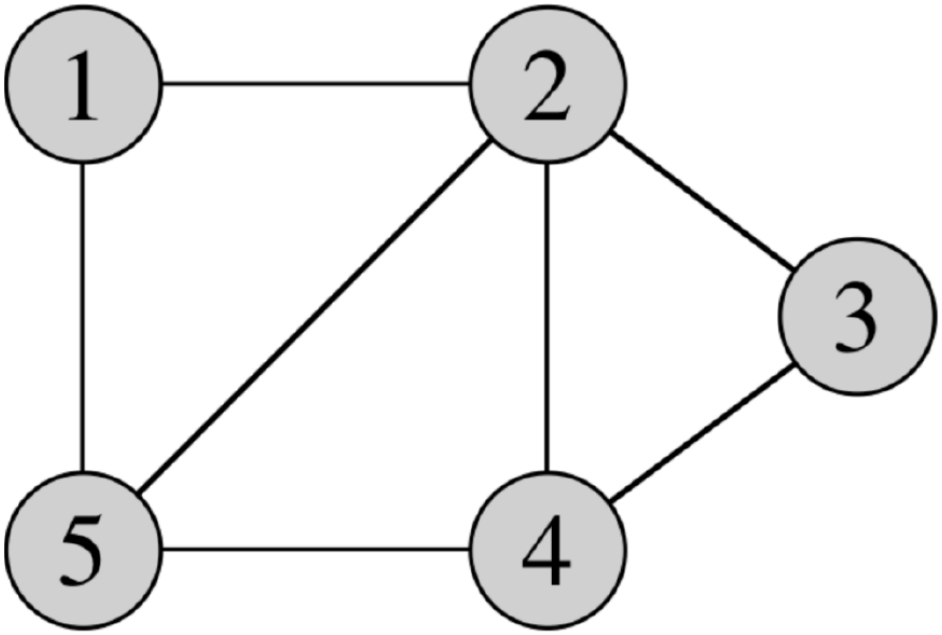


- Undirected, acyclic, connected, rooted graph
- A forest is a graph in which multiple connected components are trees

Source: UC Davis, ECS 36C course, Spring 2020

Graph Representations

- Adjacency matrix

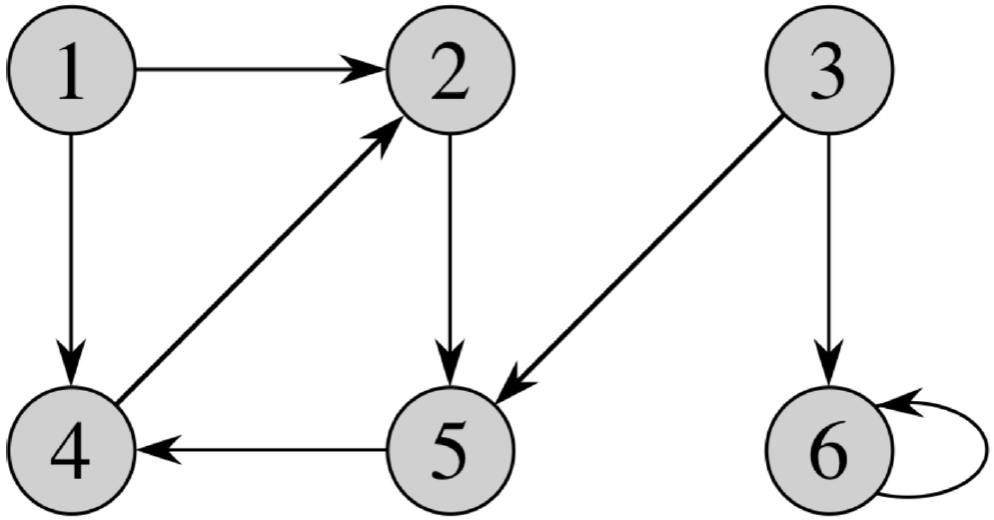


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Source: UC Riverside, CS141 course, Fall 2021

Graph Representations

- Adjacency matrix



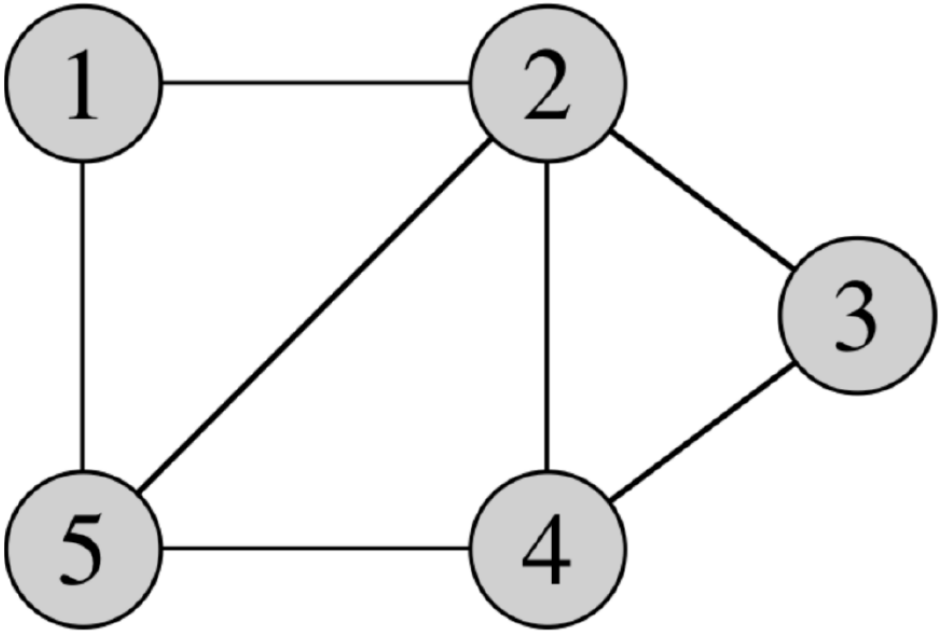
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

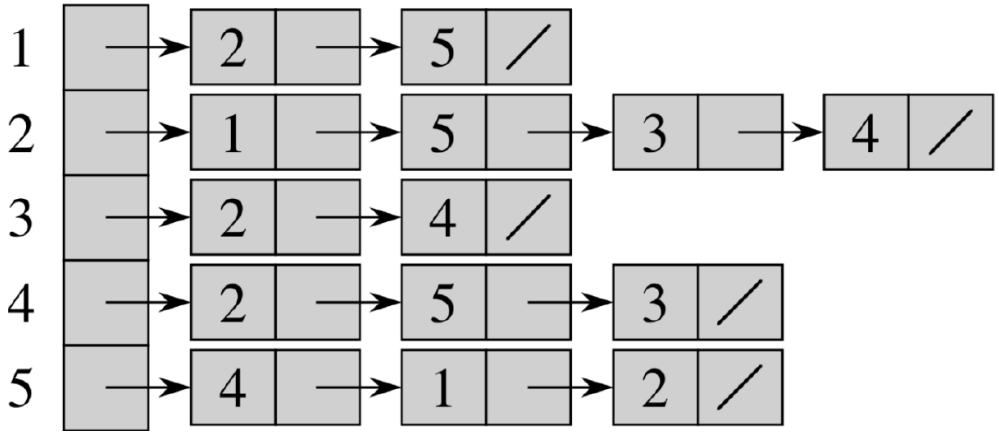
- Takes up too much space $O(n^2)$

Source: UC Riverside, CS141 course, Fall 2021

Graph Representations

- Adjacency list

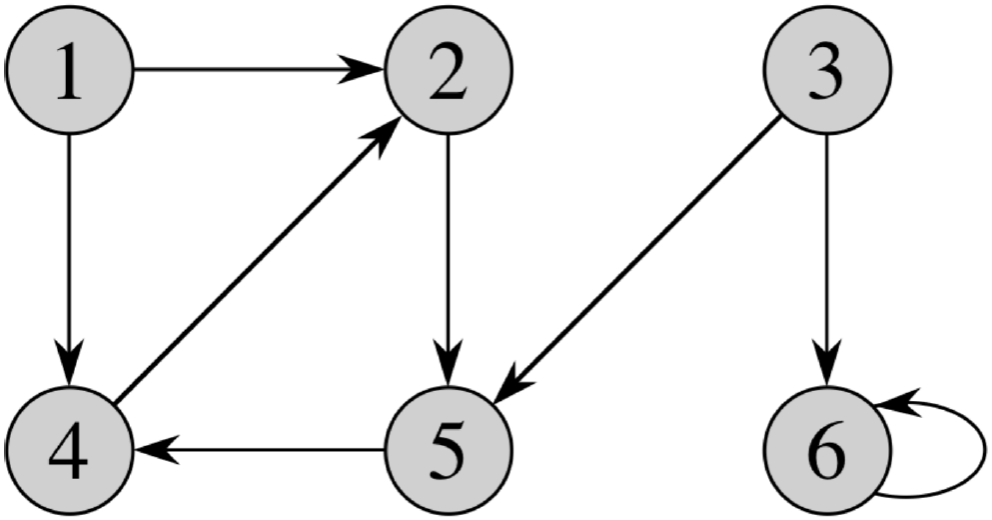


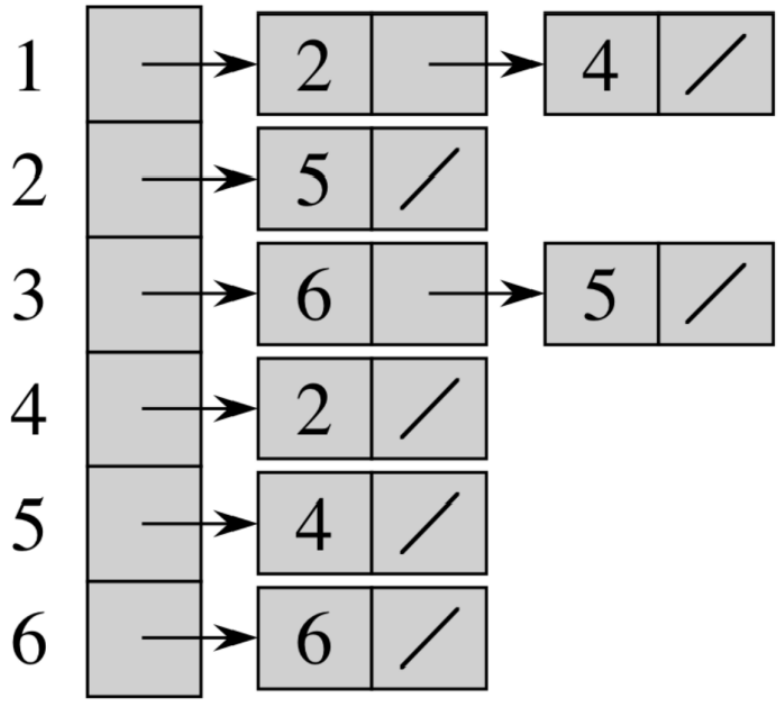


Source: UC Riverside, CS141 course, Fall 2021

Graph Representations

- Adjacency matrix





- What do scientists use in practice?

Source: UC Riverside, CS141 course, Fall 2021

Graph Traversal/Search

- Graph traversals or searches mean systematic following edges of a graph so as to visit all the vertices of the graph
- Graph-searching algorithm can discover much about the structure of a graph
- Many graph algorithms begins by building on basic graph searching

- We will start with two basic graph search algorithms
- Breadth First Search (BFS)
- Depth First Search (DFS)

- We will assume adjacency list representation for today's algorithms

Breadth First Search

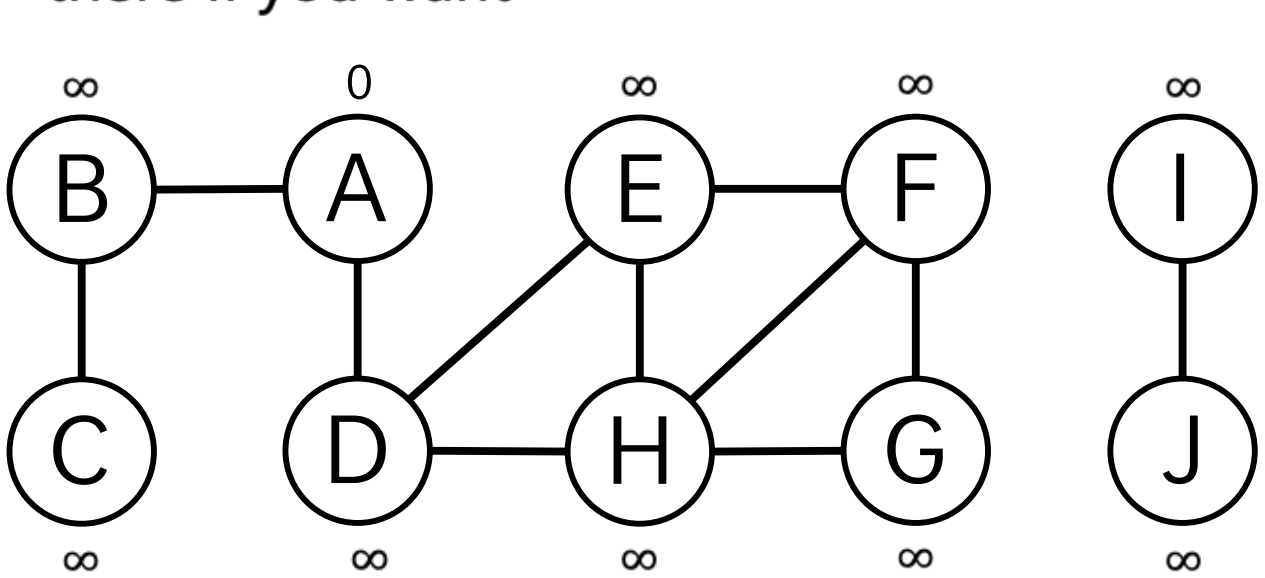
- Given a graph $G = (V, E)$ and a source vertex s, breadth-first search systematically explores the edges to discover every vertex that is reachable from s
- It computes the distance (smallest number of edges) from s to each reachable vertex

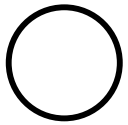
- In the process, it produces a "breadth-first tree" with root s that contains all reachable vertices from s
- For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a "shortest path" (smallest number of edges) from s to v

- The algorithm works on both directed and undirected graphs
- However, we will walk through an example on undirected graph

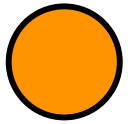
Breadth First Search

- Breadth first search is named so because it visits all vertices at distance k from s before visiting any vertex at distance $k + 1$
- At each node, you can get a list of neighbors, and choose to go there if you want

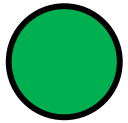




Not been there yet



Been there, haven't explored all the paths out

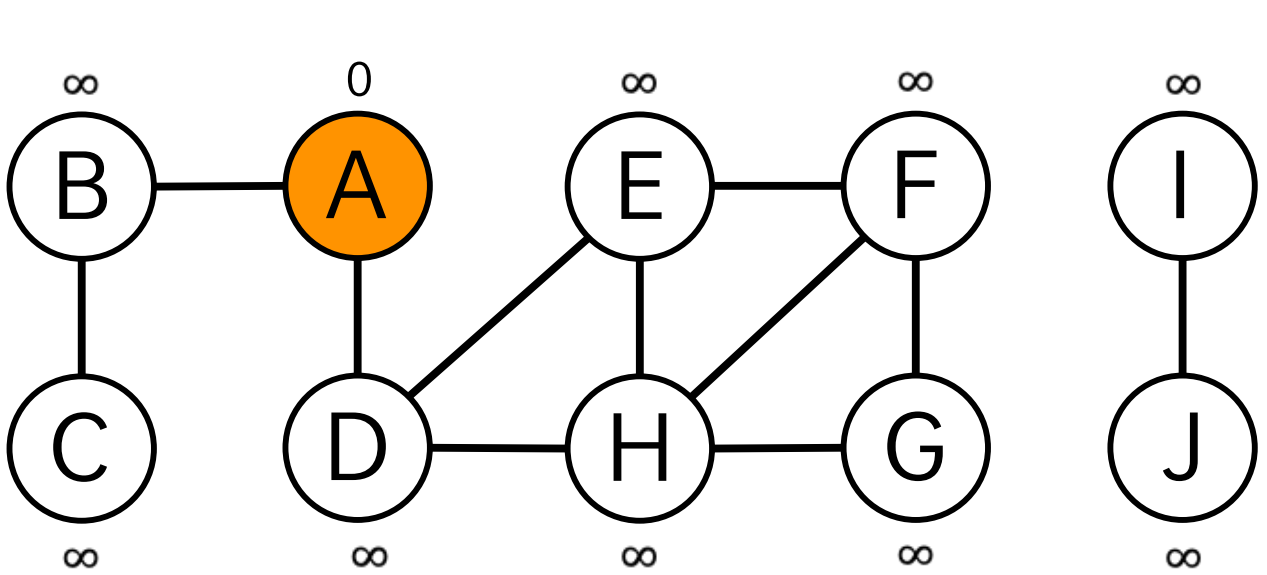


Been there, have explored all the paths out

- Start at the source node and set the initial distance to all other nodes as ∞ except for the source node itself, whose distance is 0

Breadth First Search

queue A

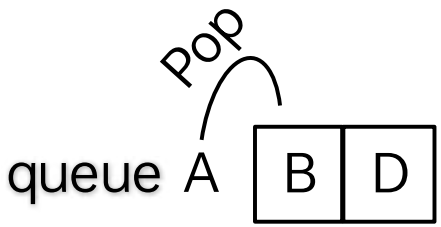


Not been there yet

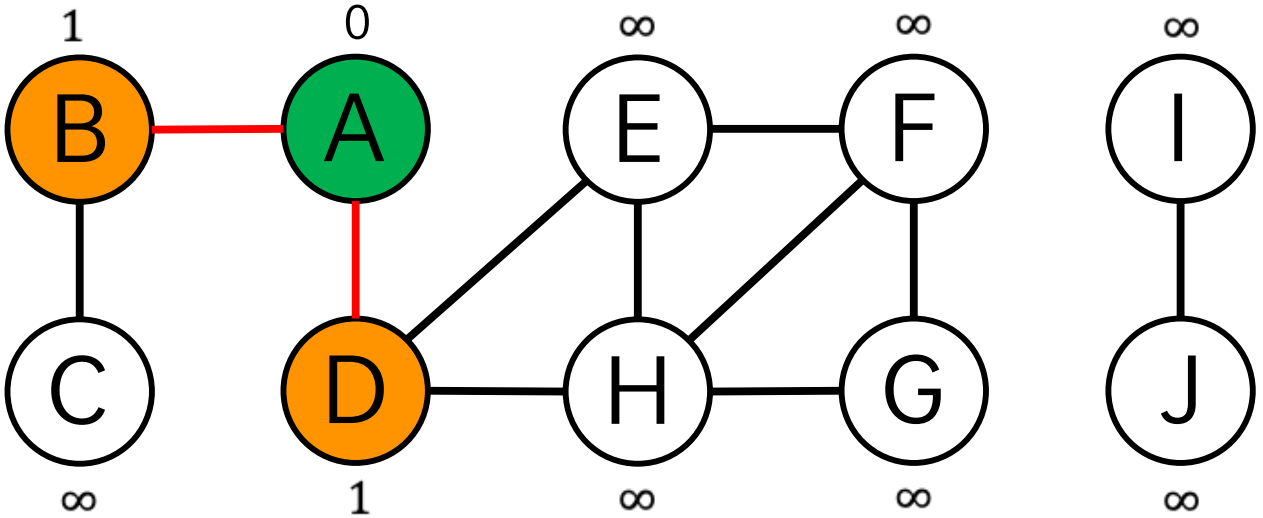
Been there, haven't explored all the paths out

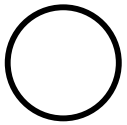
Been there, have explored all the paths out

Breadth First Search

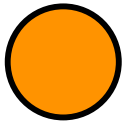


Pops: A

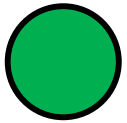




Not been there yet

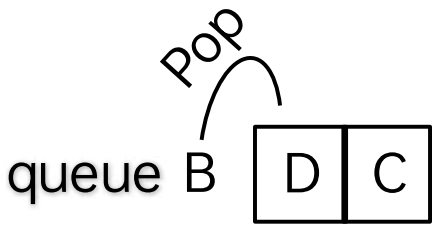


Been there, haven't explored all the paths out

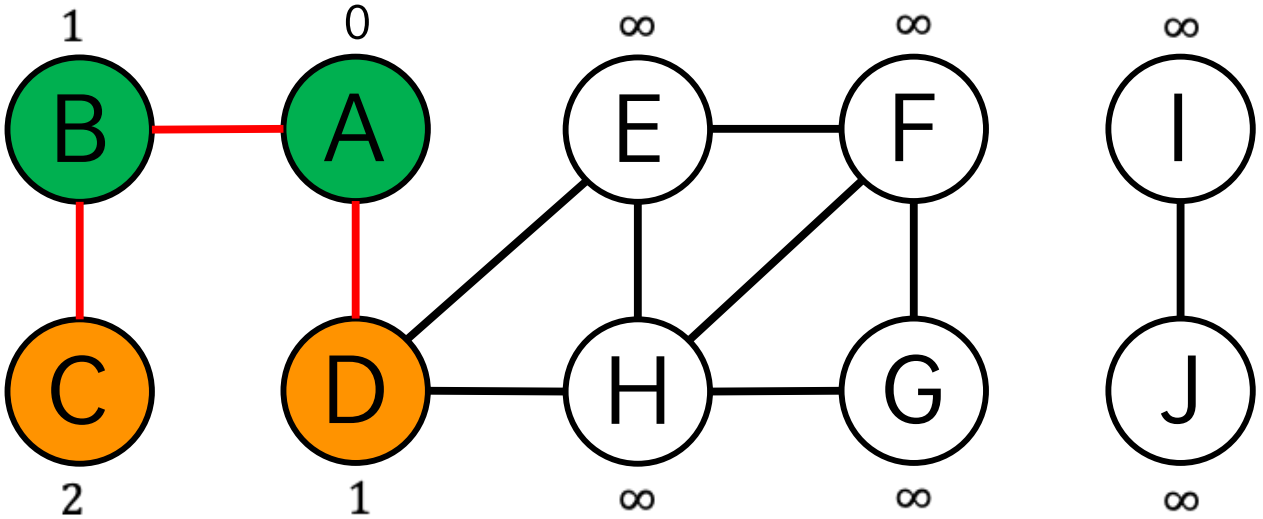


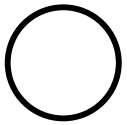
Been there, have explored all the paths out

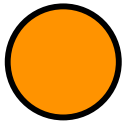
Breadth First Search

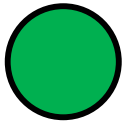


Pops: A B

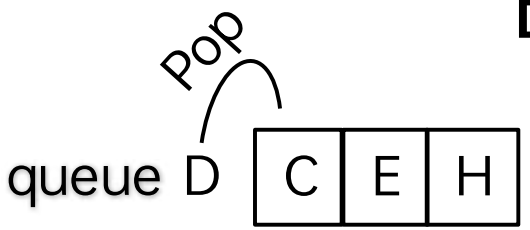



Not been there yet

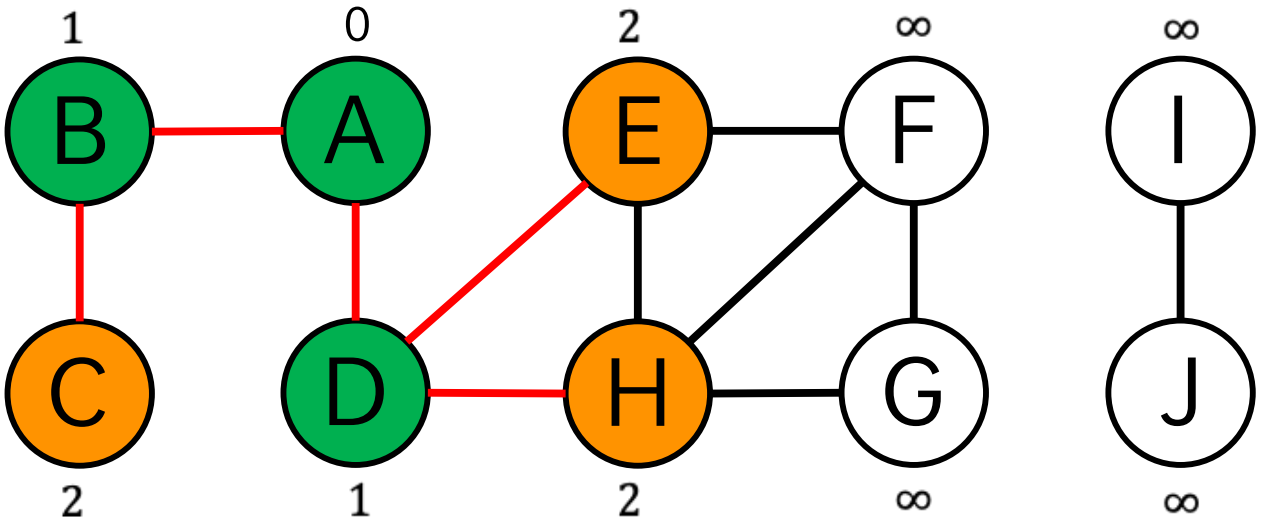

Been there, haven't explored all the paths out


Been there, have explored all the paths out

Breadth First Search



Pops: A B D

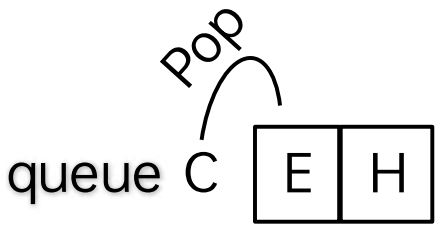


Not been there yet

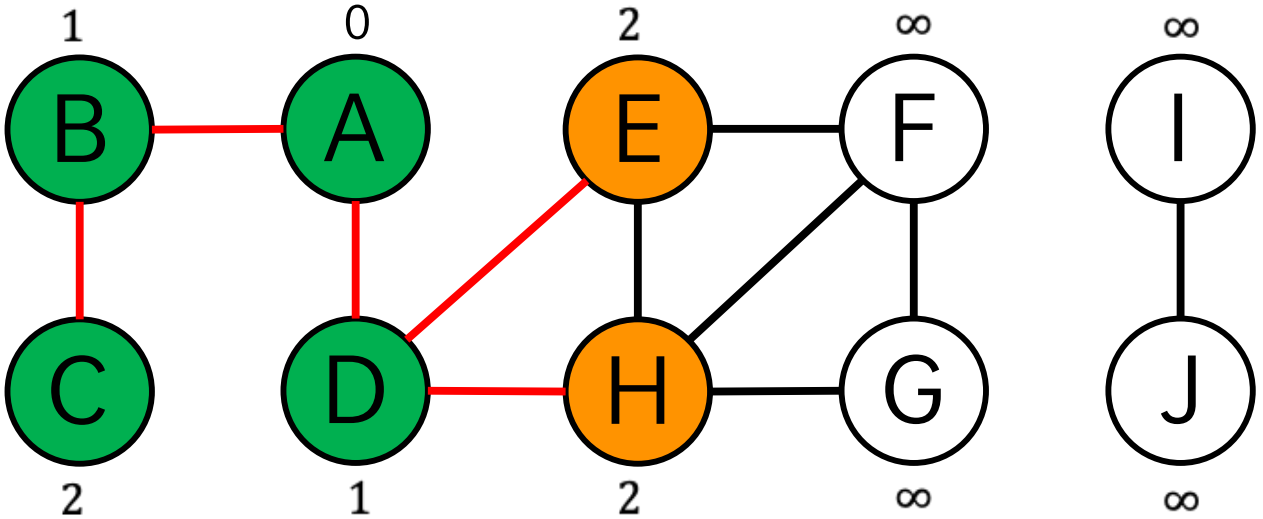
Been there, haven't explored all the paths out

Been there, have explored all the paths out

Breadth First Search



Pops: A B D C

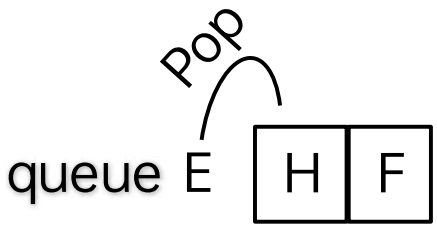


Not been there yet

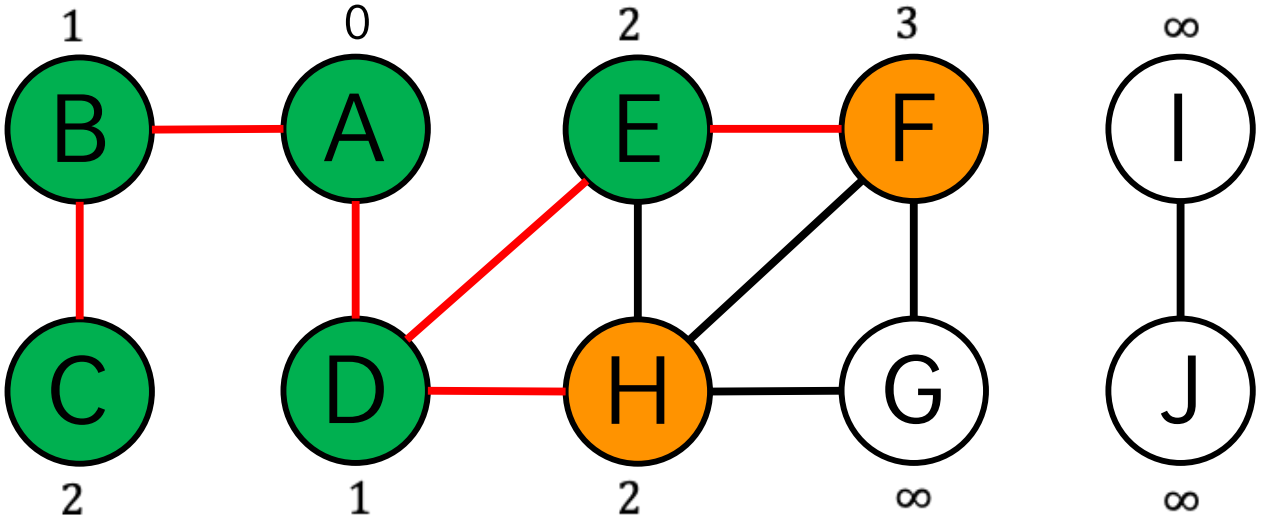
Been there, haven't explored all the paths out

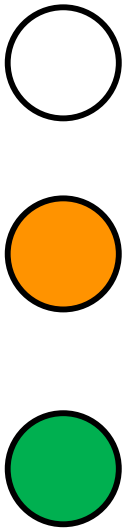
Been there, have explored all the paths out

Breadth First Search

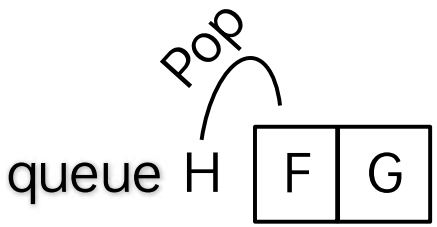


Pops: A B D C E

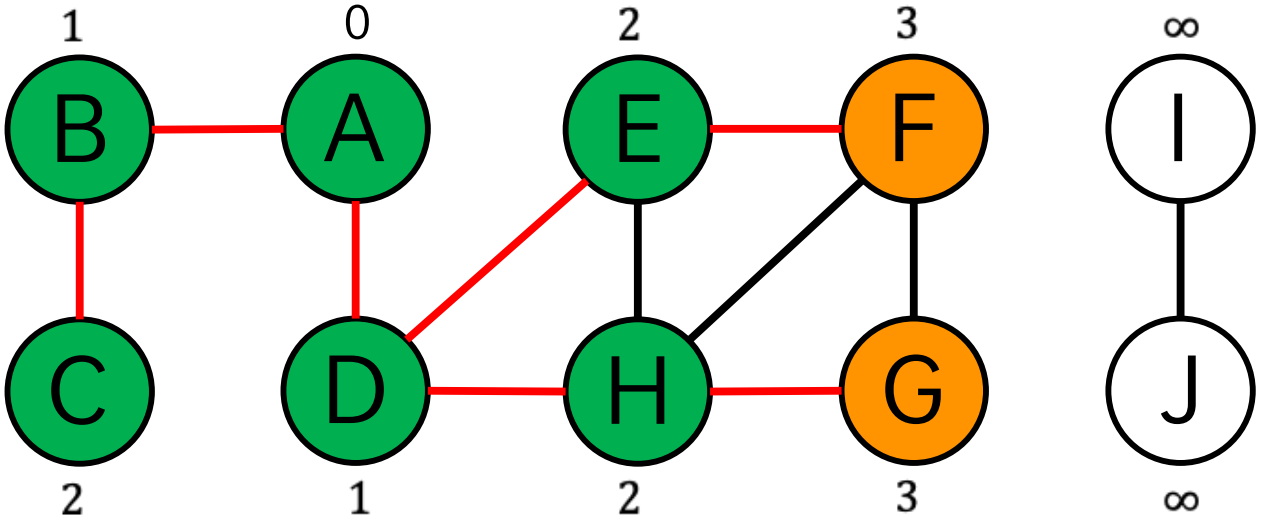


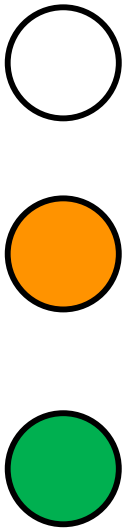


Breadth First Search

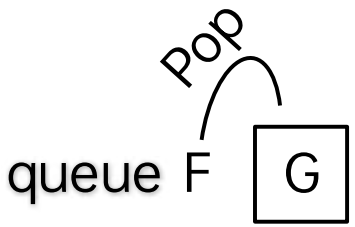


Pops: A B D C E H

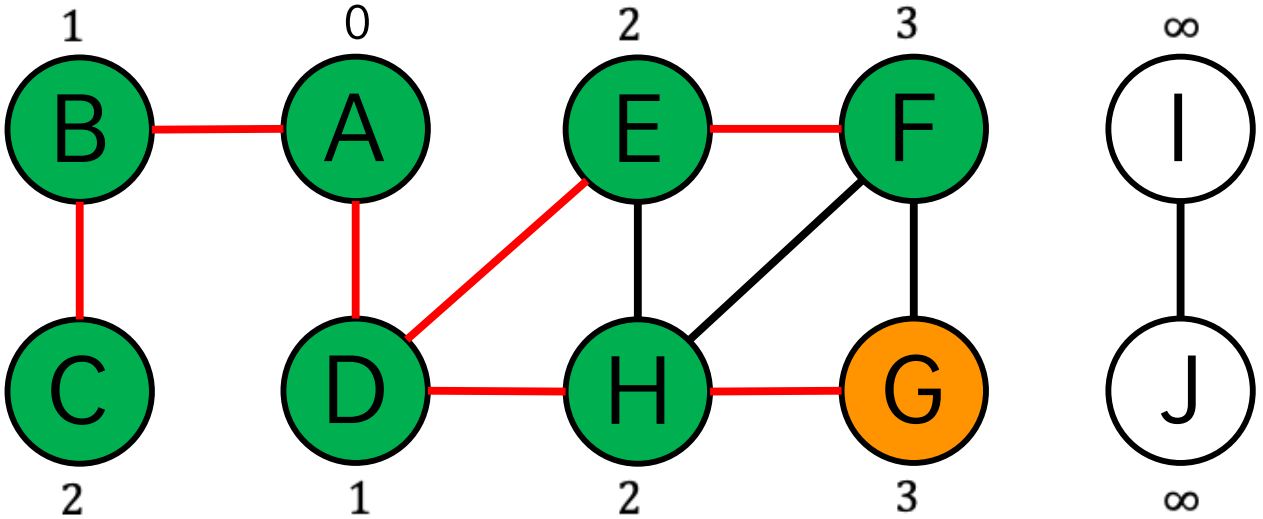


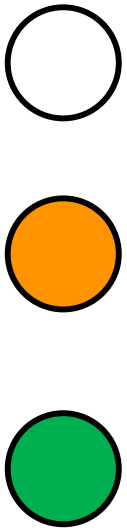


Breadth First Search

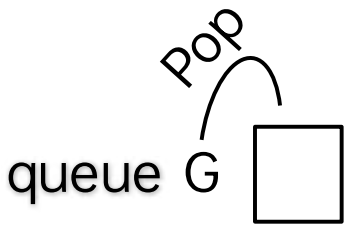


Pops: A B D C E H F

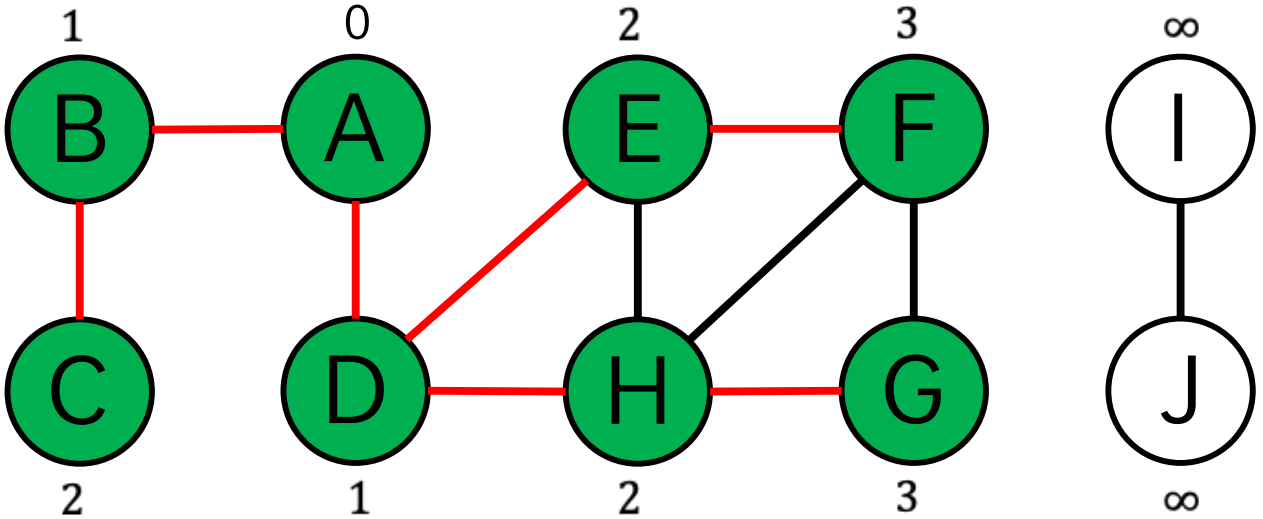


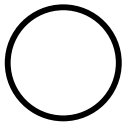


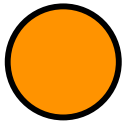
Breadth First Search

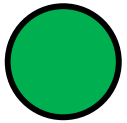


Pops: A B D C E H F G



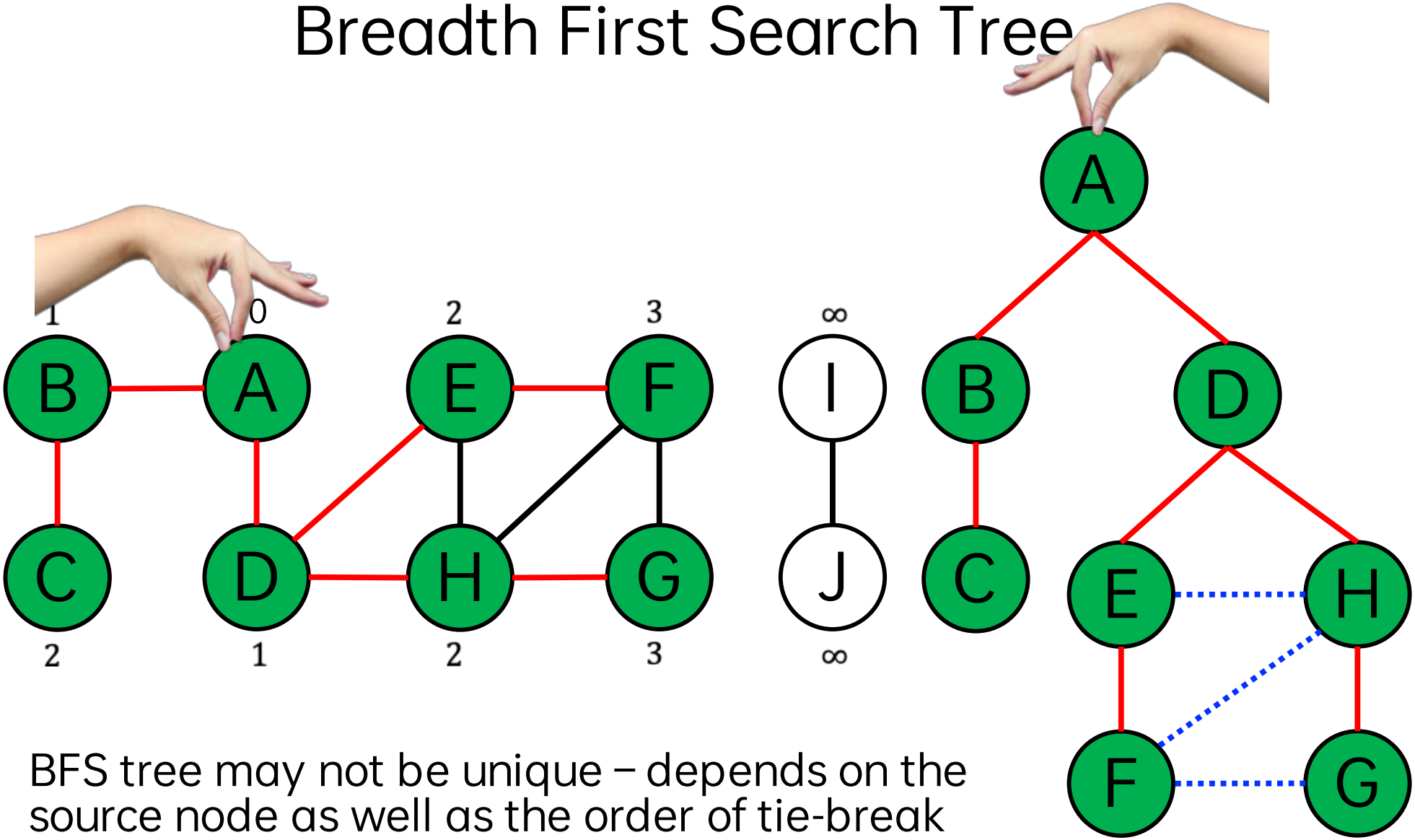

Not been there yet


Been there, haven't explored all the paths out


Been there, have explored all the paths out

- One observation is that unreachable vertices are not found by BFS
- In fact, this is true for any traversal algorithm

Breadth First Search Tree



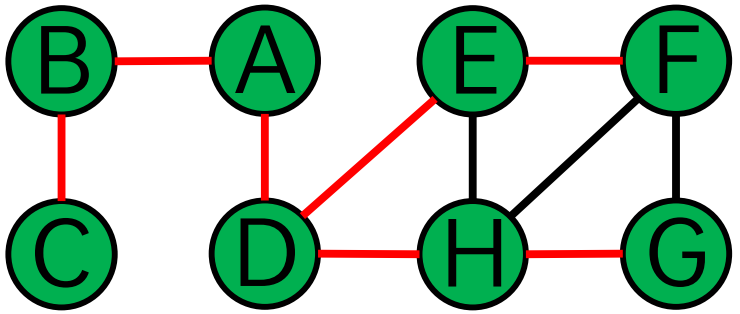
- BFS tree may not be unique – depends on the source node as well as the order of tie-break
- However, for the same source, the distances of each node from the source remain the same

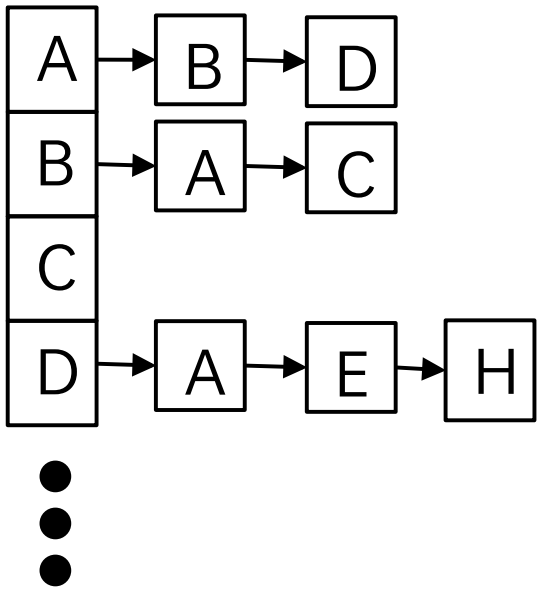
Breadth First Search - Pseudocode

- Input graph $G = (V, E)$ is represented using adjacency list
- Several additional attributes to each vertex in the graph is used
  - $u.color$ stores color of each vertex $u \in V$
  - $u.\pi$ stores the parent or predecessor of u. If u has no parent (e.g., if $u = s$ or u has not been discovered yet), then $u.\pi = NIL$
  - $u.d$ stores the distance from the source s to vertex u
- The algorithm uses a first-in, first-out queue Q

Breadth First Search - Pseudocode

BFS(G,s)
  for each vertex $u \in G.V - \{s\}$
    $u.color = WHITE,\ u.d = \infty,\ u.\pi = NIL$
  $s.color = ORANGE,\ s.d = 0,\ s.\pi = NIL$
  $Q = \phi$
  ENQUEUE(Q, s)
  while $Q \neq \phi$
    $u = DEQUE(Q)$
    for each $v \in G.Adj[u]$
      if $v.color == WHITE$
        ENQUEUE(Q, v)
        $v.color = ORANGE$
        $v.d = u.d + 1$
        $v.\pi = u$
    $u.color = GREEN$





Breadth First Search - Analysis

BFS(G,s)

for each vertex $u \in G.V - \{s\}$
    $u.color = WHITE,\ u.d = \infty,\ u.\pi = NIL$ → $\Theta(V)$
$s.color = ORANGE,\ s.d = 0,\ s.\pi = NIL$
$Q = \phi$
ENQUEUE(Q, s) → $\Theta(1)$
while $Q \neq \phi$
    $u = DEQUE(Q)$ → $O(V)$
    for each $v \in G.Adj[u]$
        if $v.color == WHITE$
            ENQUEUE(Q, v)
            $v.color = ORANGE$
            $v.d = u.d + 1$
            $v.\pi = u$
    $u.color = GREEN$

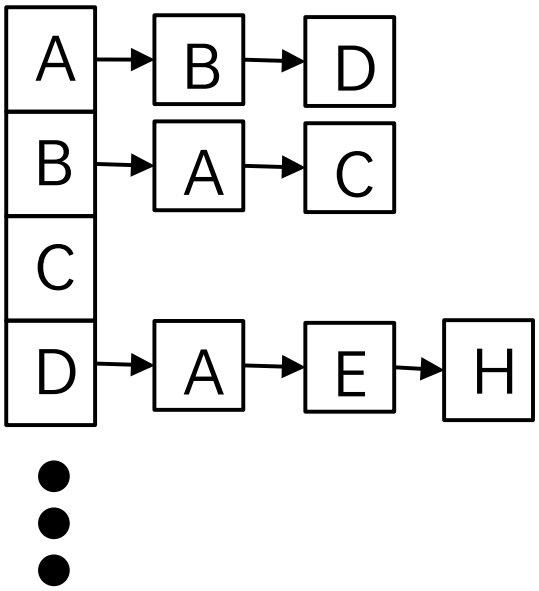
Overall, each vertex can be dequeued at most once

This line will be executed irrespective of whether the node is enqueued or not

Taking a look at the adjacency list, how many times, this gets executed?

$$O(2E) = O(E)$$

Overall asymptotic complexity is $O(V + E)$



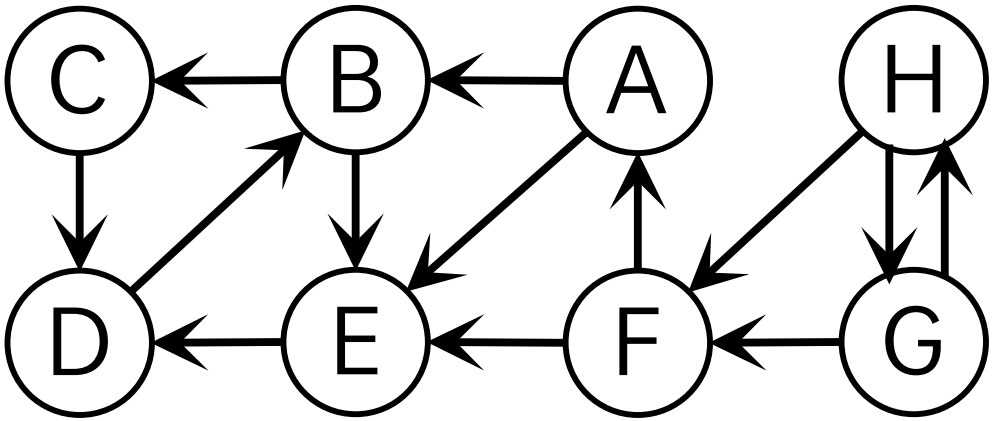
Depth First Search (DFS)

- Search "deeper" in the graph whenever possible
- It explores edges out of the most recently discovered vertex v that still has unexplored edges going out
- Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered
- The process continues until all reachable vertex are discovered
- If any undiscovered vertices remain, DFS selects one of them as a new source and repeats the search
- The algorithm repeats the entire process until it discovers every vertex
- Note that unlike BFS, here source is not given. Source is chosen randomly until all nodes are discovered

- DFS like BFS works for both directed and undirected graphs
- However, we will walk through an example on directed graph this time

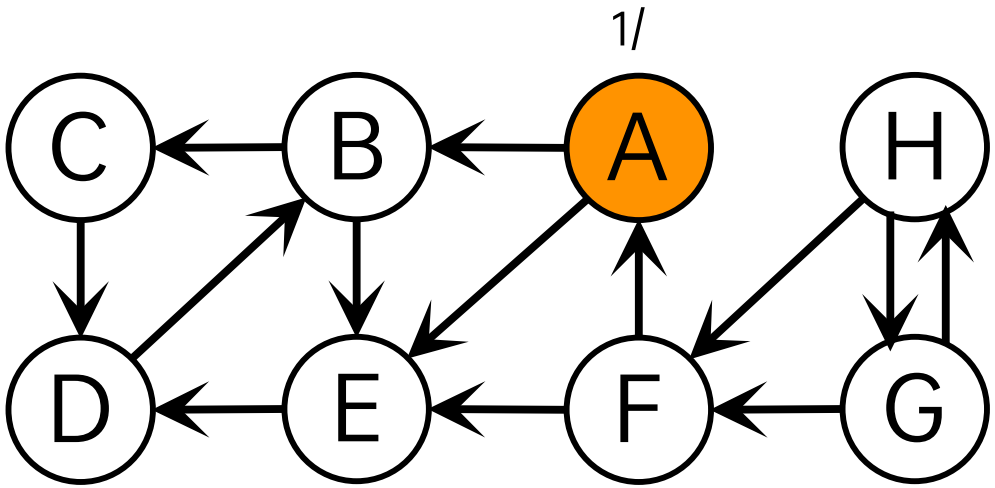
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



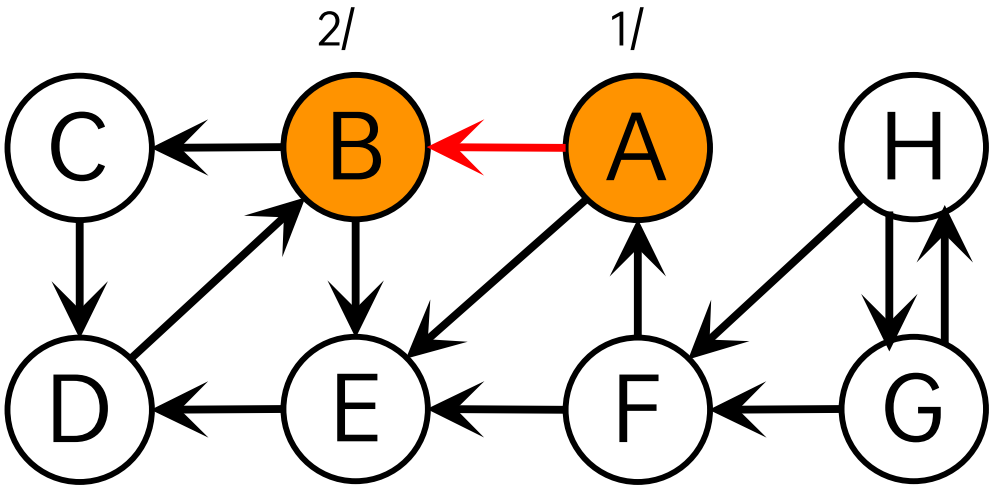
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



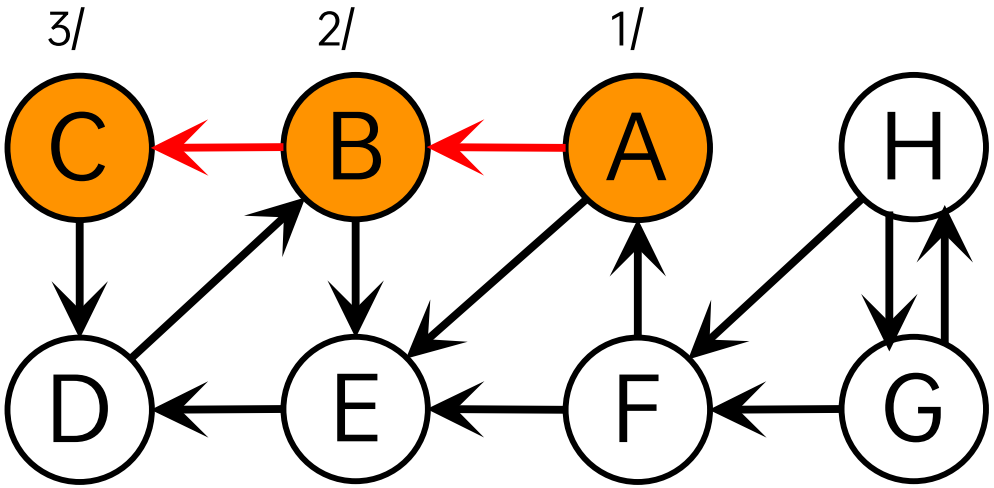
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



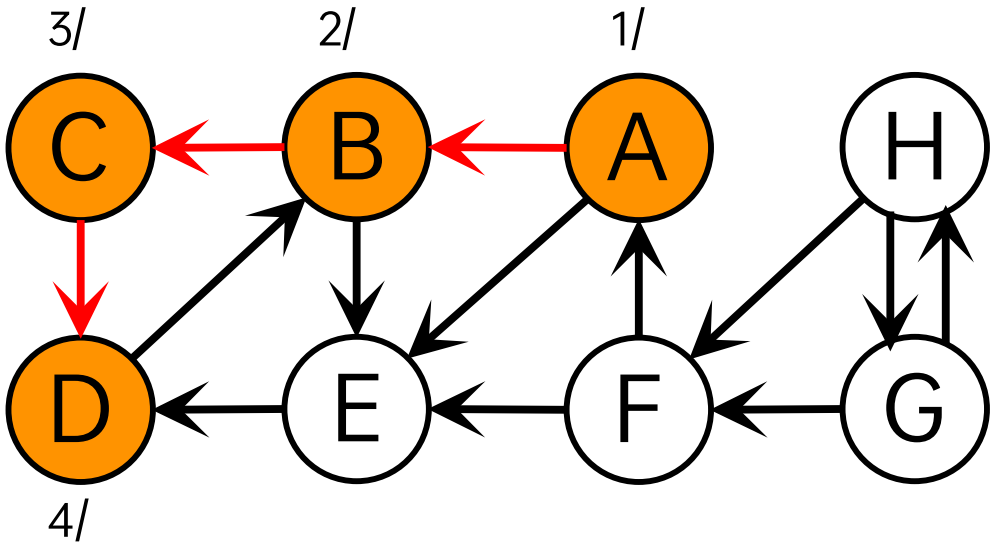
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



Depth First Search

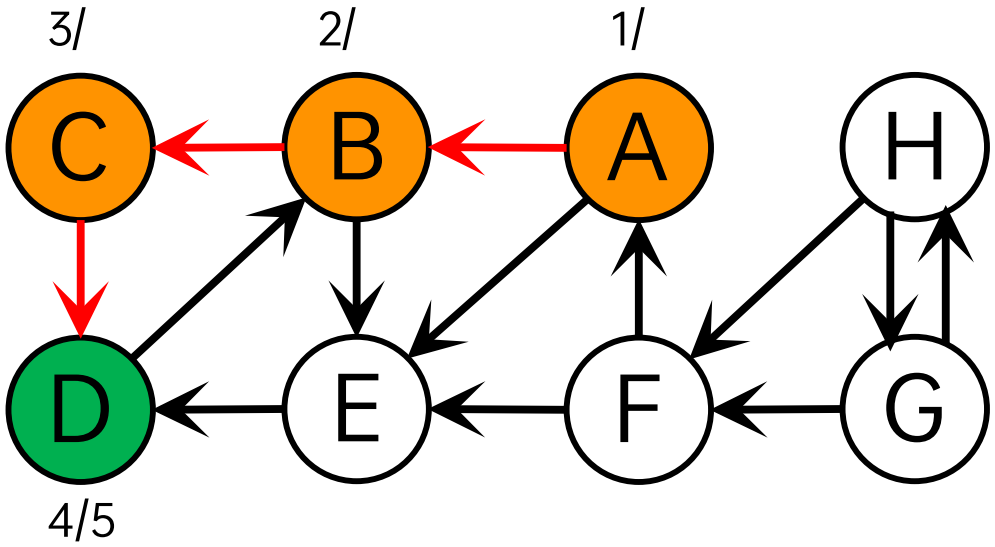
- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



- We will not go to B as that will make a cycle
  - Rule is – if an edge takes you back to a "discovered but not finished" node then we will say that's a backwards edge (or simply a back edge)

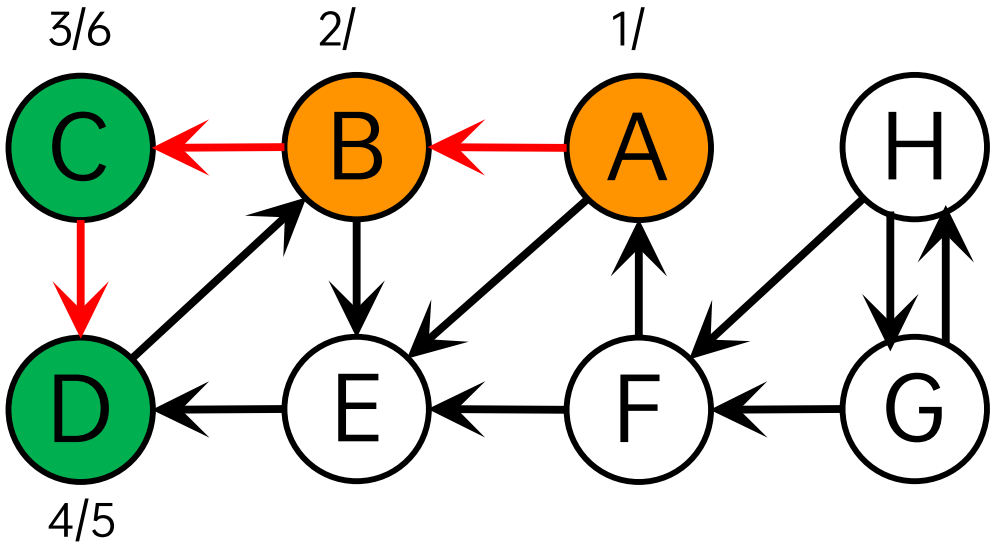
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



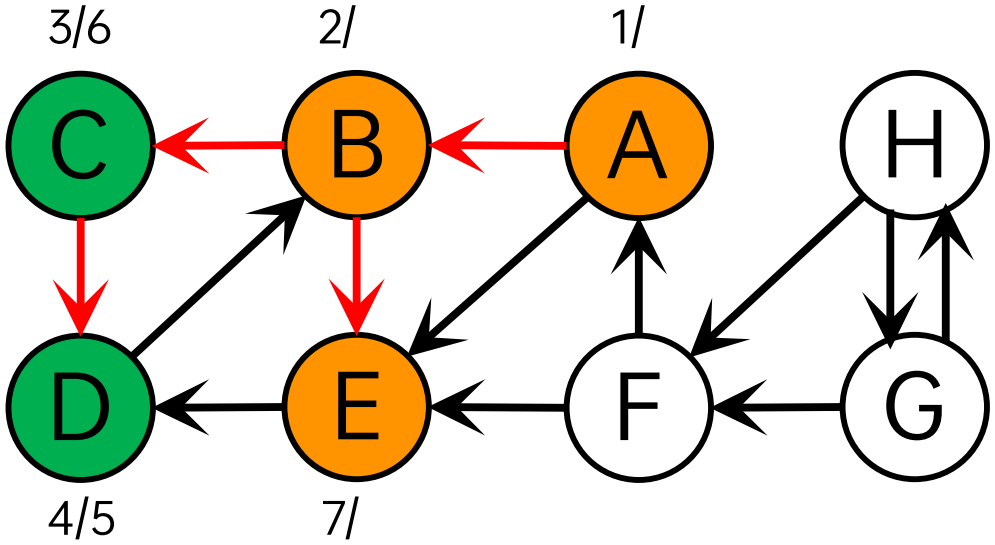
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



Depth First Search

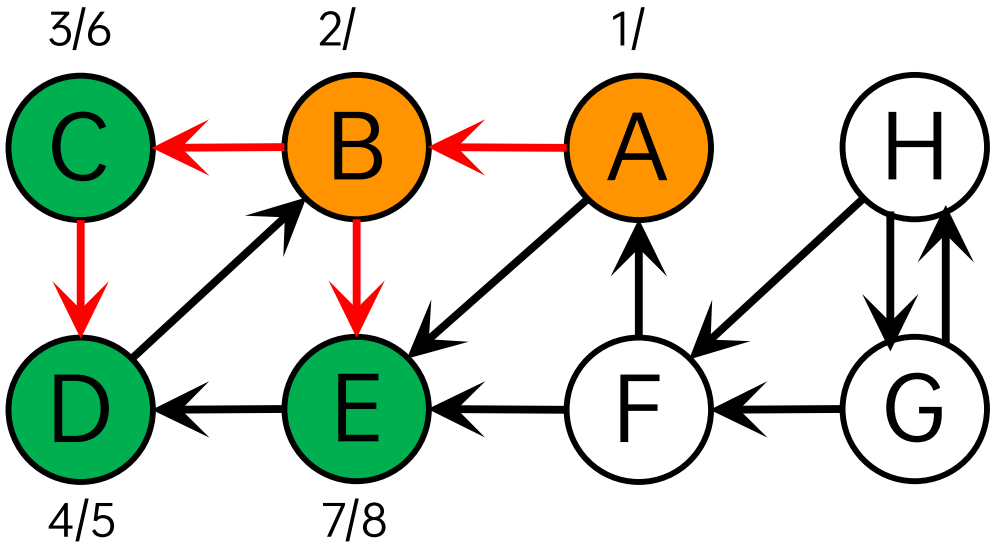
- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



- However, edge (E, D) is not a back edge
  - It does not get us back to an "unfinished" node, rather takes us to an already finished node – meaning no cycle

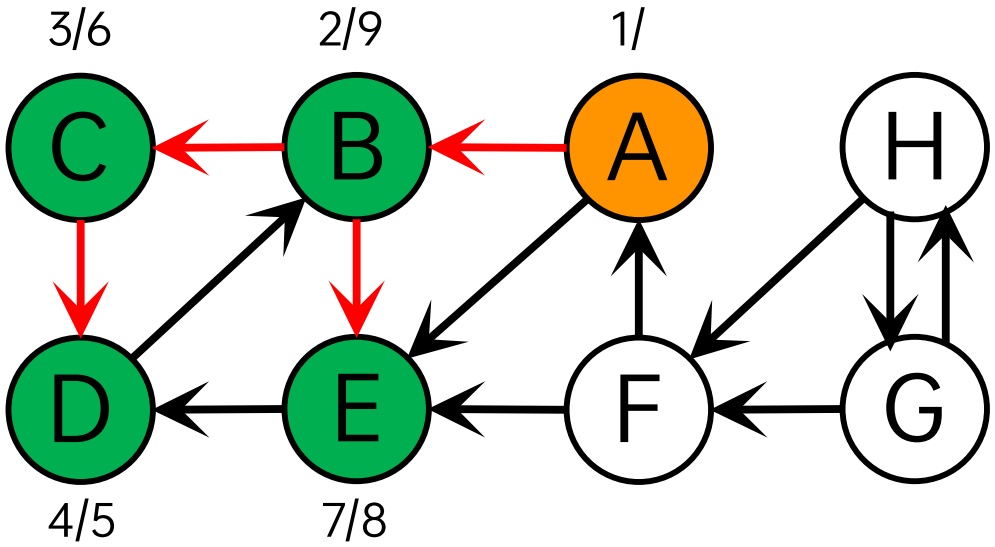
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



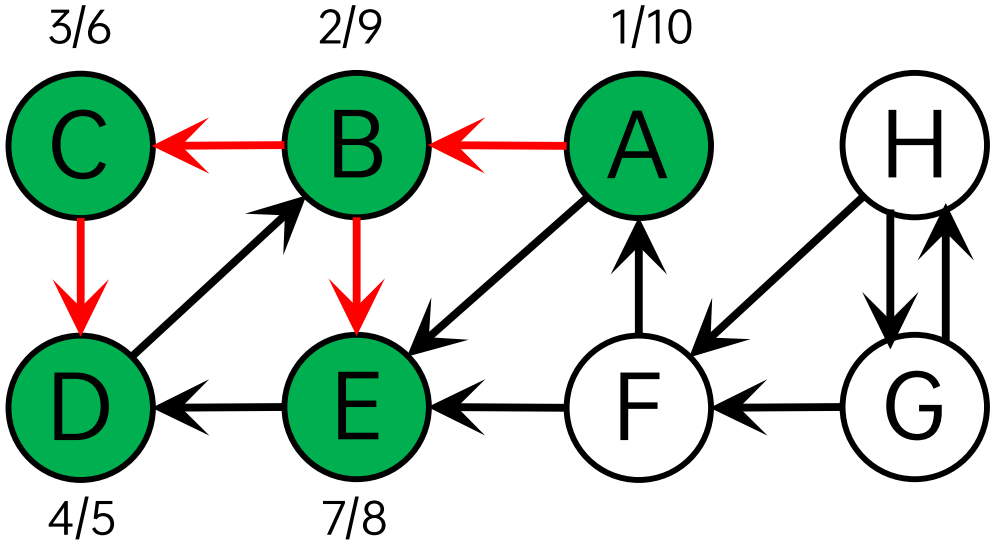
Depth First Search

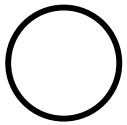
- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited

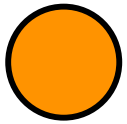


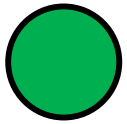
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited




Not been there yet

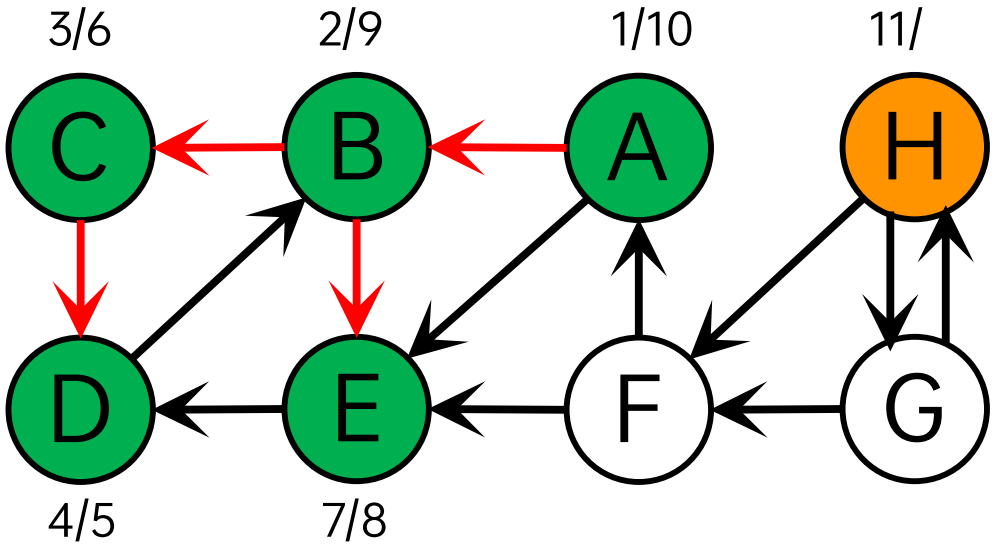

Been there, haven't explored all the paths out


Been there, have explored all the paths out

- The fact that we always go back to the last node is consistent with the behavior of a stack

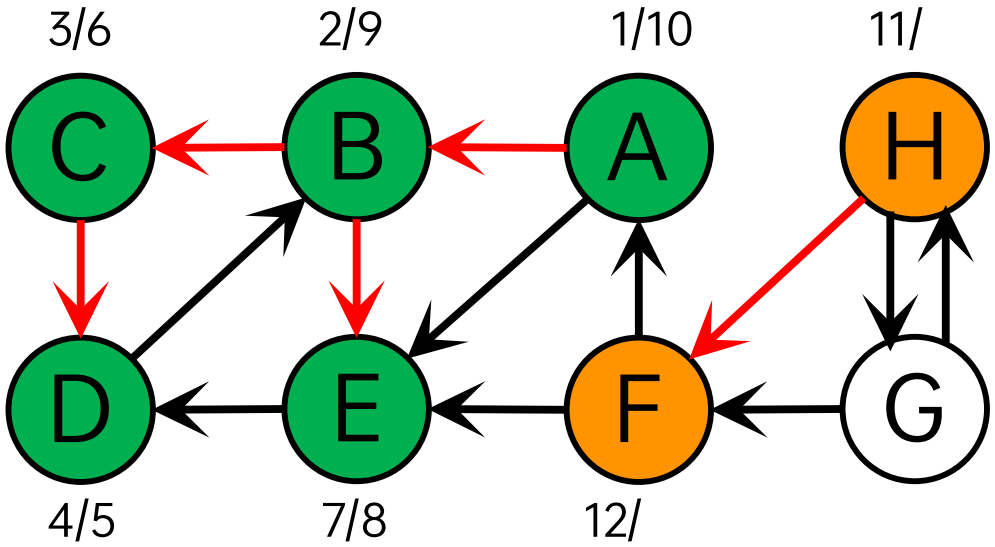
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



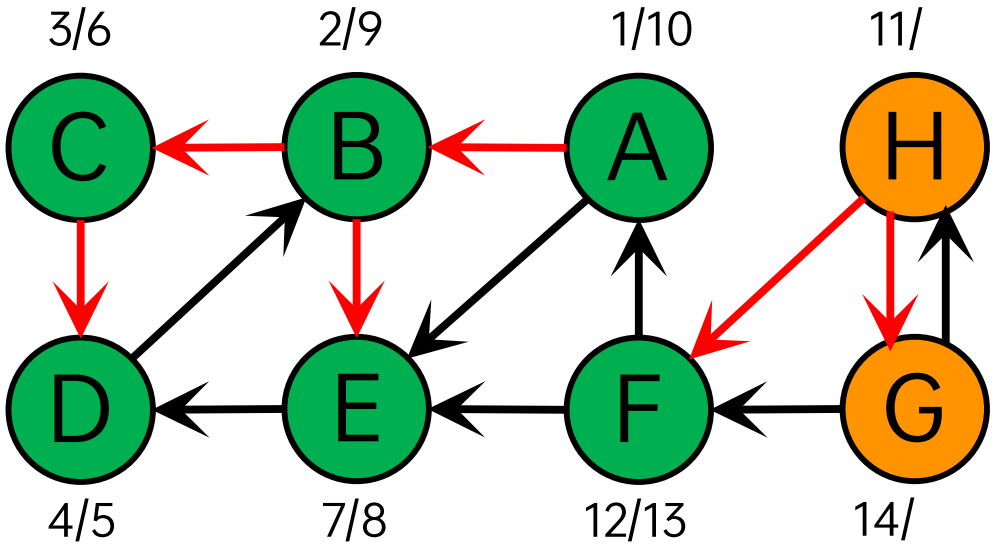
Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

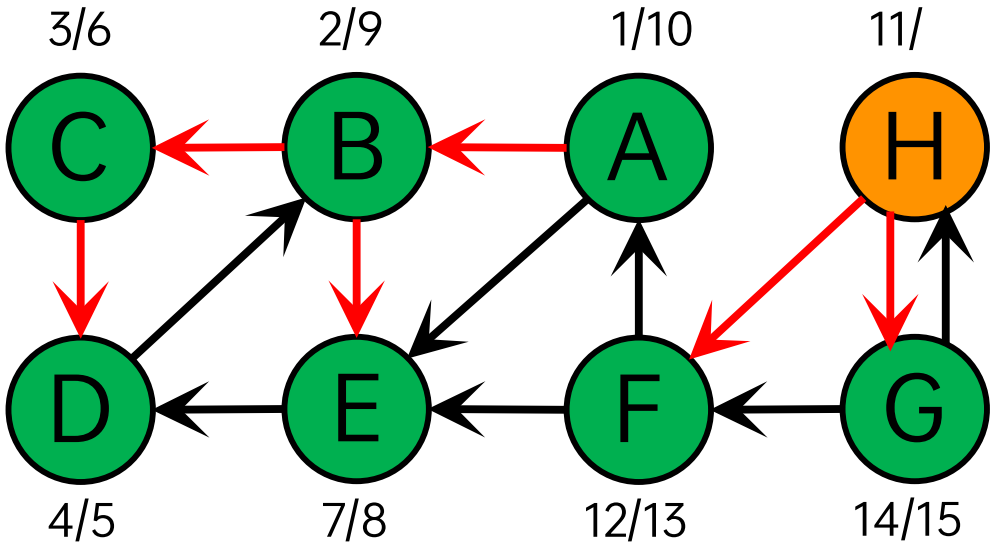
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



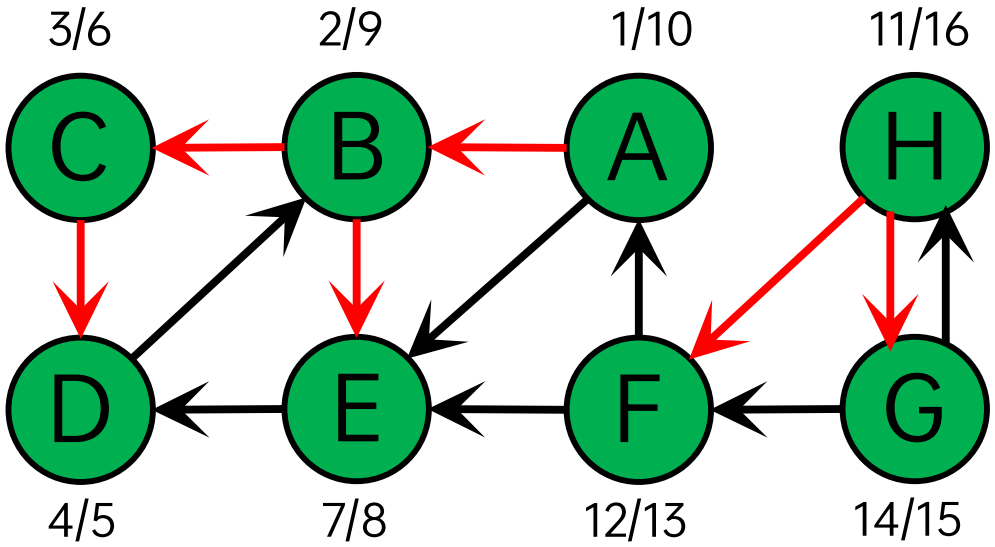
Depth First Search

- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



Depth First Search

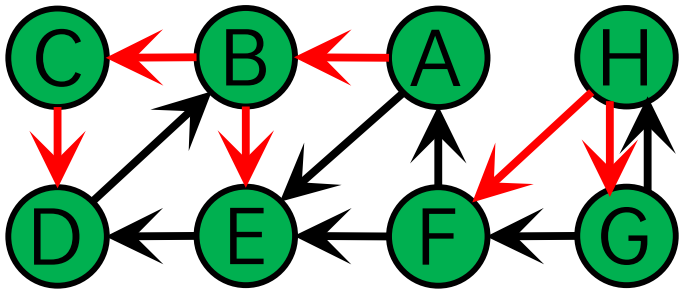
- Each vertex maintains two timestamps
  - start time: when vertex is discovered
  - finish time: When all nodes in adjacency list is visited



- Of course, DFS is not unique (as source is not given)
- Can you identify a node as first the very first source that would have explored all nodes without going for another source node?

Depth First Search - Pseudocode

```
DFS(G)
    for each vertex u ∈ G.V
        u.color = WHITE, u.π = NIL
    time = 0 // Global variable
    // Go through all vertices to find
    // a source
    for each vertex u ∈ G.V
        if u.color == WHITE
            EXPLORE(G, u)
```




```
EXPLORE(G, s)
    s.color = orange
    time++
    s.d = time // Discover time
    for each vertex v ∈ G.Adj[s]
        if v.color == WHITE
            EXPLORE(G, v)
            v.π = s
    // After all neighbors of s is visited
    s.color = GREEN
    time++
    s.f = time // finish time
```

Depth First Search - Analysis

DFS(G)
  for each vertex $u \in G.V$
    $u.color = WHITE,\ u.\pi = NIL$ → $\Theta(V)$
  $time = 0$ // Global variable
  // Go through all vertices to find
  // a source
  for each vertex $u \in G.V$
    if $u.color == WHITE$ → Only this line?$\Theta(V)$
      EXPLORE(G, u) → Only this line?$O(V)$ Gets called equal to number of trees in the forest

Depth First Search - Pseudocode

```
EXPLORE(G, s)
    s.color = orange
    time++
    s.d = time // Discover time
    for each vertex v ∈ G.Adj[s]
        if v.color == WHITE
            EXPLORE(G, v)
            v.π = s
    // After all neighbors of s is visited
    s.color = GREEN
    time++
    s.f = time // finish time
```

$\Theta(V)$

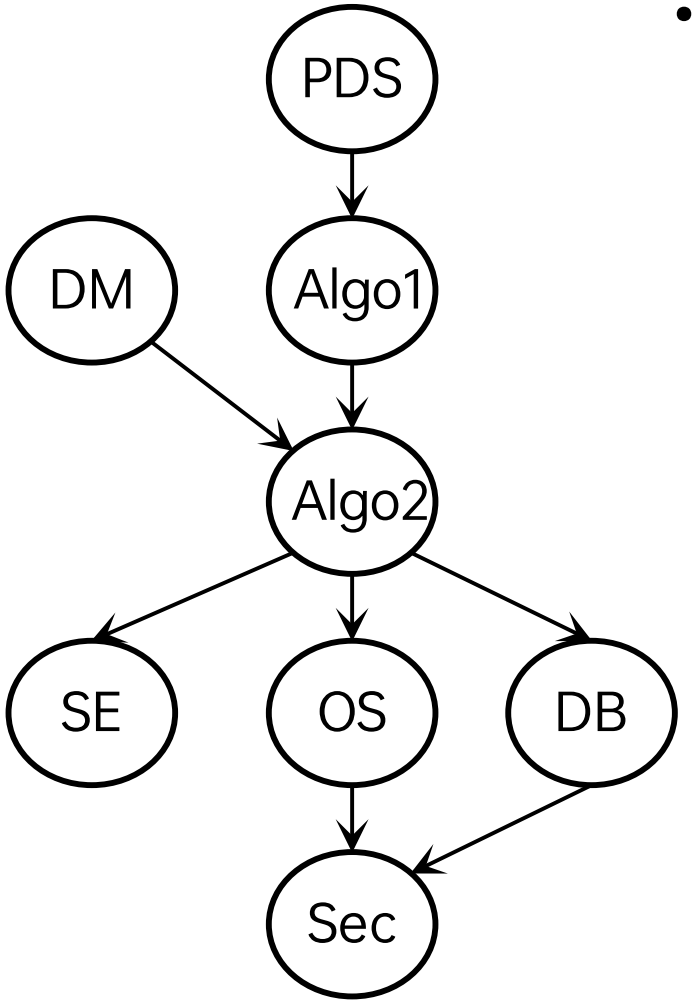
Analysis exactly like BFS

In aggregate, $\theta(E)$

Overall asymptotic complexity is $O(V + E)$

Topological Sort

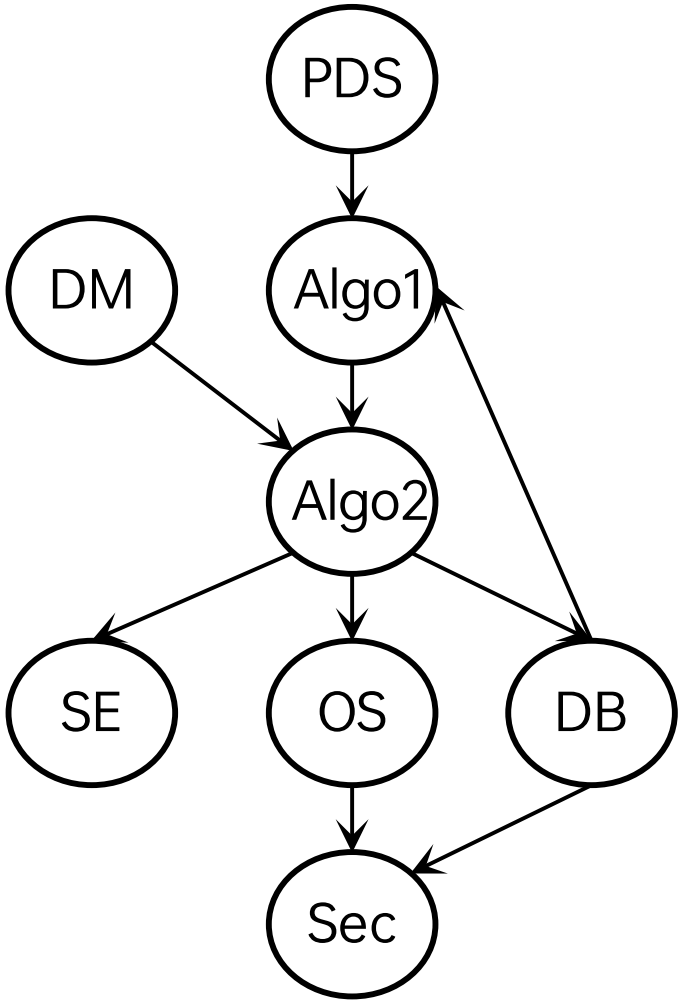
- Lets consider a dependency graph showing the dependency of different courses for a (part of a) CS curriculum



  - We need to come up with an ordering of the courses that satisfies the dependencies

PDS	PDS
Algo1	Algo1
DM	Algo2
Algo2	•
OS	•
SE	•
DB	
Sec	
✅	❌

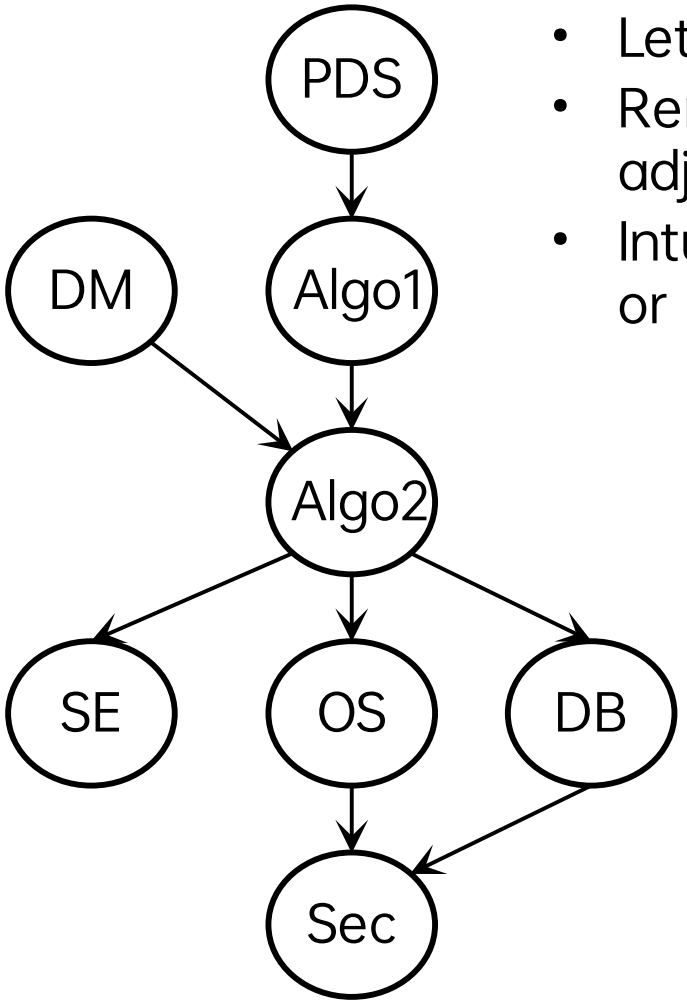
Topological Sort



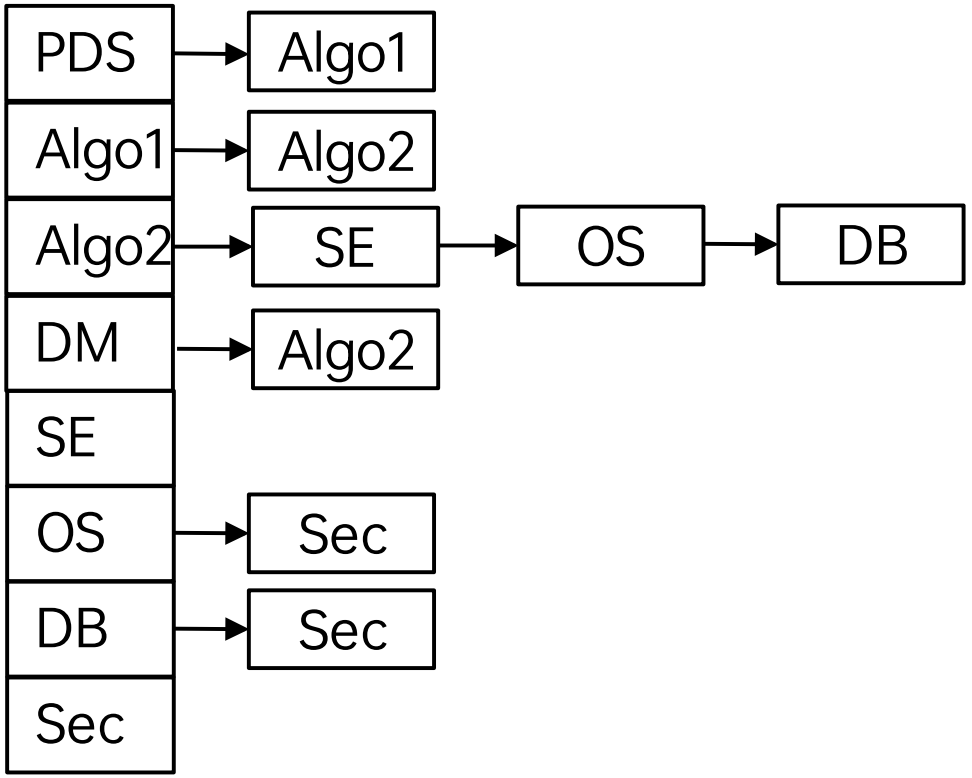
- A topological sort of a Directed Acyclic Graph (DAG) $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering

- If the graph contains a cycle, then no linear ordering is possible

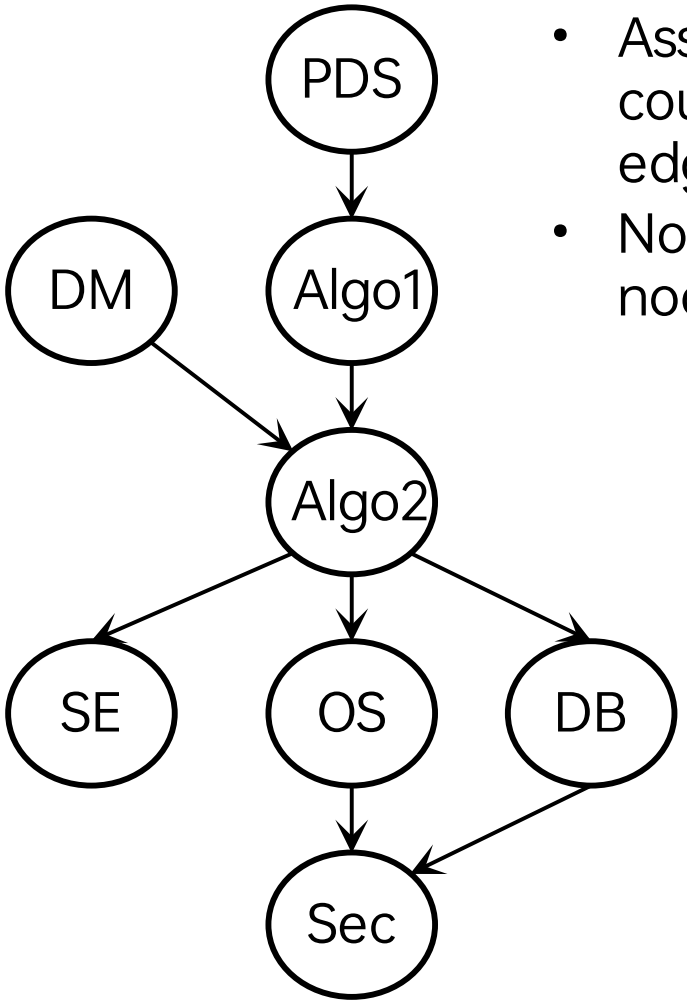
Topological Sort



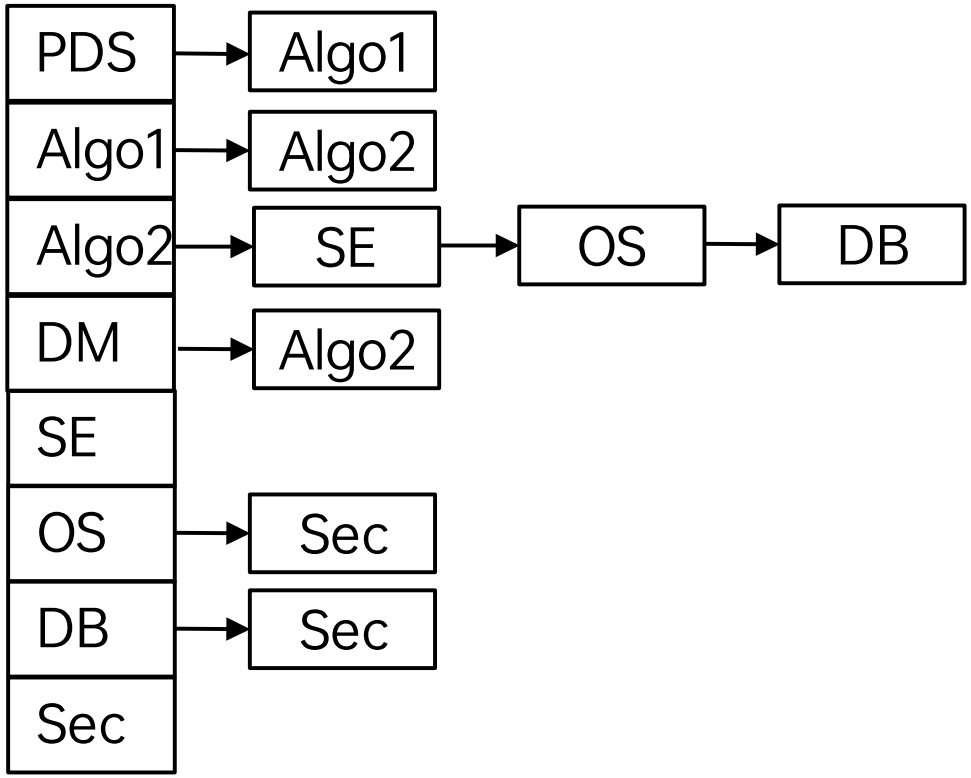
- Let us try to think of an algorithm
- Remember that the graph is represented as an adjacency list which is the input to the algorithm
- Intuitively, the ordering should start either with 'PDS' or 'DM'. How to find them from the adjacency list?



Topological Sort



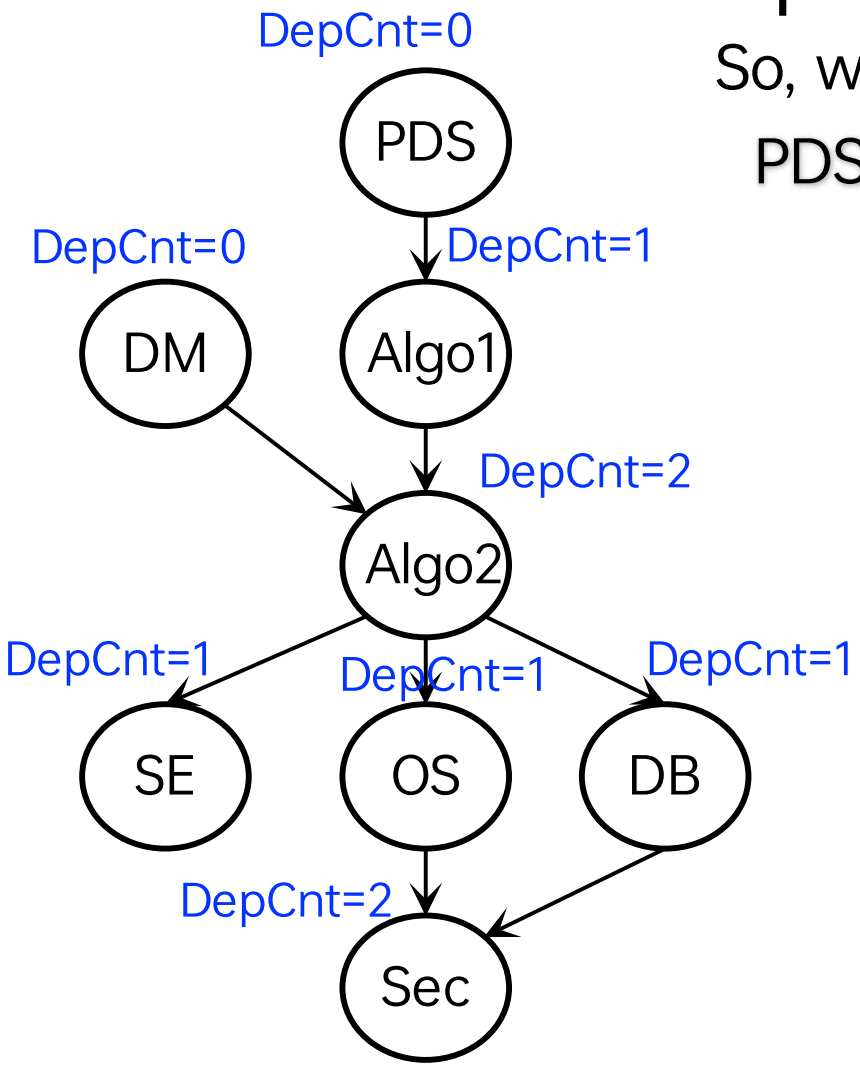
- Assign a counter to each node of the graph that counts, from the adjacency list, how many incoming edges are connected to each node
- Nodes with 0 incoming edges are potential start nodes

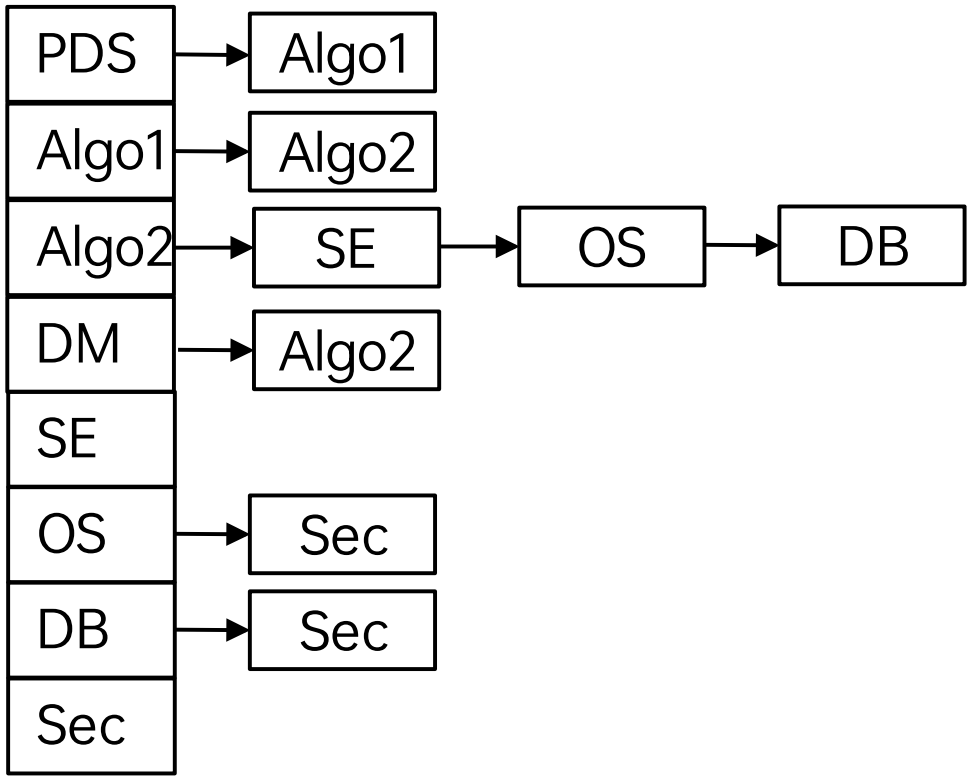


Topological Sort

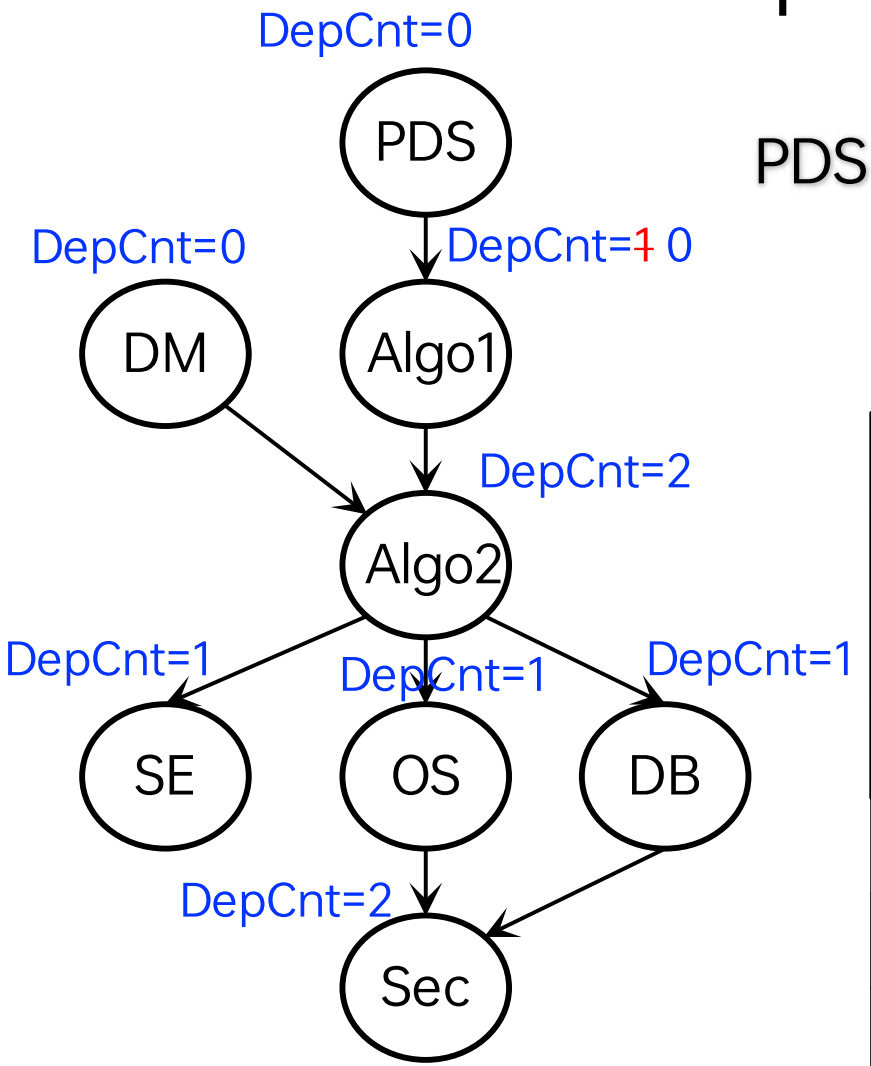
So, we start with say PDS

PDS

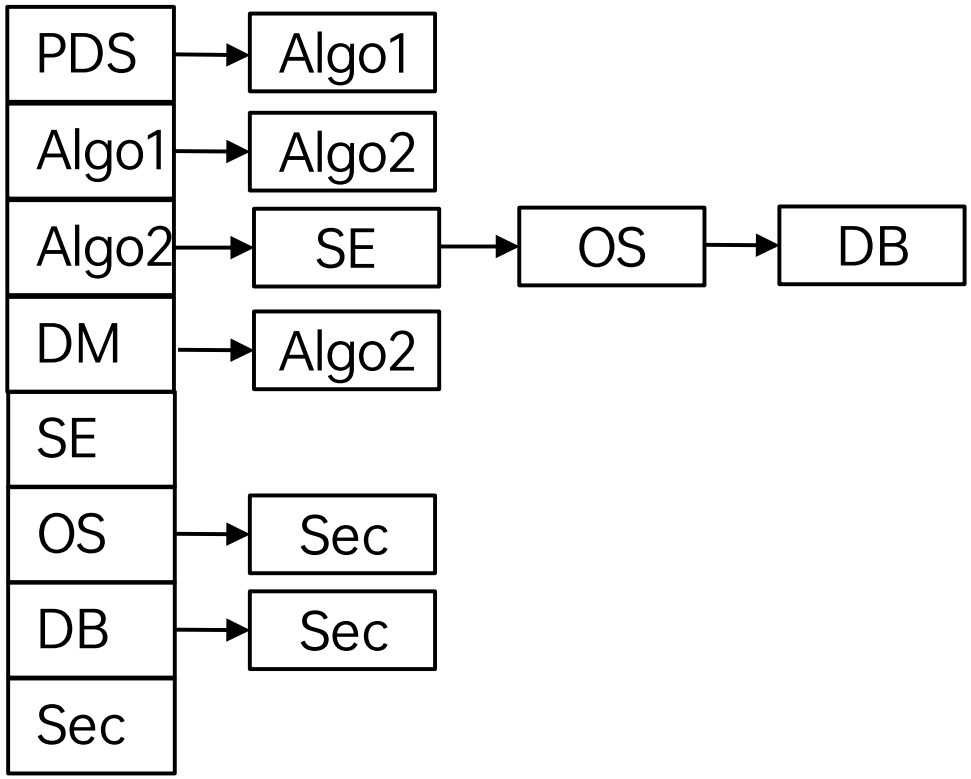




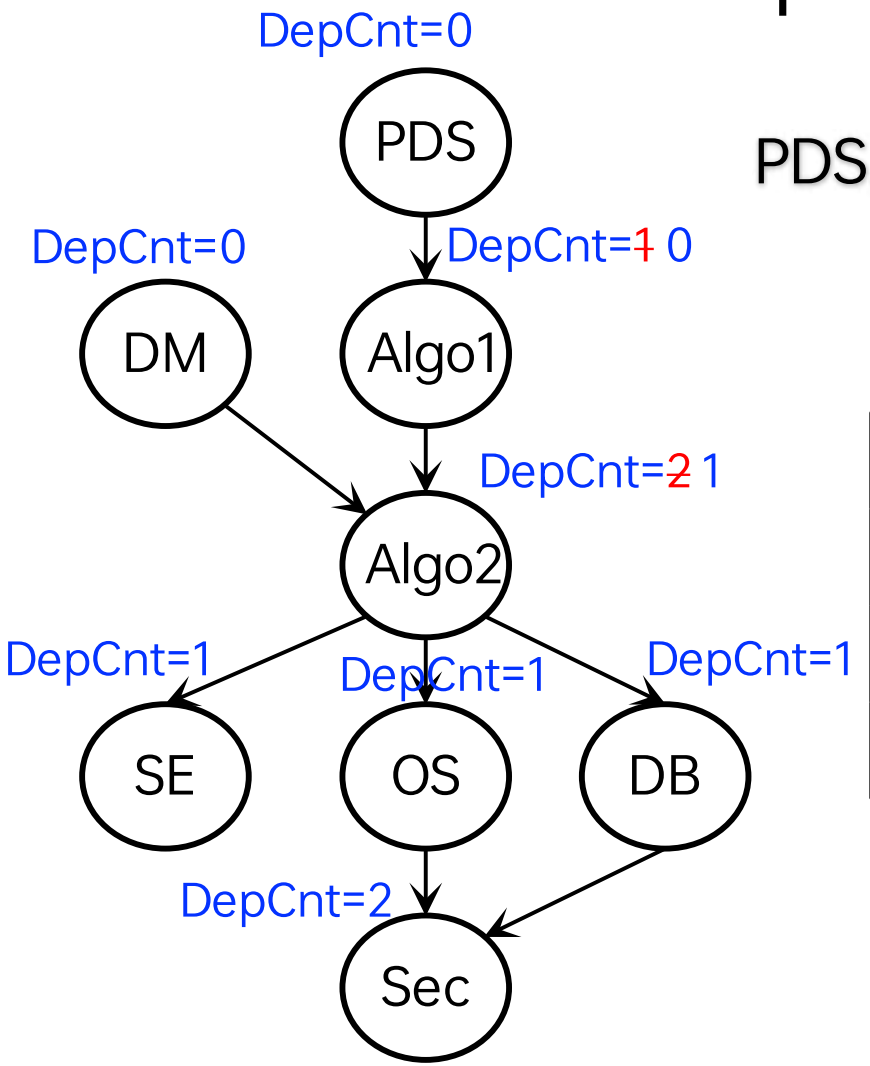
Topological Sort



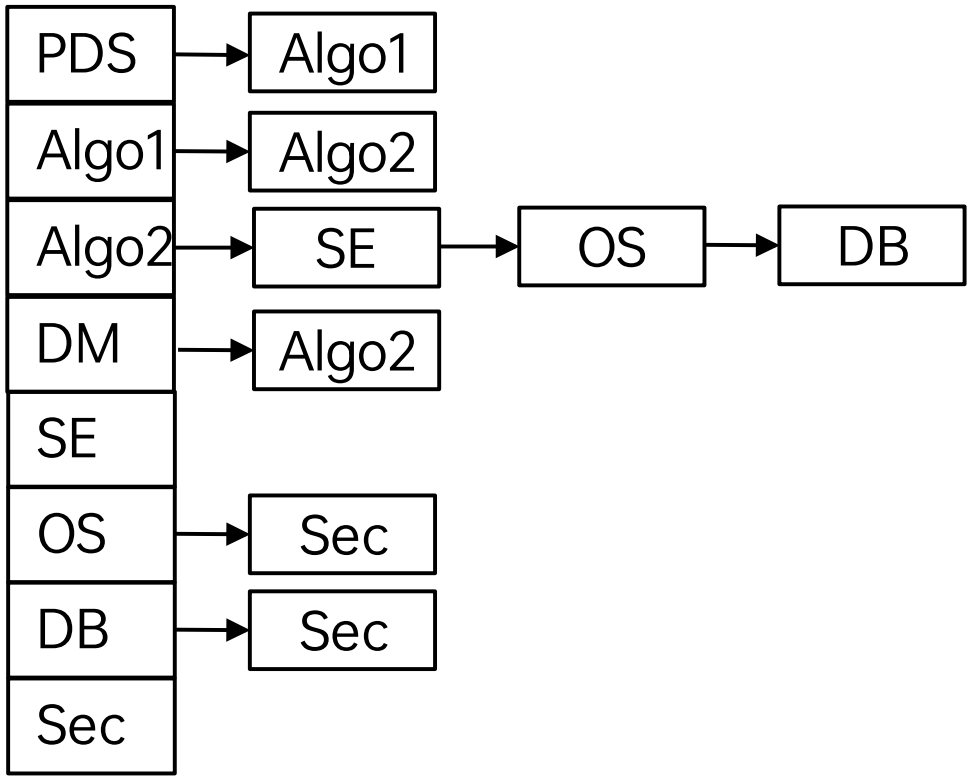
PDS



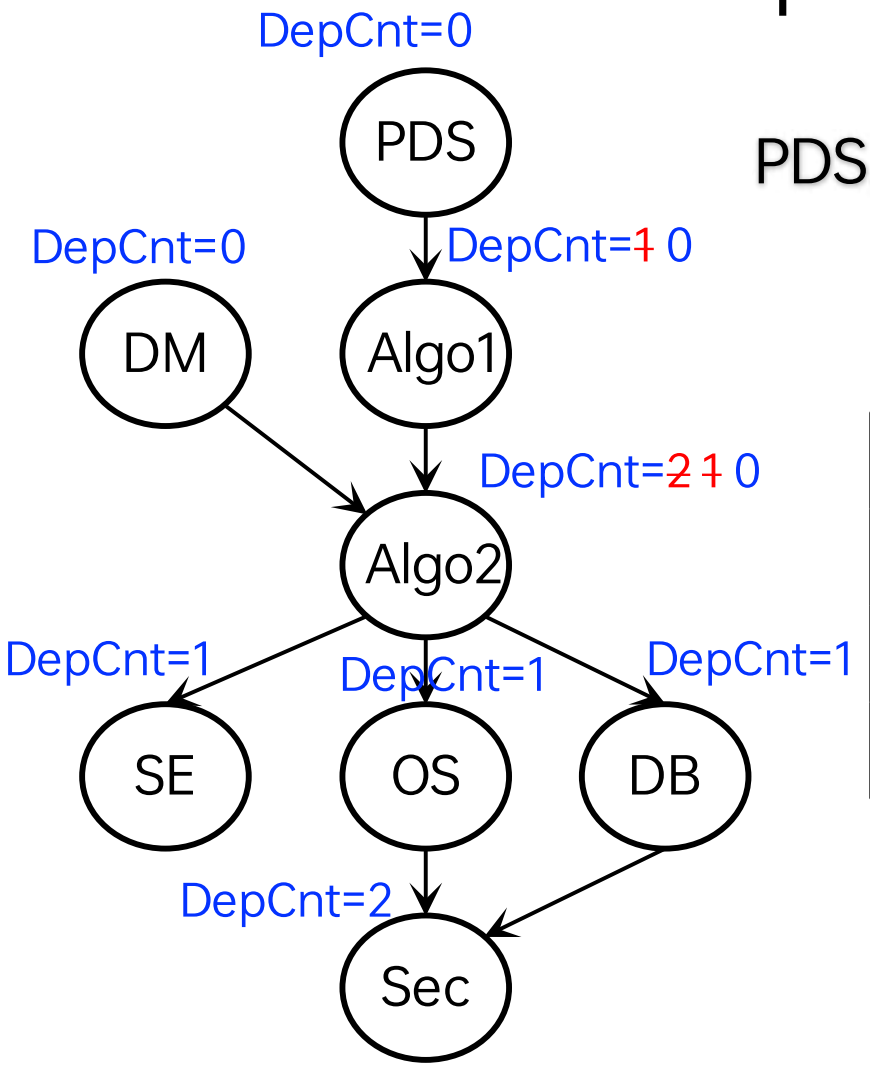
Topological Sort



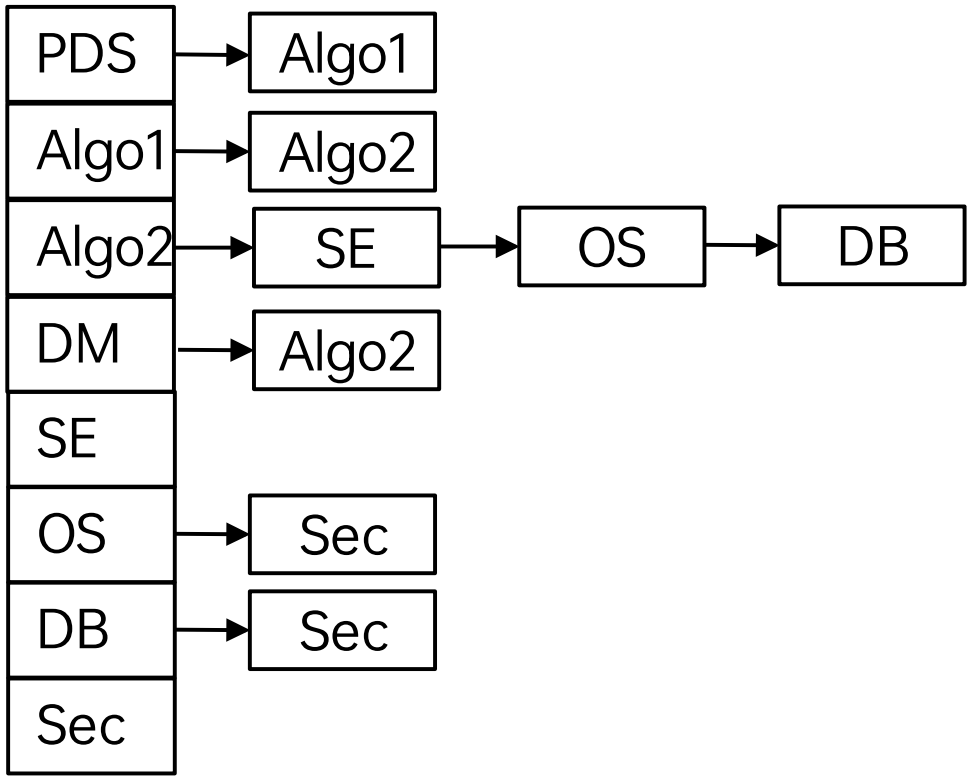
PDS, Algo1



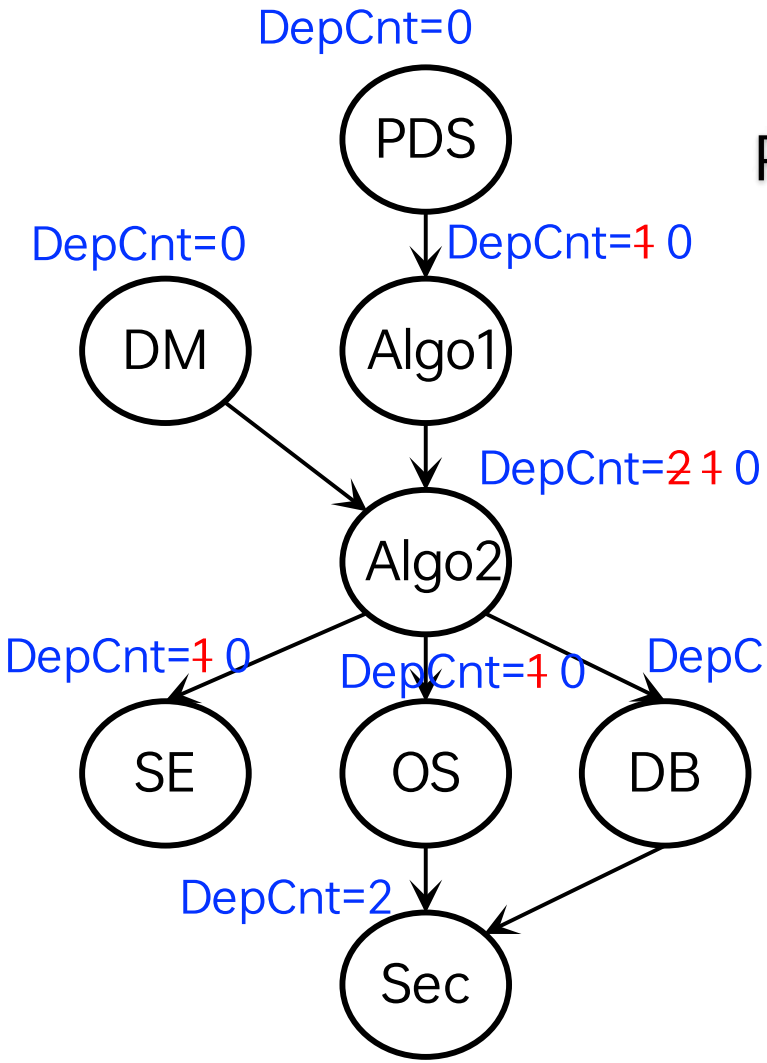
Topological Sort



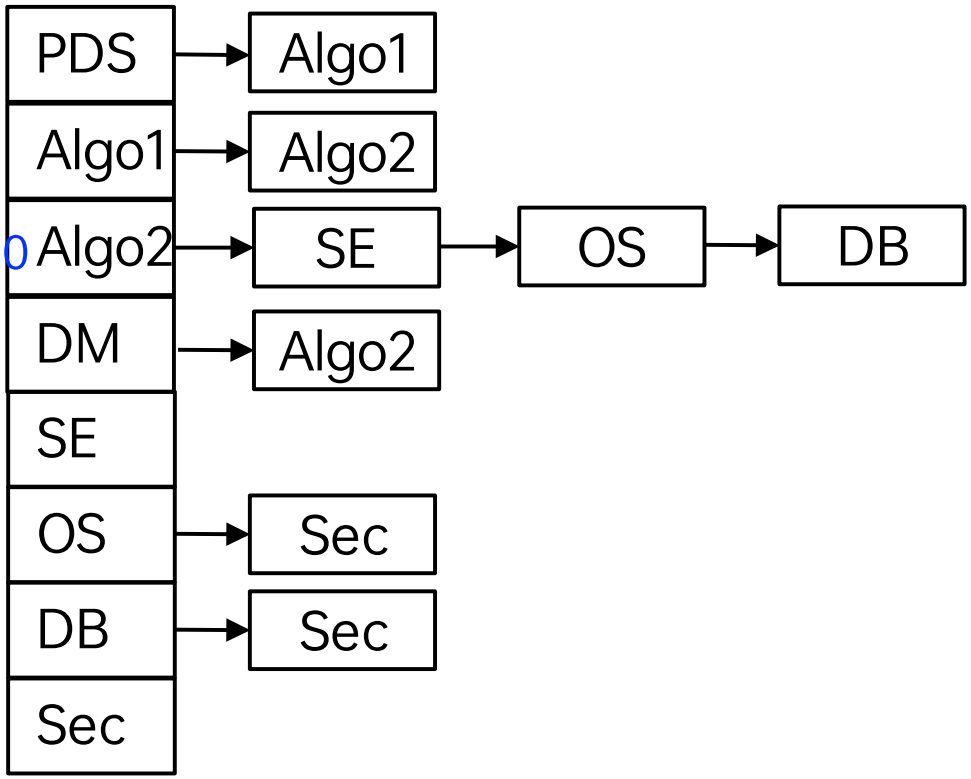
PDS, Algo1, DM



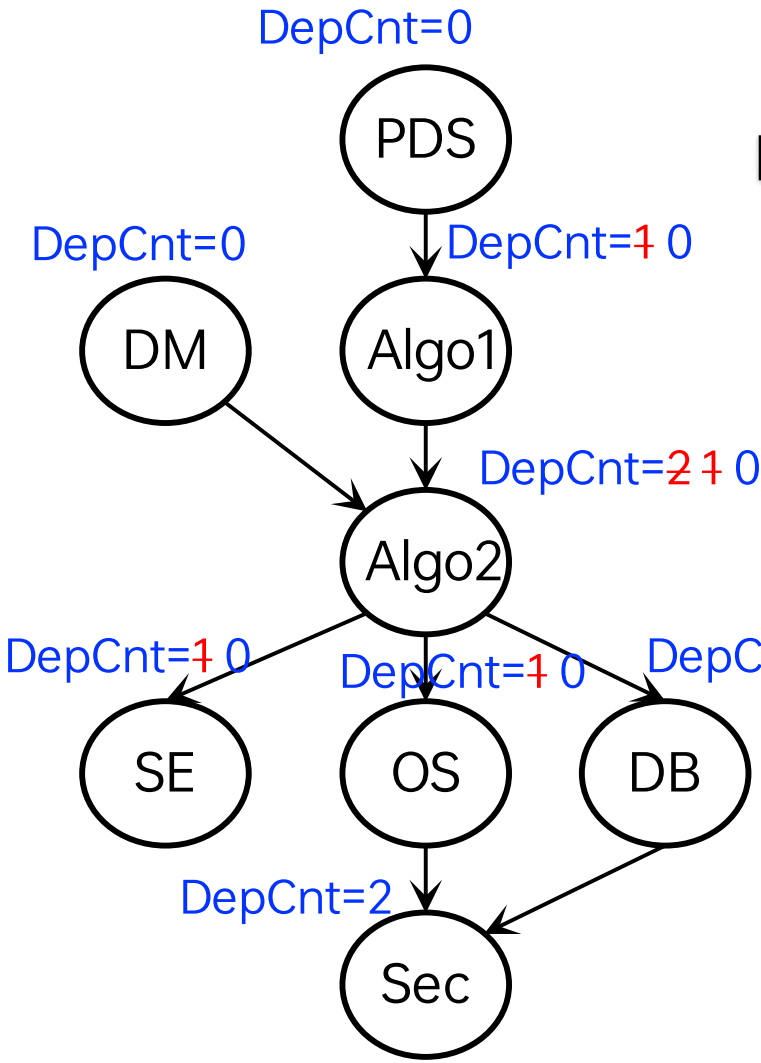
Topological Sort



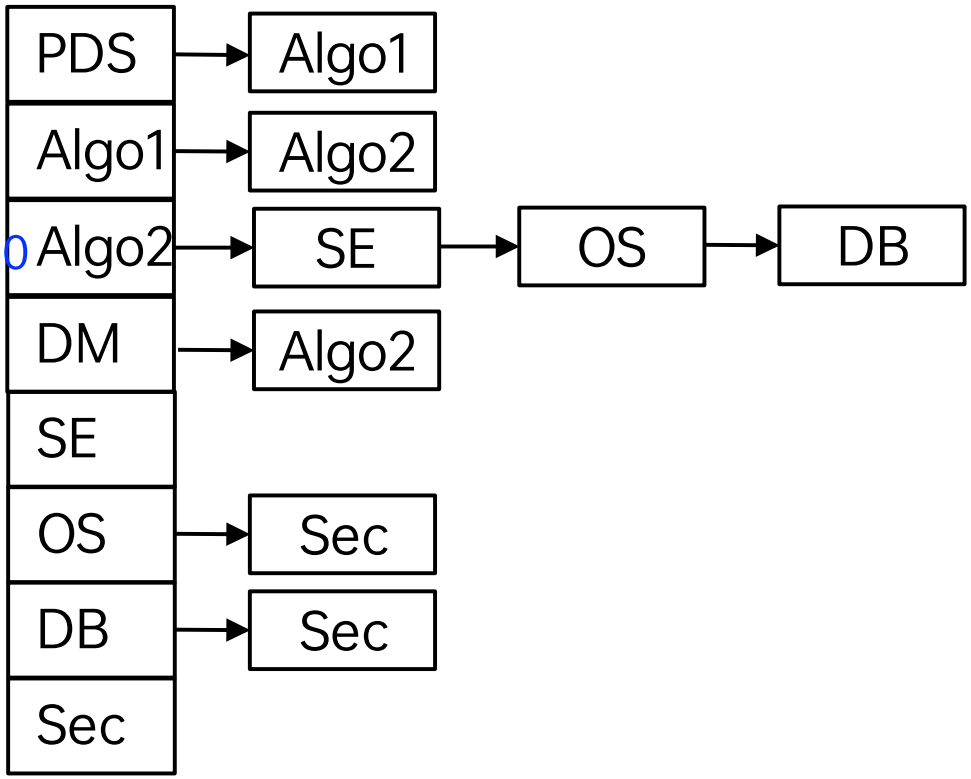
PDS, Algo1, DM, Algo2



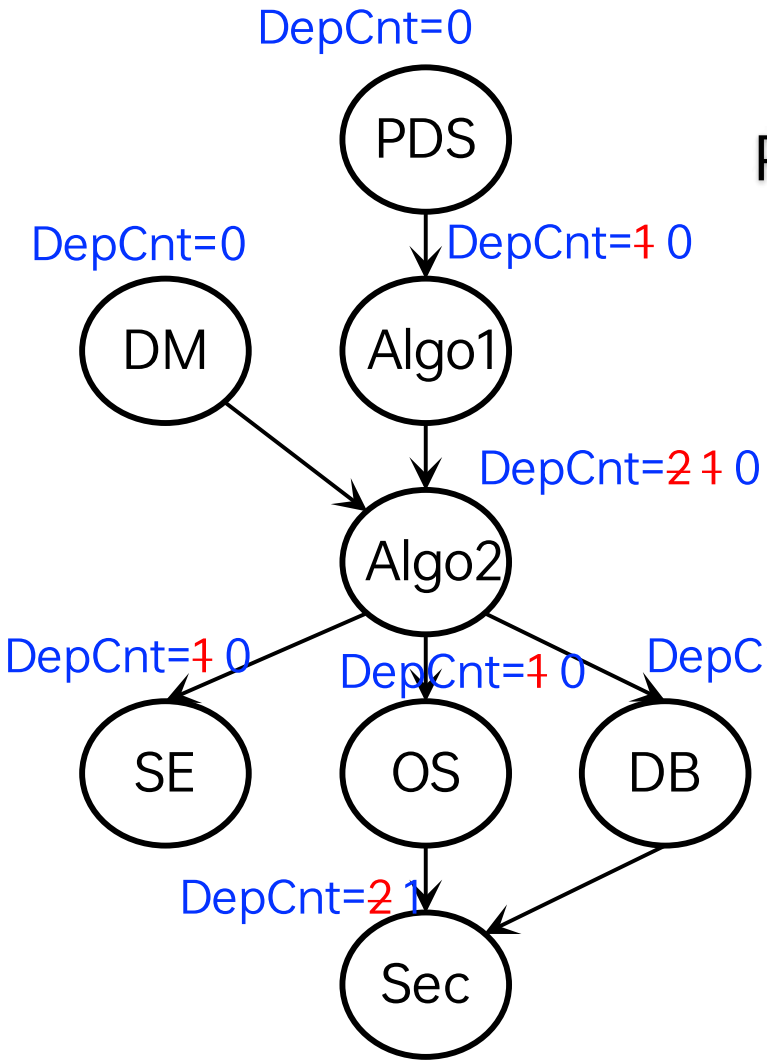
Topological Sort



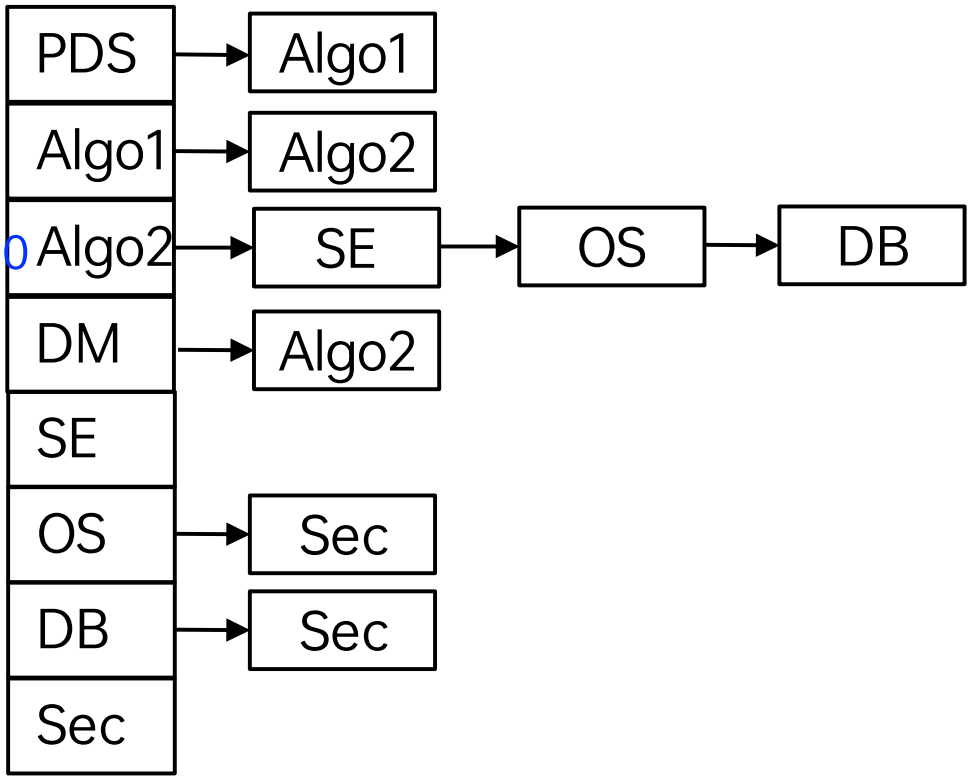
PDS, Algo1, DM, Algo2, SE



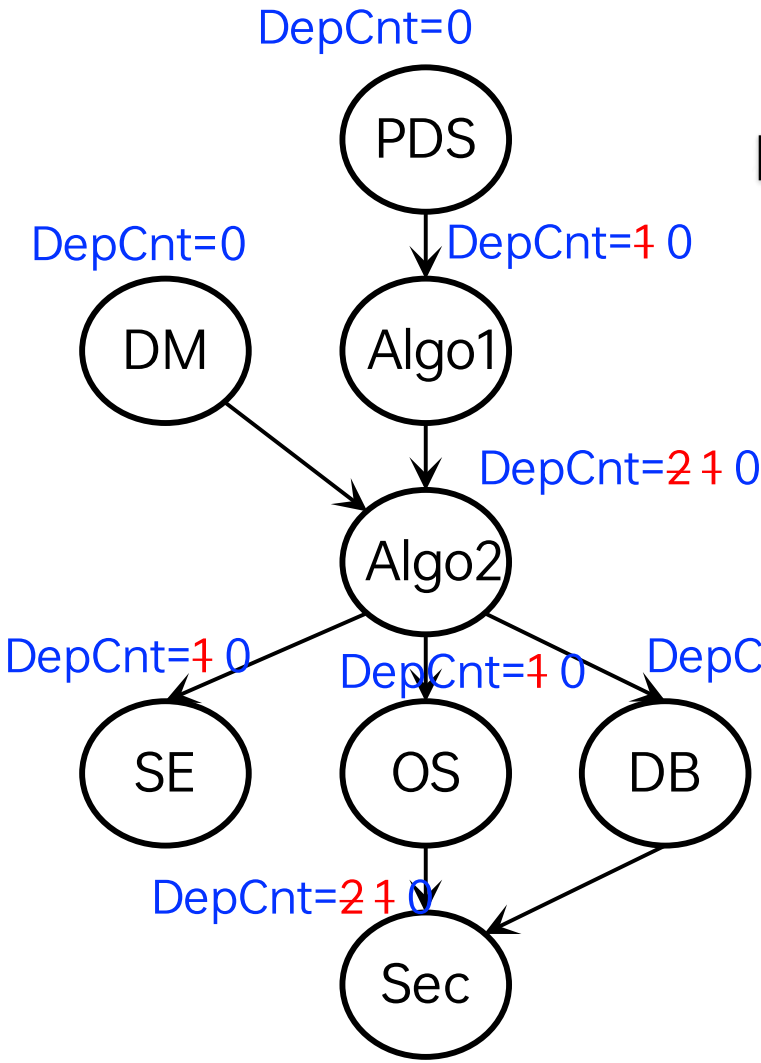
Topological Sort



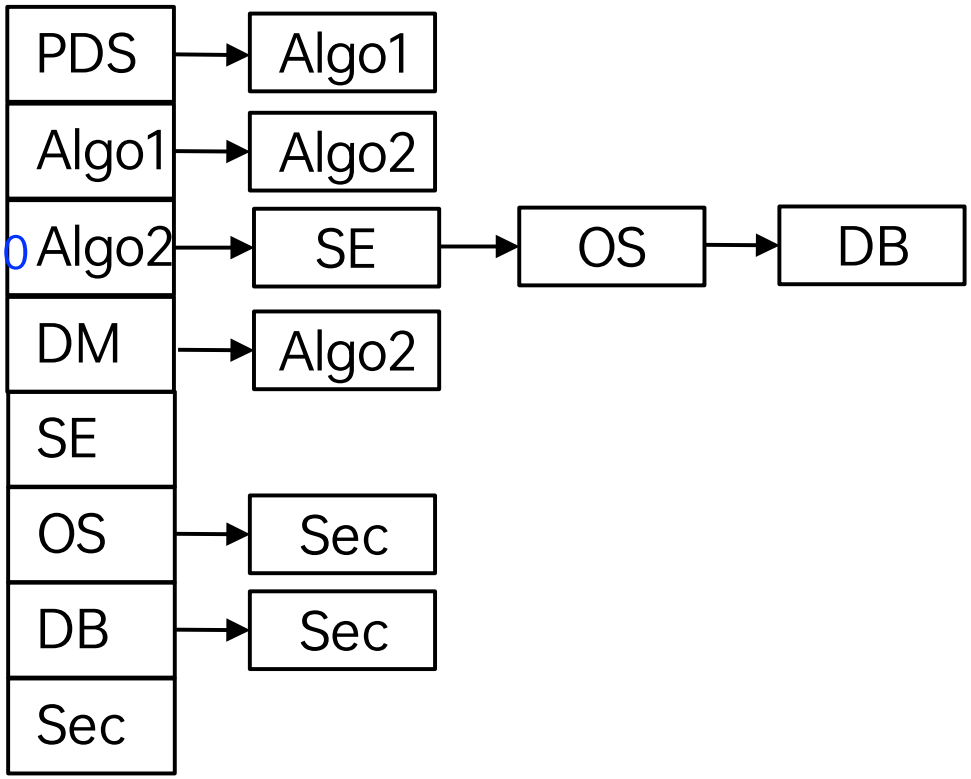
PDS, Algo1, DM, Algo2, SE, OS



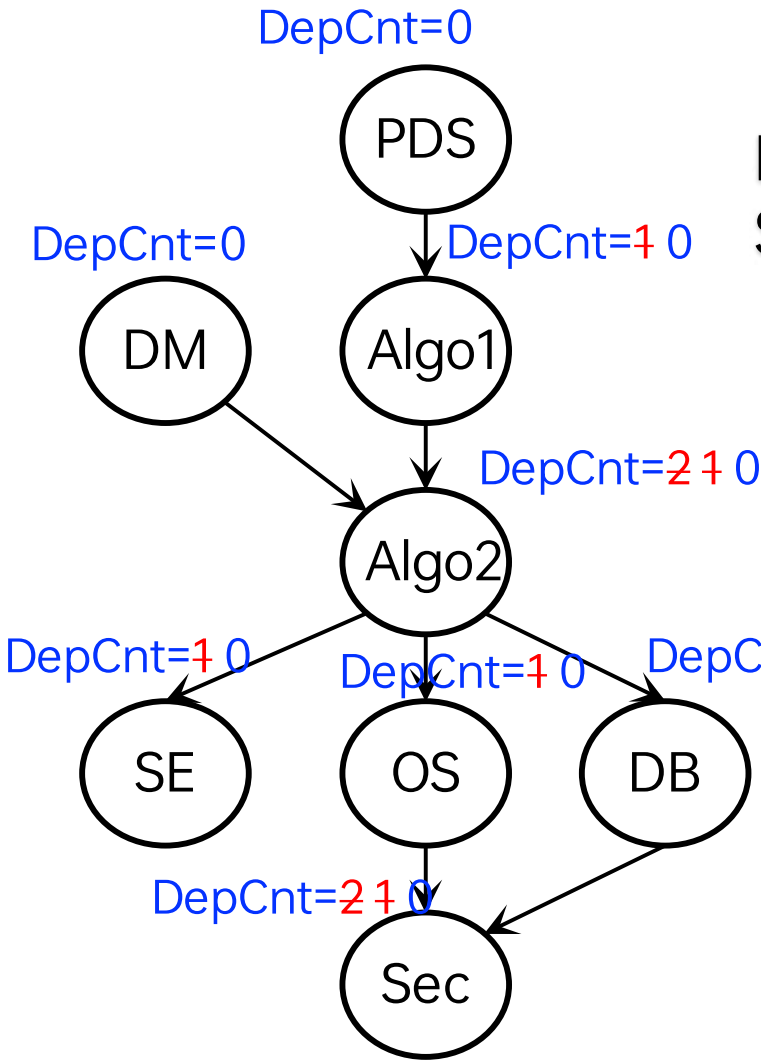
Topological Sort



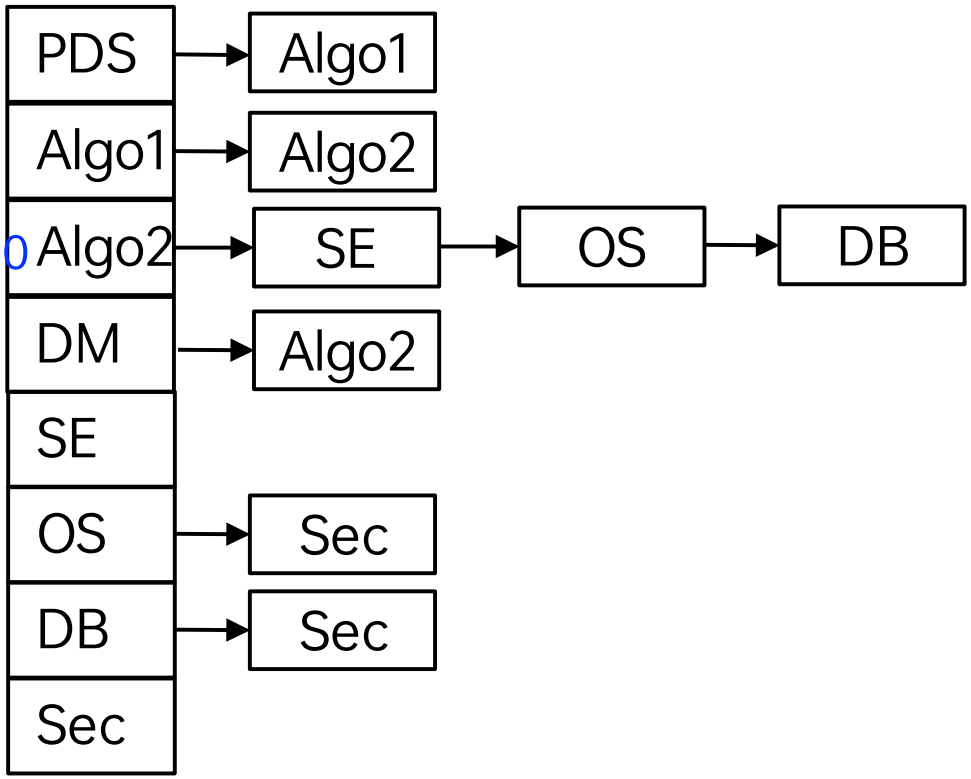
PDS, Algo1, DM, Algo2, SE, OS, DB



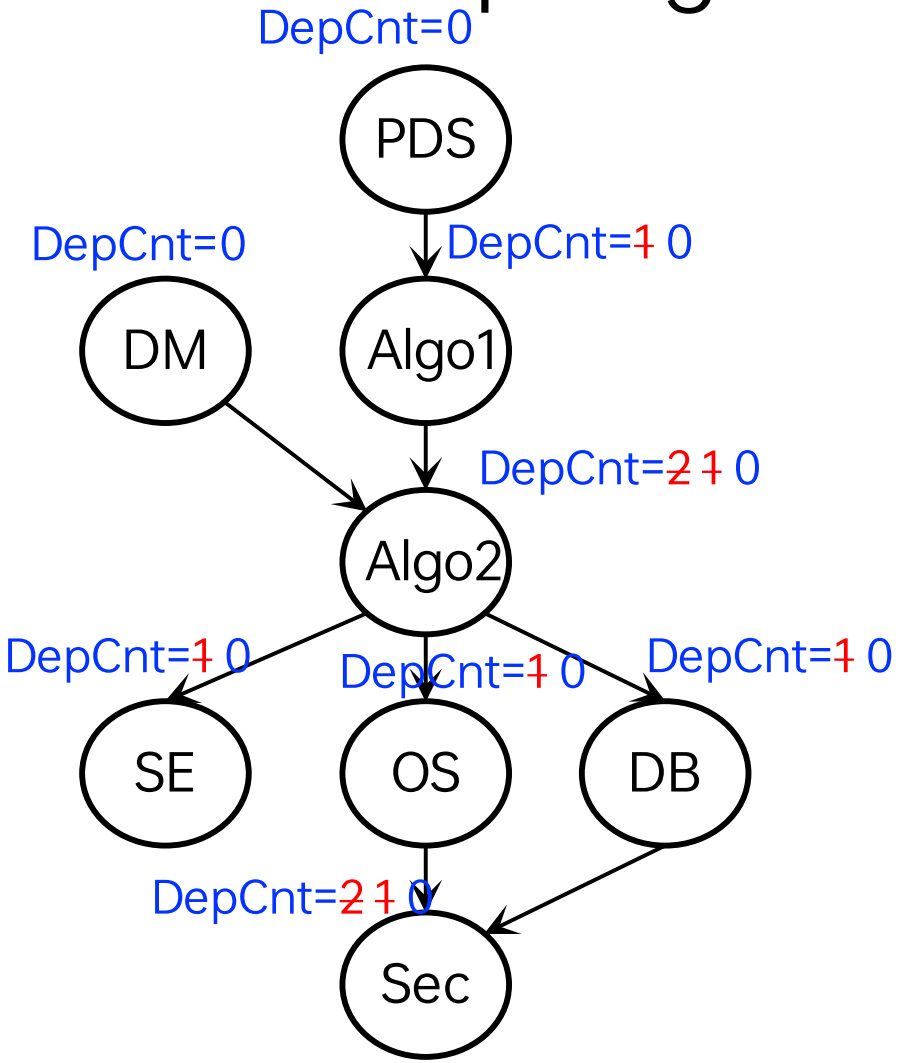
Topological Sort



PDS, Algo1, DM, Algo2, SE, OS, DB, Sec

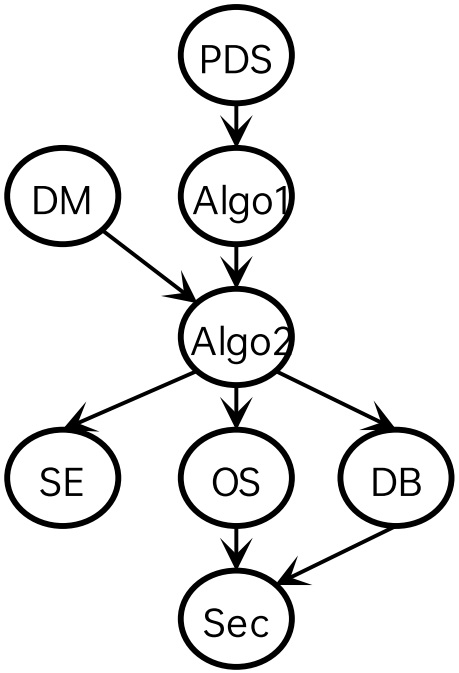


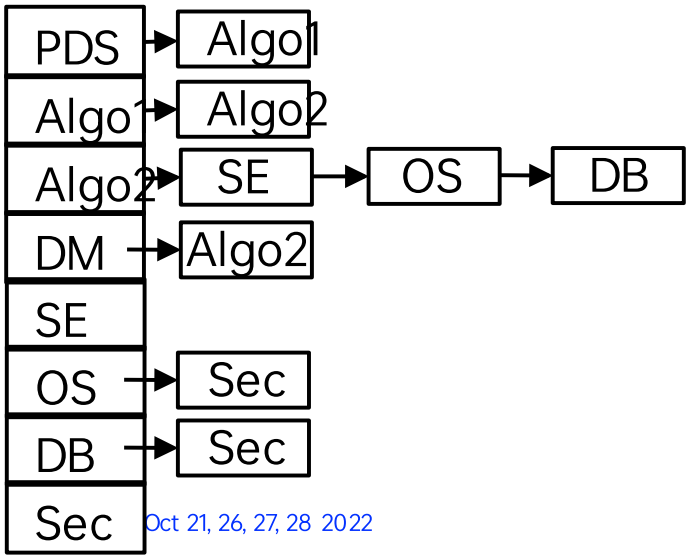
Topological Sort - Pseudocode




```
TopoSort(G)
    for each vertex u ∈ G.V
        u.DepCnt = 0
    for each vertex u ∈ G.V
        for each neighbor v ∈ G.Adj[u]
            v.DepCnt++
    // Initialize two lists – one for 'ready' nodes and
    one for 'solution' nodes
    for each vertex u ∈ G.V
        if u.DepCnt == 0
            R.Append(u)
    S = ϕ
    while R ≠ ϕ
        u = R.Pop()
        S.Append(u)
        for each neighbor v ∈ G.Adj[u]
            v.DepCnt--
            if v.DepCnt == 0
                R.Append(v)
```

Topological Sort - Analysis

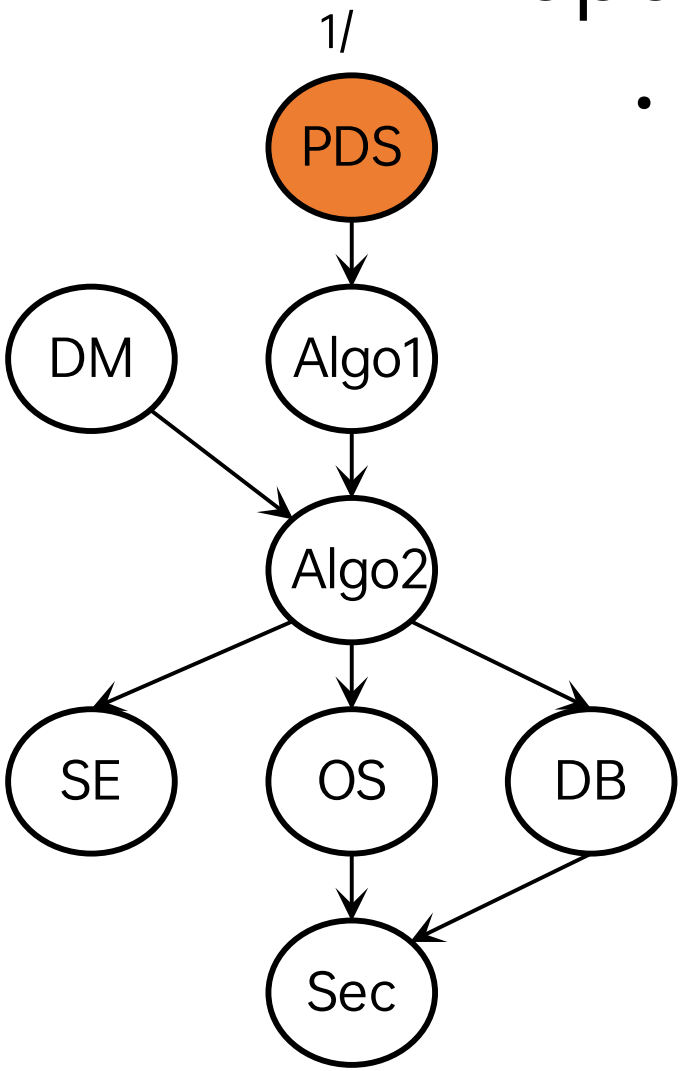





```
TopoSort(G)
    for each vertex u ∈ G.V                          → Θ(V)
        u.DepCnt = 0
    for each vertex u ∈ G.V                          → θ(E)
        for each neighbor v ∈ G.Adj[u]
            v.DepCnt++
    // Initialize two lists – one for 'ready' nodes and
    one for 'solution' nodes
    for each vertex u ∈ G.V                          → Θ(V)
        if u.DepCnt == 0
            R.Append(u)
    S = ϕ
    while R ≠ ϕ
        u = R.Pop()
        S.Append(u)
        for each neighbor v ∈ G.Adj[u]               → θ(E)
            v.DepCnt--
            if v.DepCnt == 0
                R.Append(v)
```

Overall asymptotic complexity is $\theta(V + E)$

Topological Sort - DFS



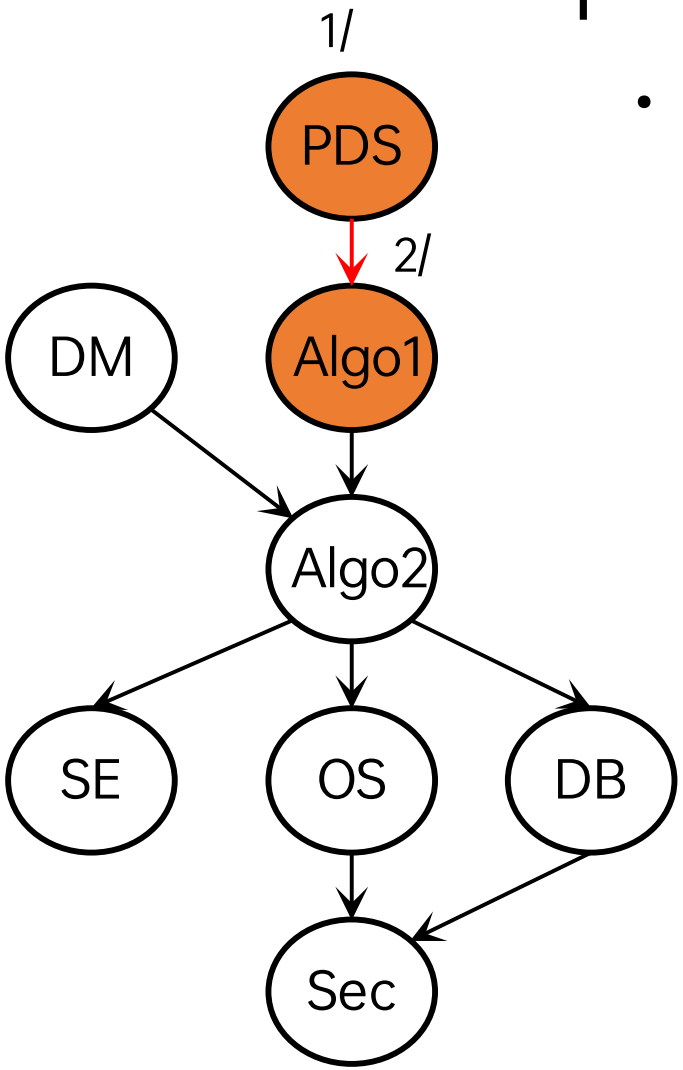
- DFS can be an easier way to perform topological sorting

Not been there yet

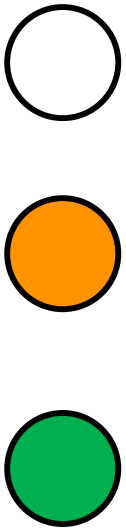
Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting

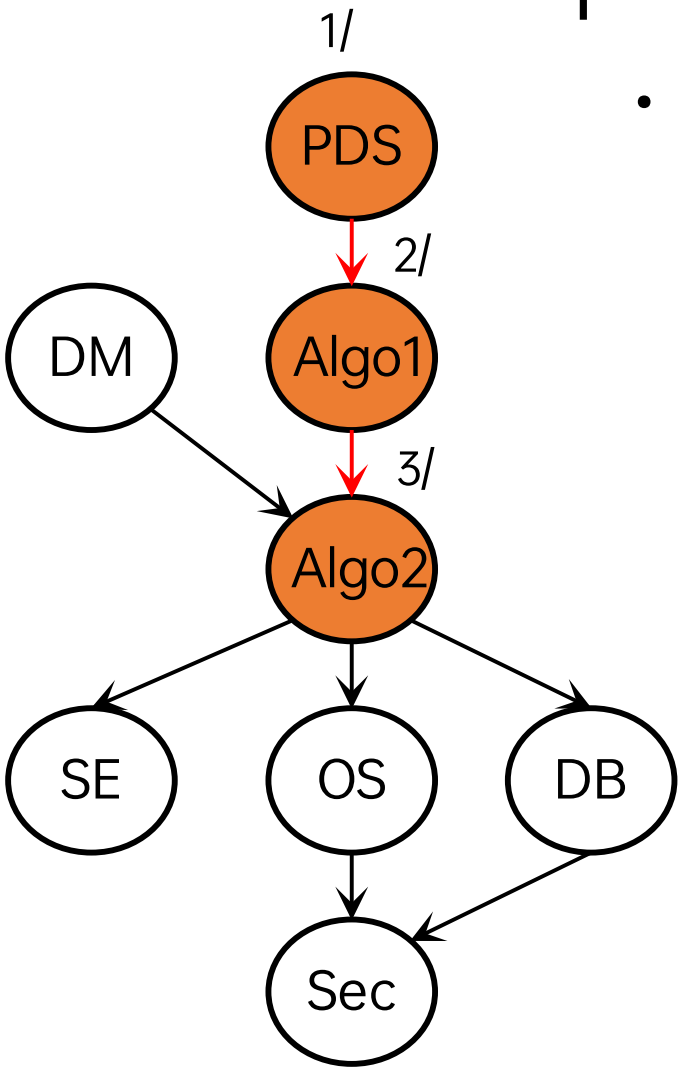


Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



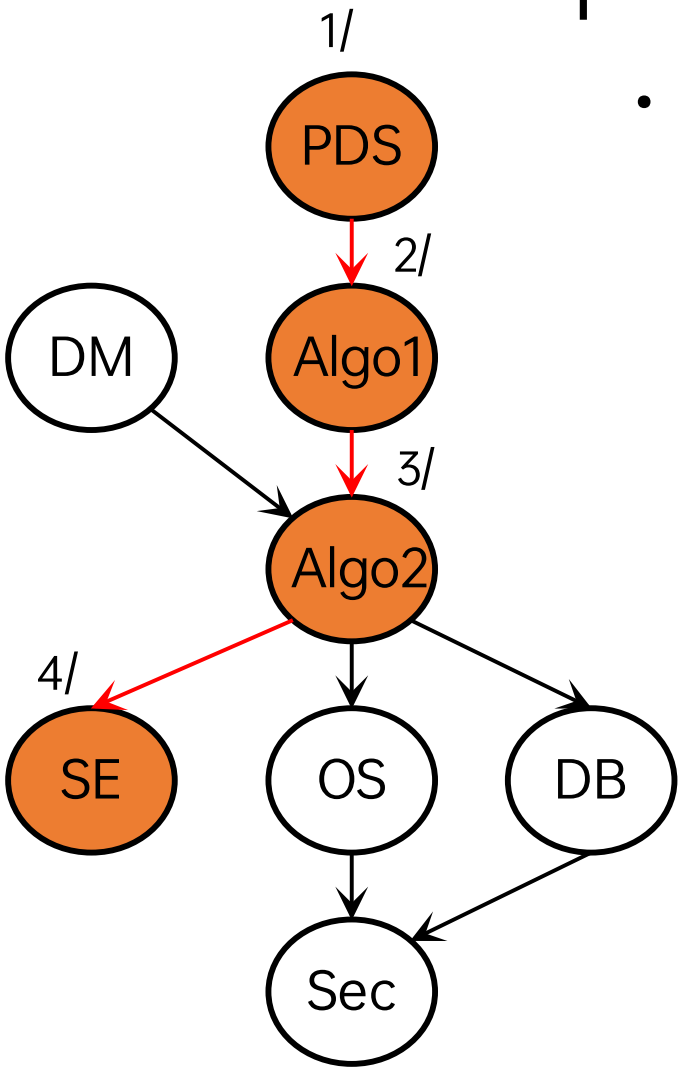
- DFS can be an easier way to perform topological sorting

Not been there yet

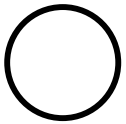
Been there, haven't explored all the paths out

Been there, have explored all the paths out

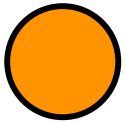
Topological Sort - DFS



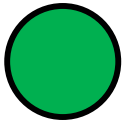
- DFS can be an easier way to perform topological sorting



Not been there yet

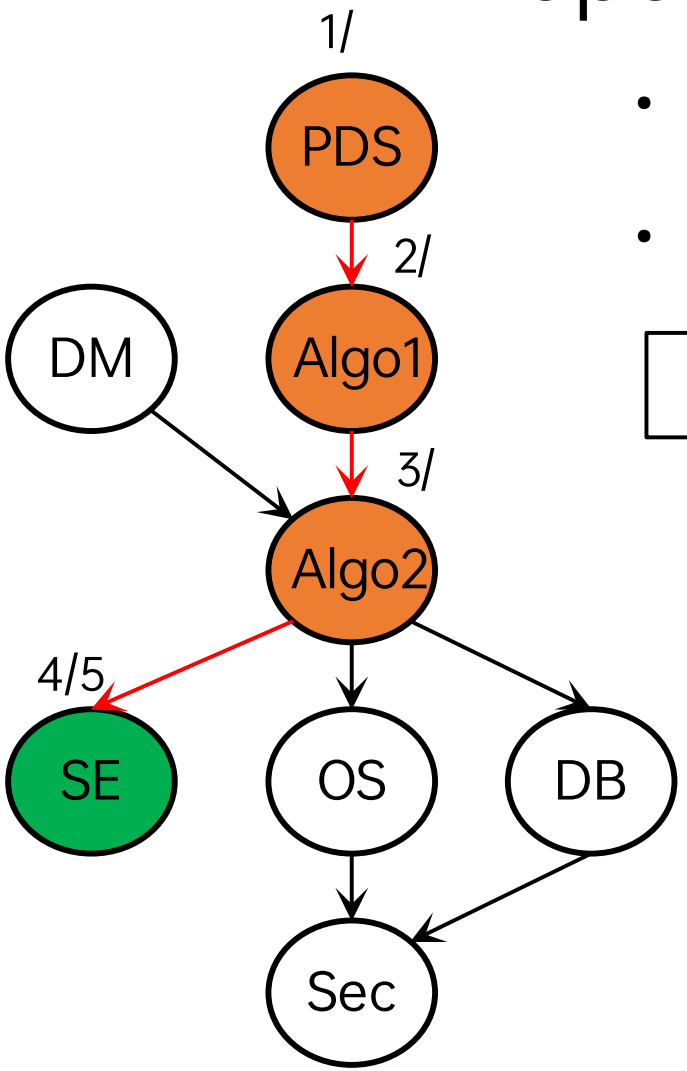


Been there, haven't explored all the paths out



Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

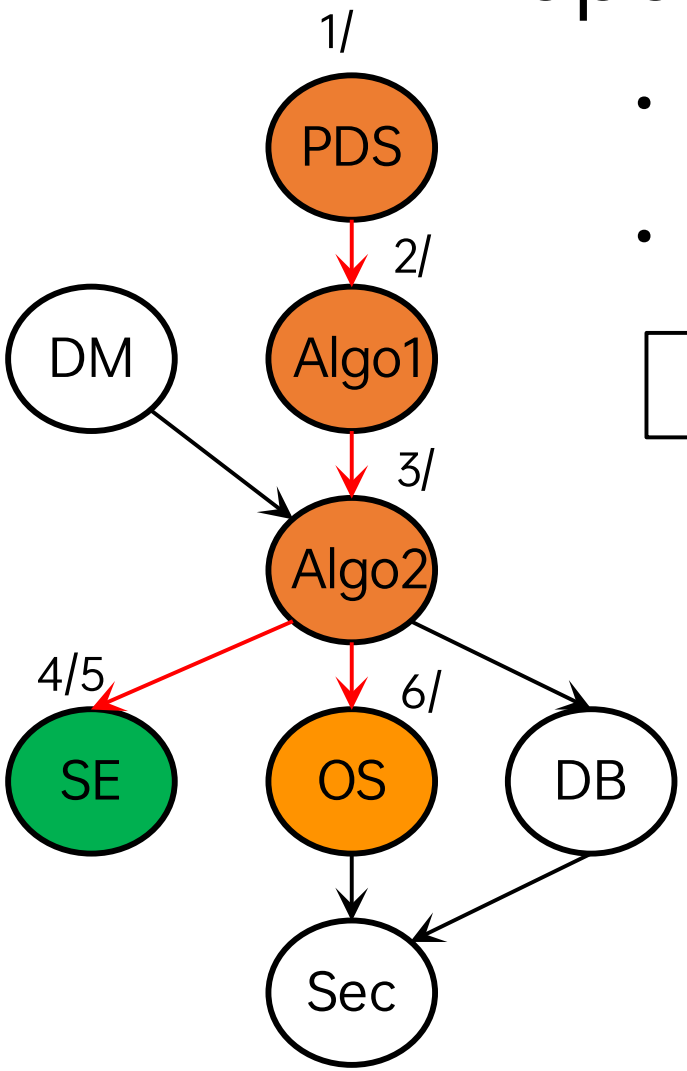
	SE

Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

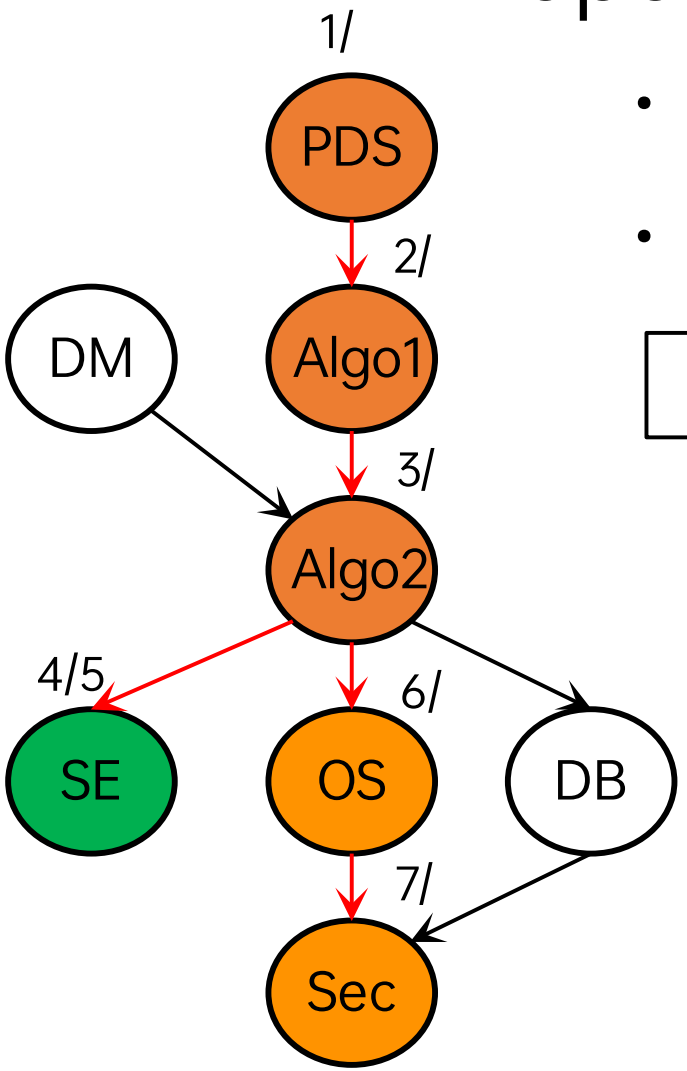
	SE

Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

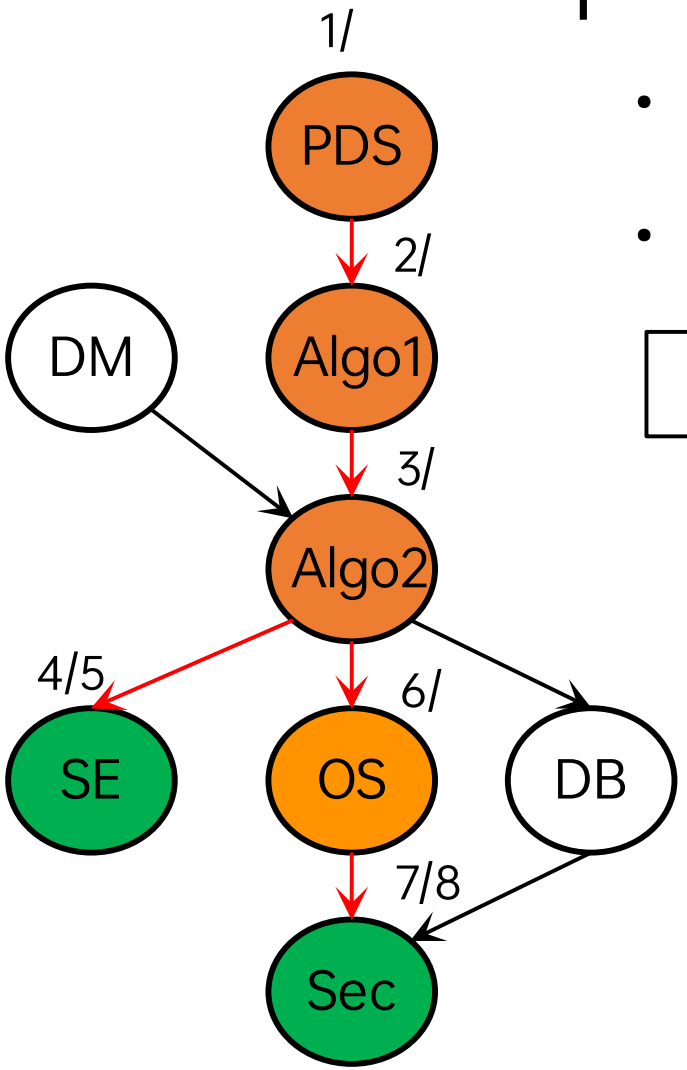
	SE

Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

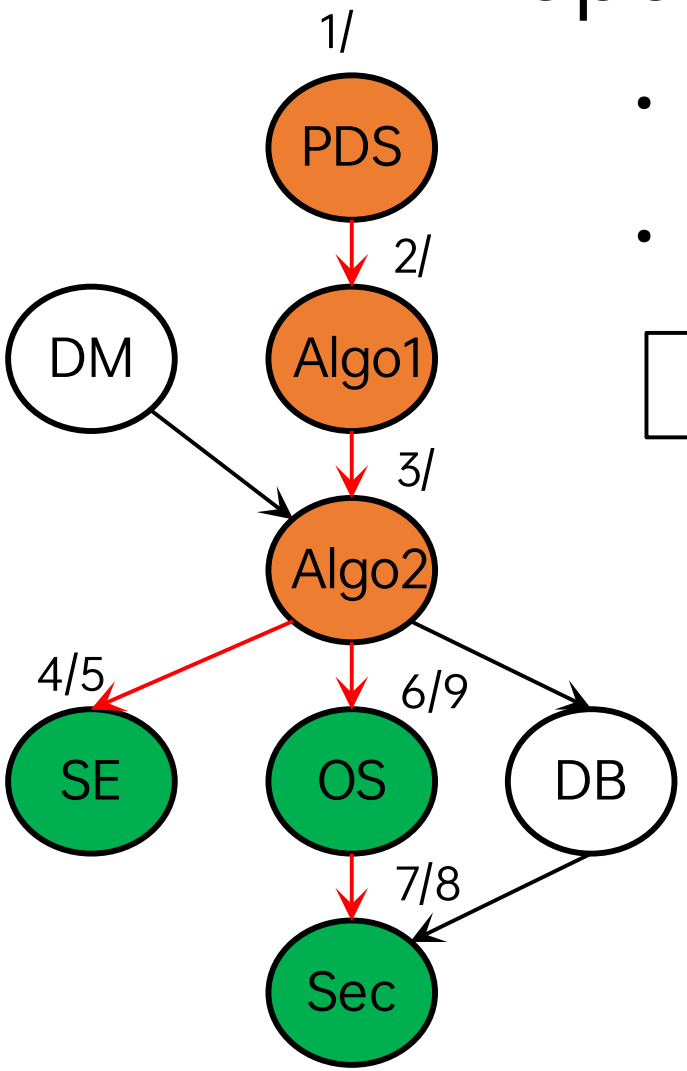
Sec, SE

Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

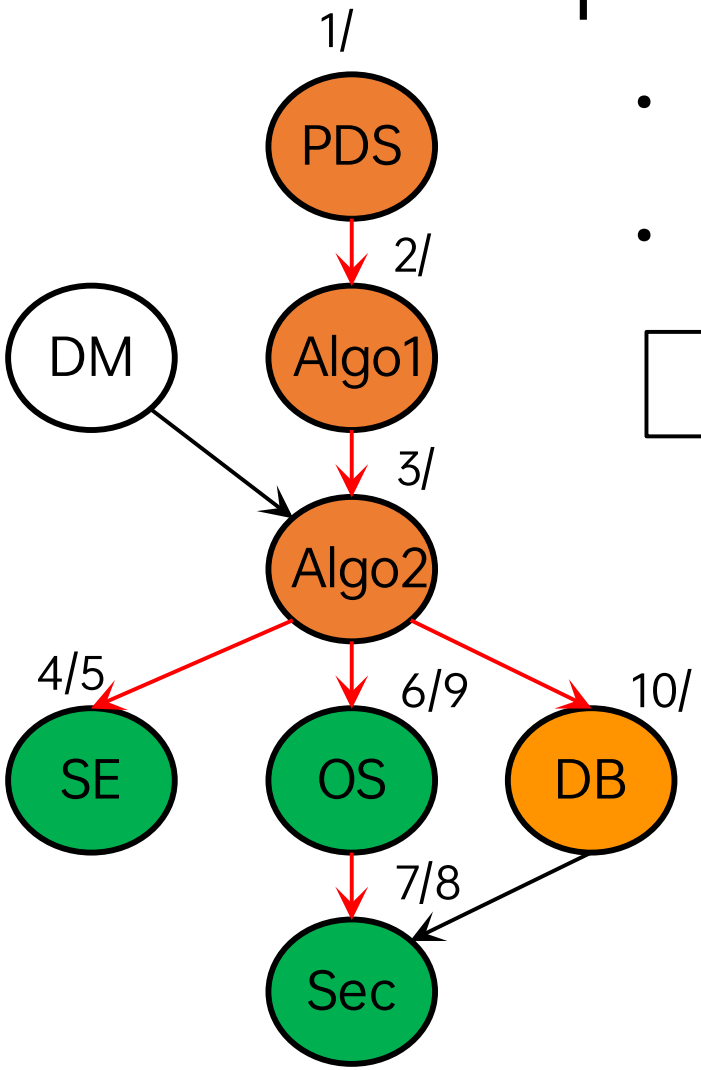
OS, Sec, SE

Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

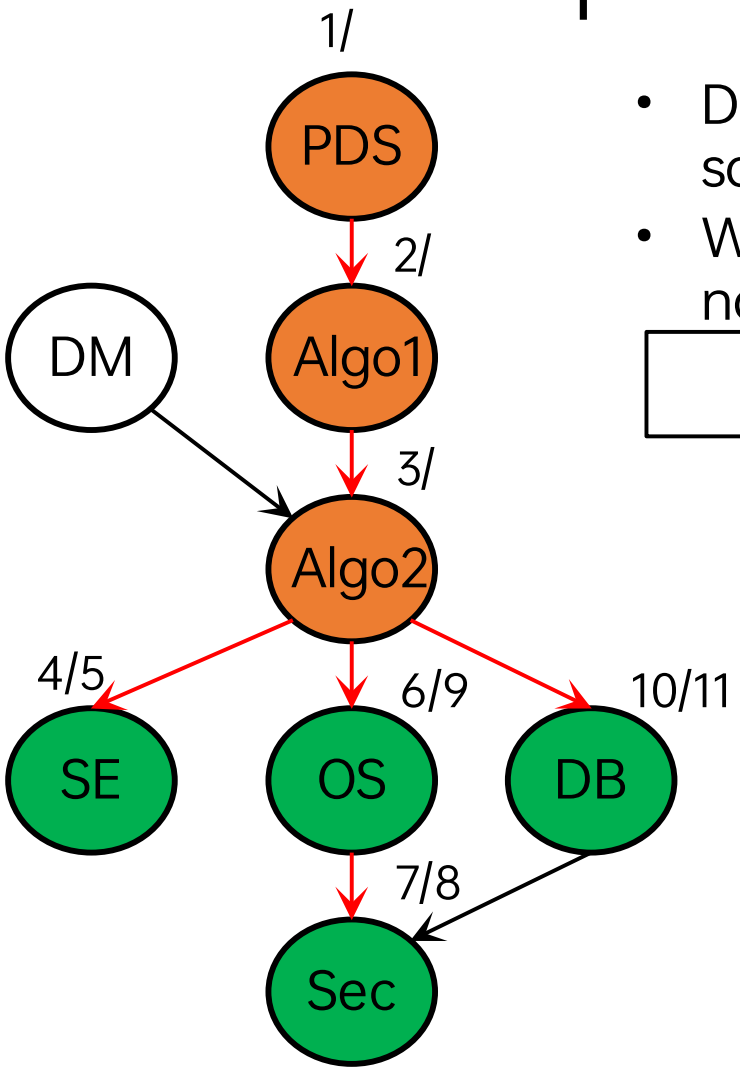
OS, Sec, SE

Not been there yet

Been there, haven't explored all the paths out

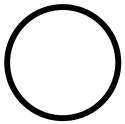
Been there, have explored all the paths out

Topological Sort - DFS

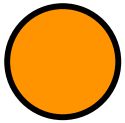


- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

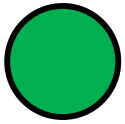
DB, OS, Sec, SE



Not been there yet

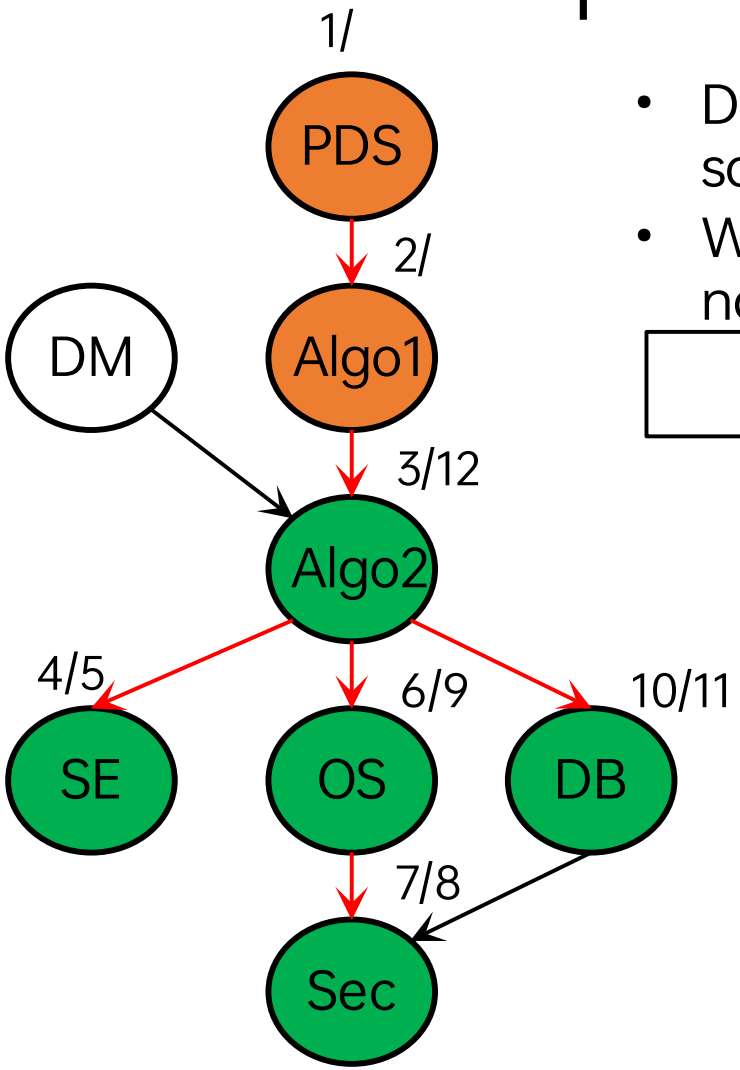


Been there, haven't explored all the paths out



Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

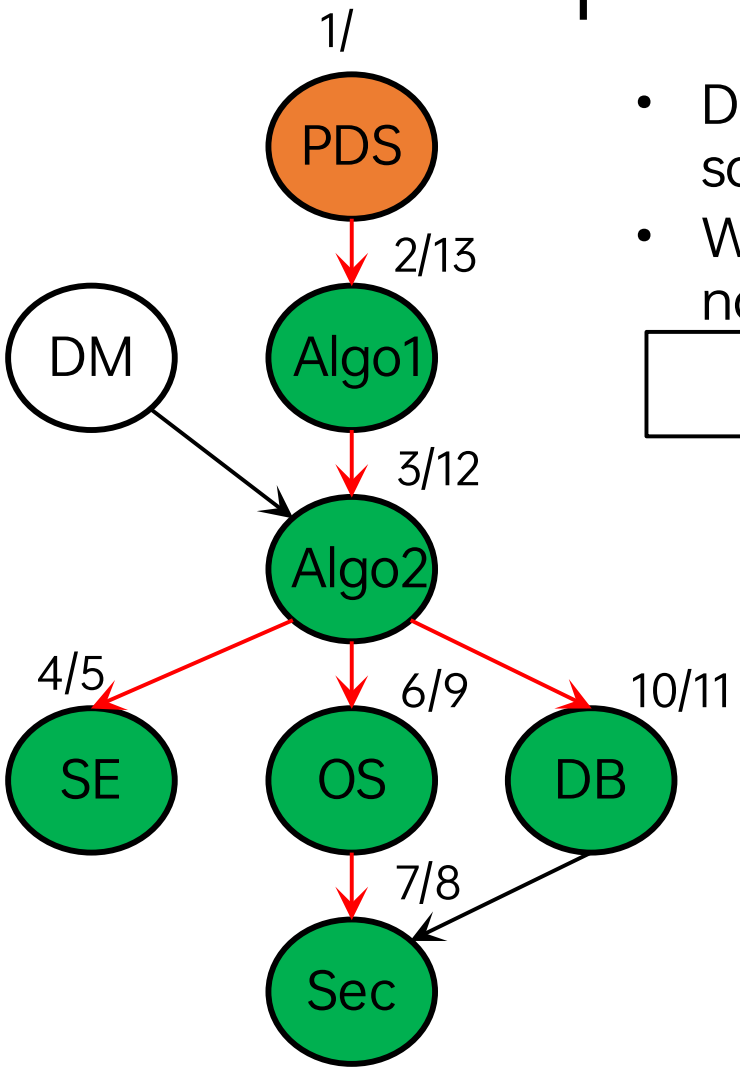
Algo2, DB, OS, Sec, SE

Not been there yet

Been there, haven't explored all the paths out

Been there, have explored all the paths out

Topological Sort - DFS



- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

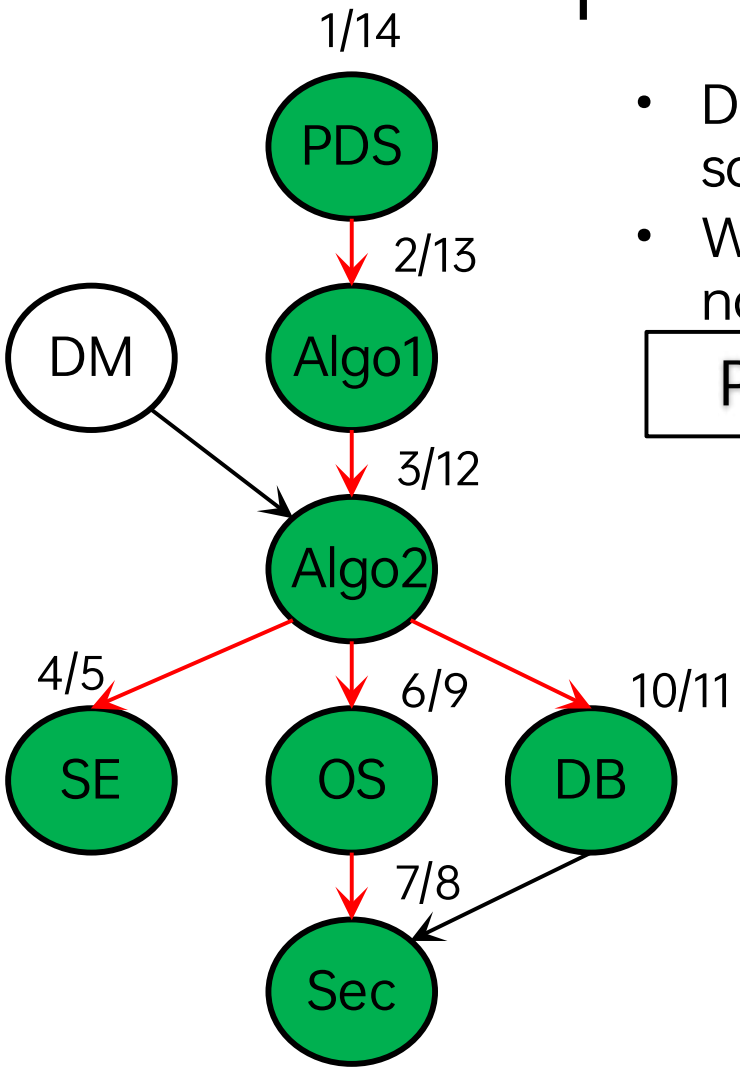
Algo1, Algo2, DB, OS, Sec, SE

Not been there yet

Been there, haven't explored all the paths out

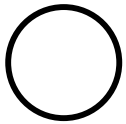
Been there, have explored all the paths out

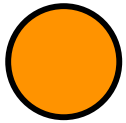
Topological Sort - DFS

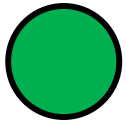


- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

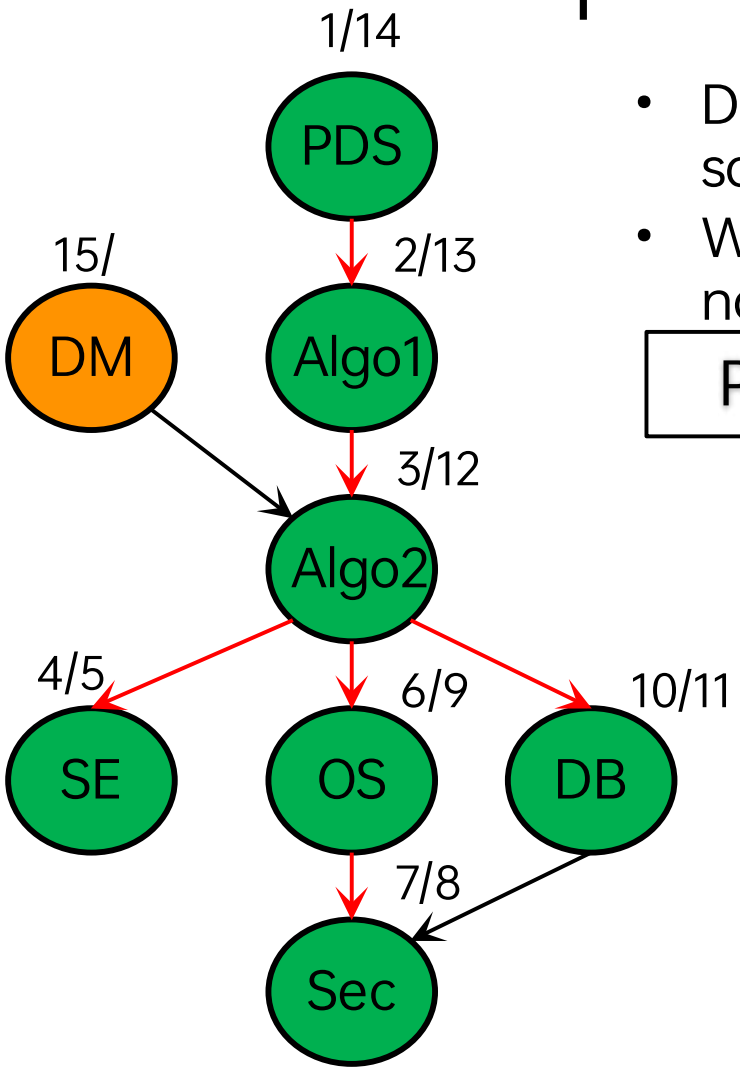
PDS, Algo1, Algo2, DB, OS, Sec, SE


Not been there yet


Been there, haven't explored all the paths out

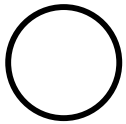

Been there, have explored all the paths out

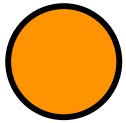
Topological Sort - DFS

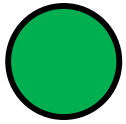


- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

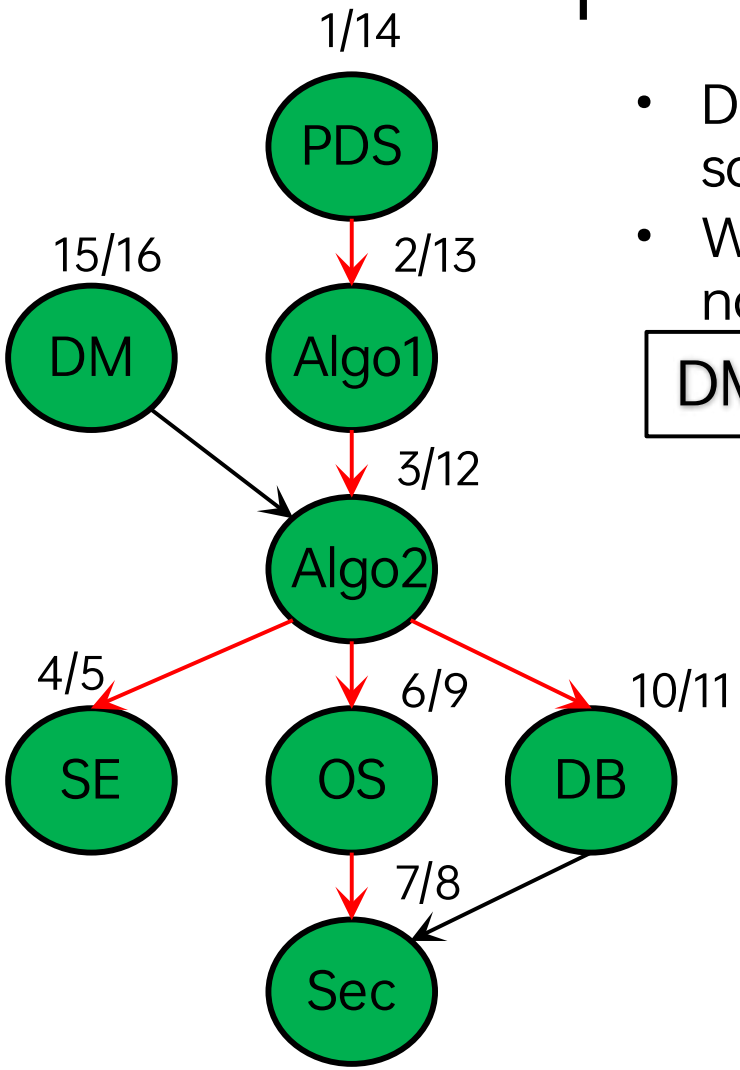
PDS, Algo1, Algo2, DB, OS, Sec, SE


Not been there yet


Been there, haven't explored all the paths out

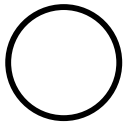

Been there, have explored all the paths out

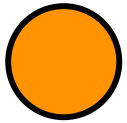
Topological Sort - DFS

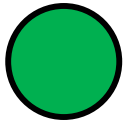


- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

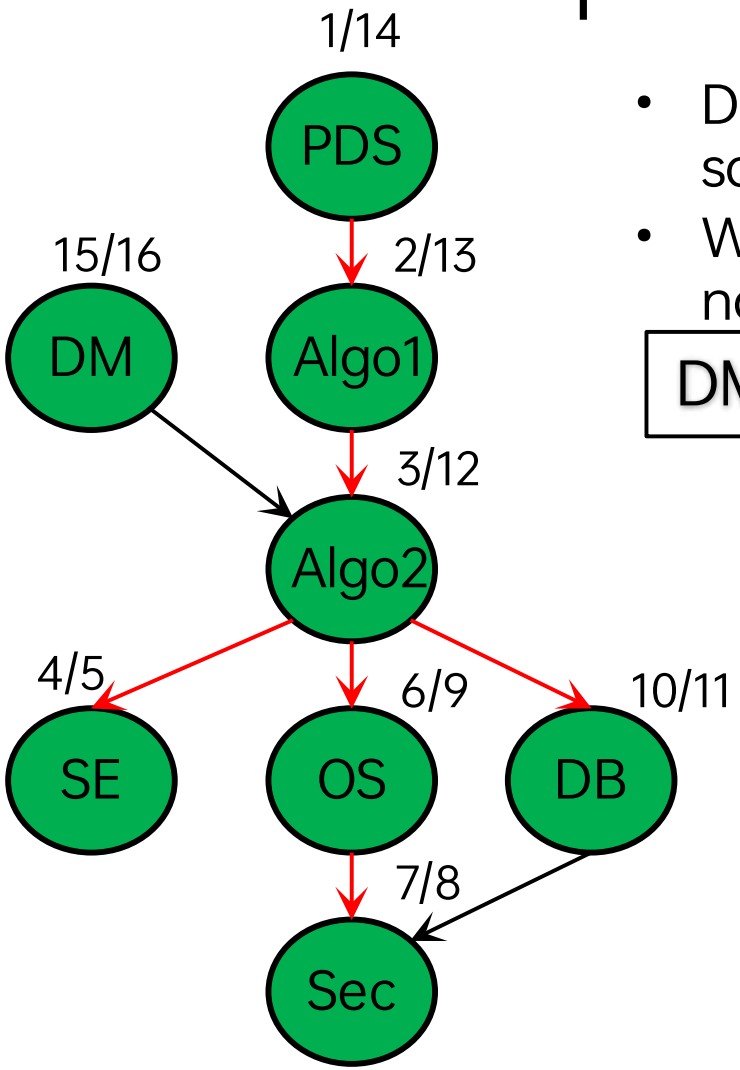
DM, PDS, Algo1, Algo2, DB, OS, Sec, SE


Not been there yet


Been there, haven't explored all the paths out


Been there, have explored all the paths out

Topological Sort - DFS

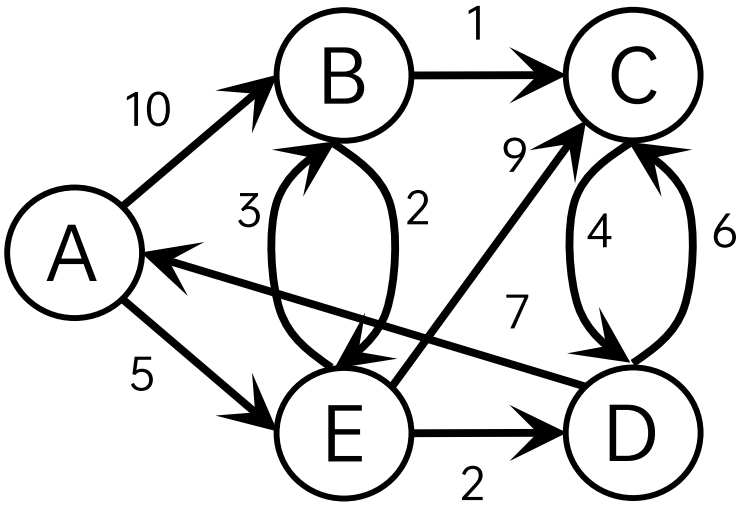


- DFS can be an easier way to perform topological sorting
- What can we say or where can we put a finished node in the list?

DM, PDS, Algo1, Algo2, DB, OS, Sec, SE

- The algorithm simply put is,
- Do DFS and whenever a vertex u is finished do $S.AddToHead(u)$
- The complexity is same as DFS i.e., $\theta(V + E)$

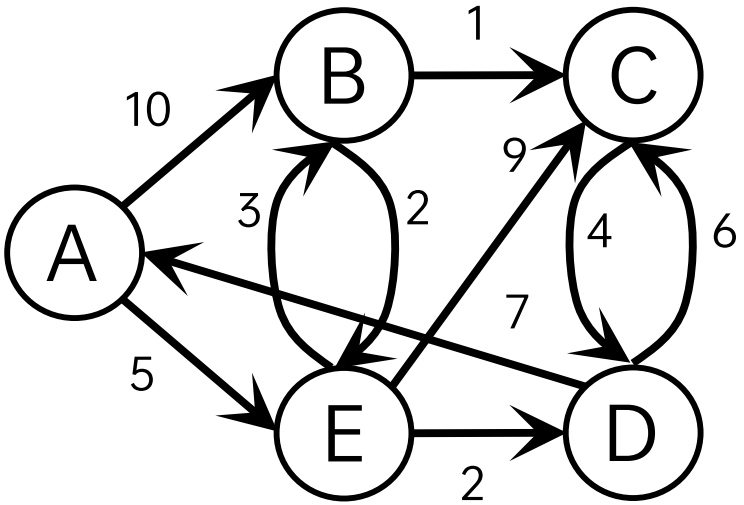
Single Source Shortest Paths



- Given a weighted directed graph and a source the problem is to find the shortest paths to every vertex of the graph from the source
- If A is the source, then shortest distances to all the nodes i.e., shortest distances $(A, B), (A, C), (A, D)$ and (A, E) are required
- Note that it's a weighted graph

- **Shortest-paths problem**: We are given a weighted, directed graph $G = (V, E)$, with weight function $w: E \rightarrow \mathbb{R}$ mapping edges to real-valued weights. The weight $w(p)$ of path $p =< v_0, v_1, \ldots, v_k >$ is the sum of the weights of its constituent edges.
- We define the shortest-path weight $\delta(u, v)$ from u to v by
- $\delta(u, v) = \begin{cases} \min\{w(p): u \overset{p}{\rightsquigarrow} v\} & if\ there\ is\ a\ path\ from\ u\ to\ v \\ \infty. & otherwise \end{cases}$
- A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$

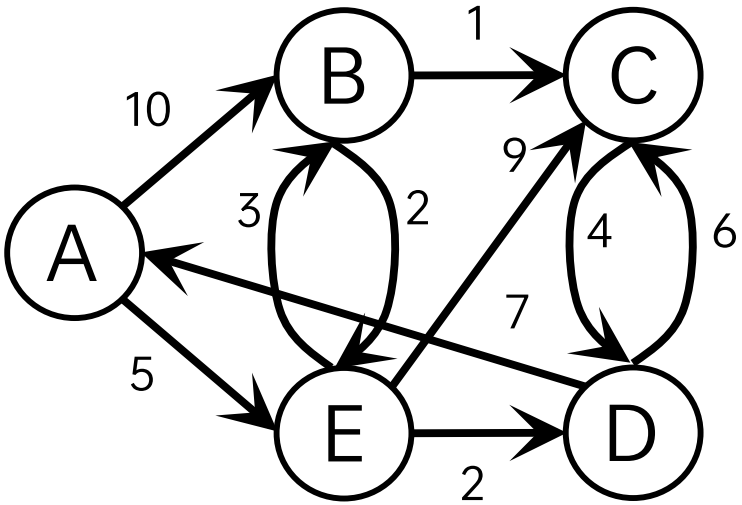
Shortest Path Variants



- **Single-destination shortest-paths problem**: Finding a shortest path to a given destination vertex t from each vertex v
- By reversing the direction of each edge in the graph the problem can be reduced to a single-source problem

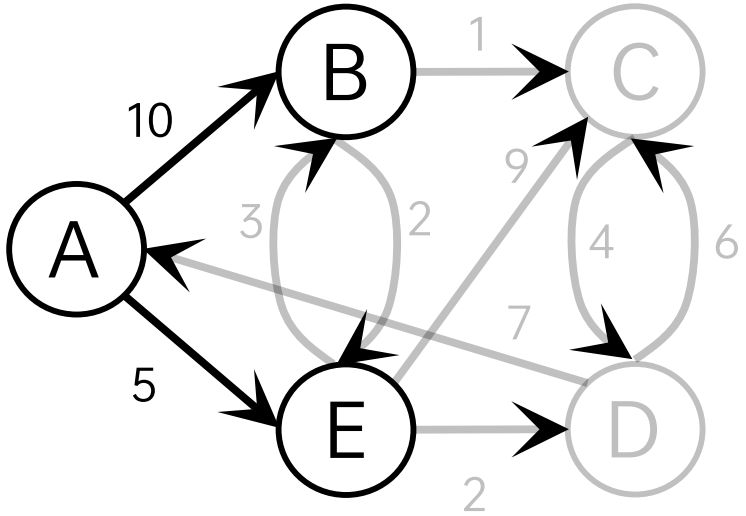
- **Single-pair shortest-path problem**: Find a shortest path from u to v for given vertices u and v
- If we solve the single-source problem with source vertex u, we solve this problem also
- **All-pairs shortest-paths problem**: Find a shortest path from u to v for every pair of vertices u and v
- Can be solved by running a single source algorithm from each vertex. However, we usually can solve it faster
- Will see such algorithms later (if time permits)

Single Source Shortest Paths



- Of course, if the graph is unweighted, we can use BFS
- As the problem in hand is more general, the solution is expected to be more complex than BFS
- Let us consider A as the source node and try to observe a few things

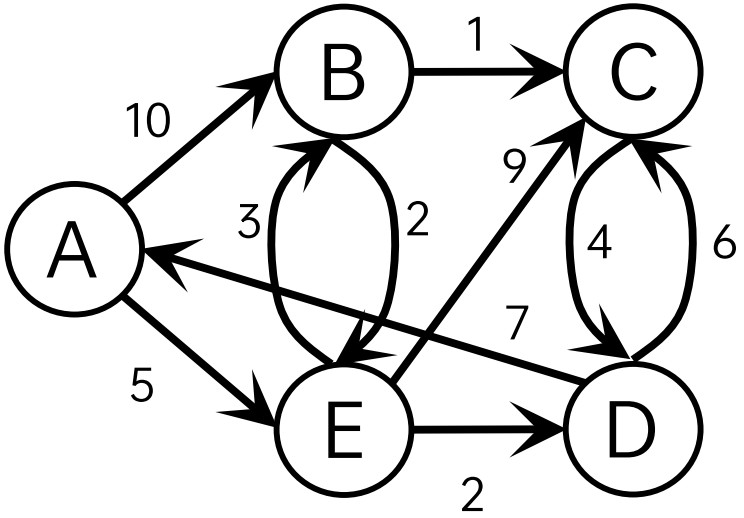
Single Source Shortest Paths

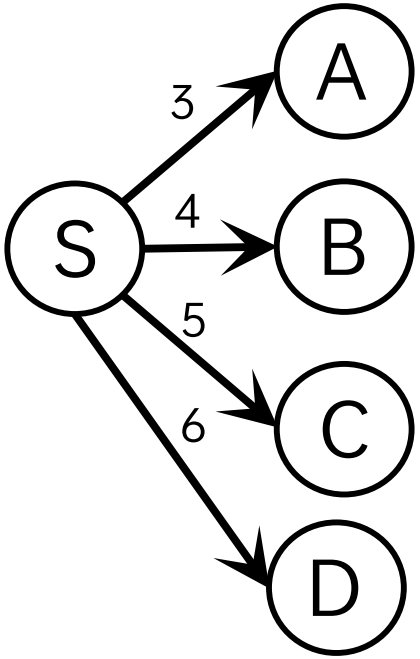


- Of course, if the graph is unweighted, we can use BFS
- As the problem in hand is more general, the solution is expected to be more complex than BFS
- Let us consider A as the source node and try to observe a few things

- Considering A as the source
  - Is 5 the shortest distance between A and E?
  - Is 10 the shortest distance between A and B?
  - Answer is Yes and 'can't tell' respectively
  - For AE, any other path (via any other neighbor of A) will incur more cost than 5 (as 5 is the least weight among the neighboring edges of A)
  - For AB, we can not say this for sure as there may be paths via other neighbors of A (say path via AE) whose sum of weights may be less

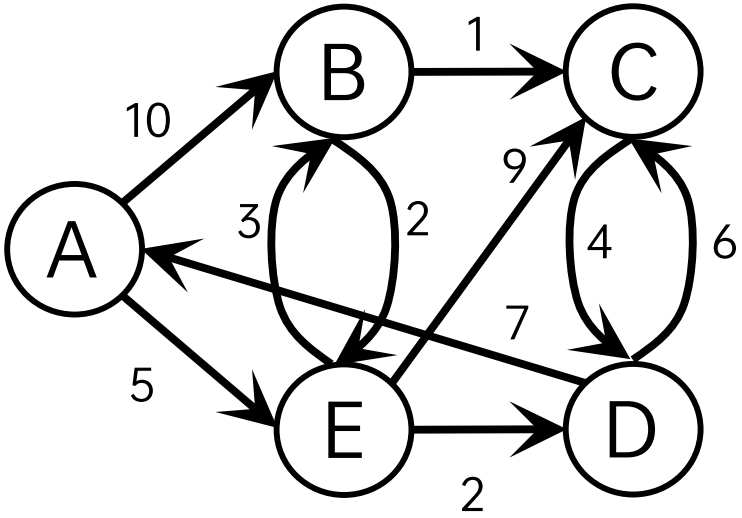
Single Source Shortest Paths

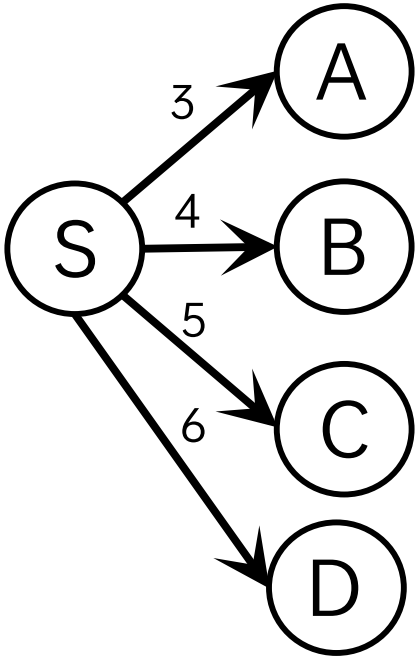




- Generally speaking for a scenario shown in top-right, only for edge SA we know for sure that this is the shortest path from S to A
- For others, we are not sure
- What is the assumption here?
- Edge weights are always positive
- Dijkstra's algorithm is guaranteed to work for graphs with positive edge weights
- For graphs with negative edge weights, we will study another algorithm

Single Source Shortest Paths





- For the node A, the path is the shortest path to A
- For all other nodes we are not sure
- This slick fact is exploited in Dijkstra's algorithm
- Dijkstra's algorithm is thus a greedy algorithm
- Lets see what Dijkstra's philosophy about 'algorithms' is

Single Source Shortest Paths

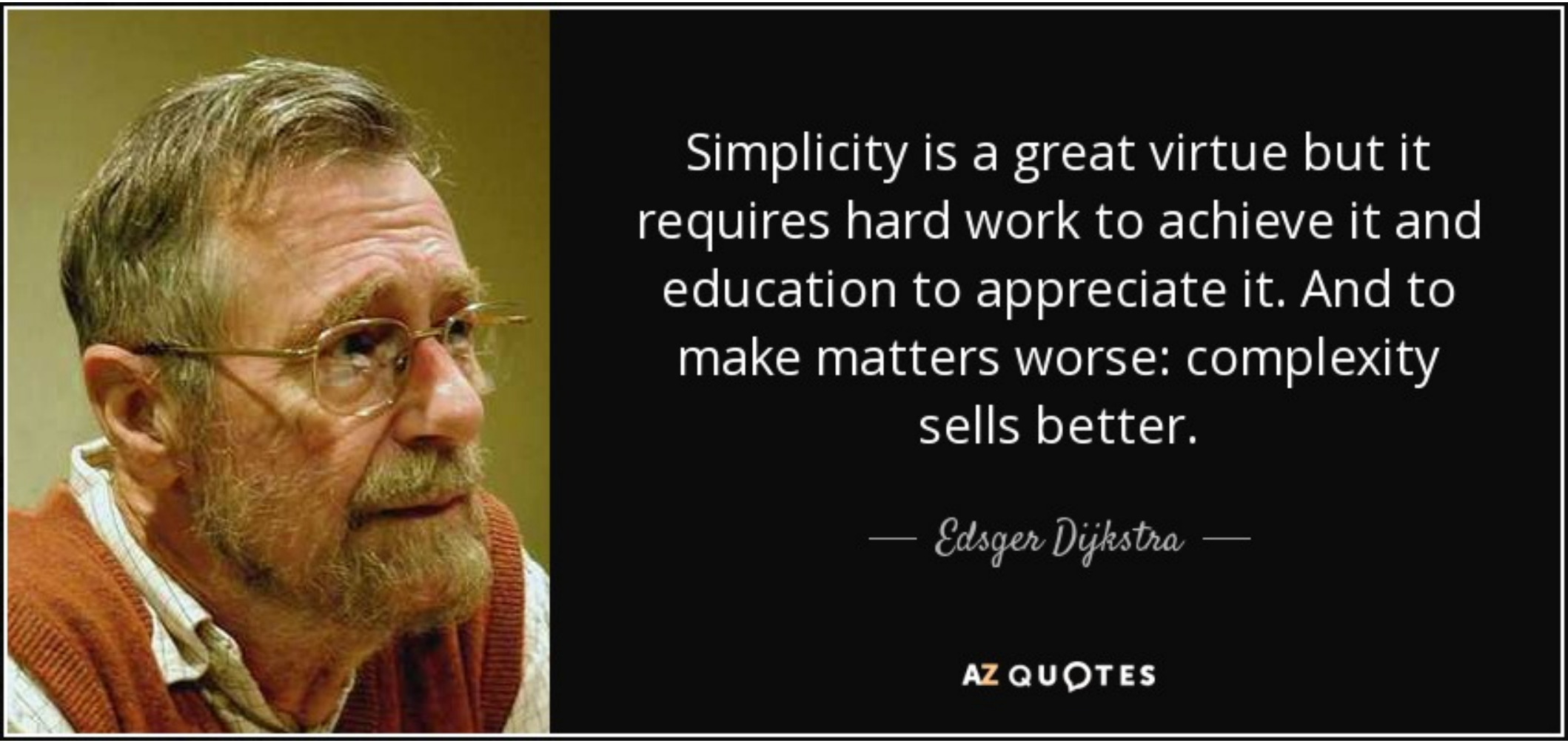
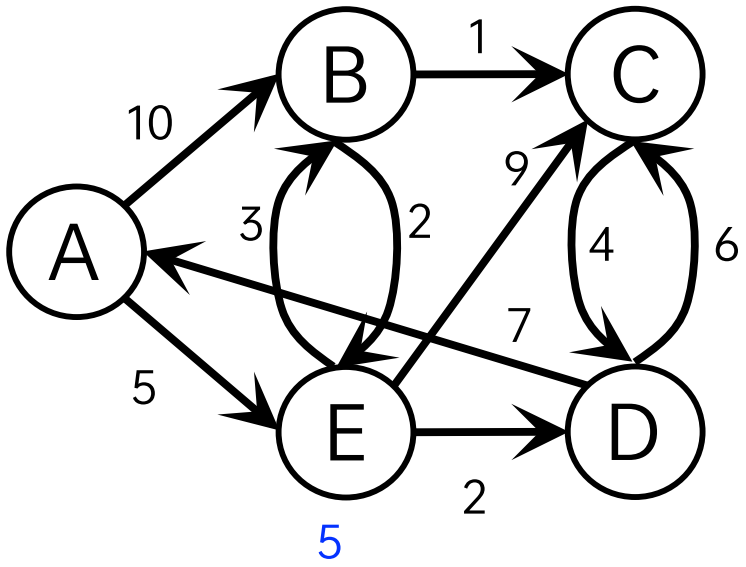


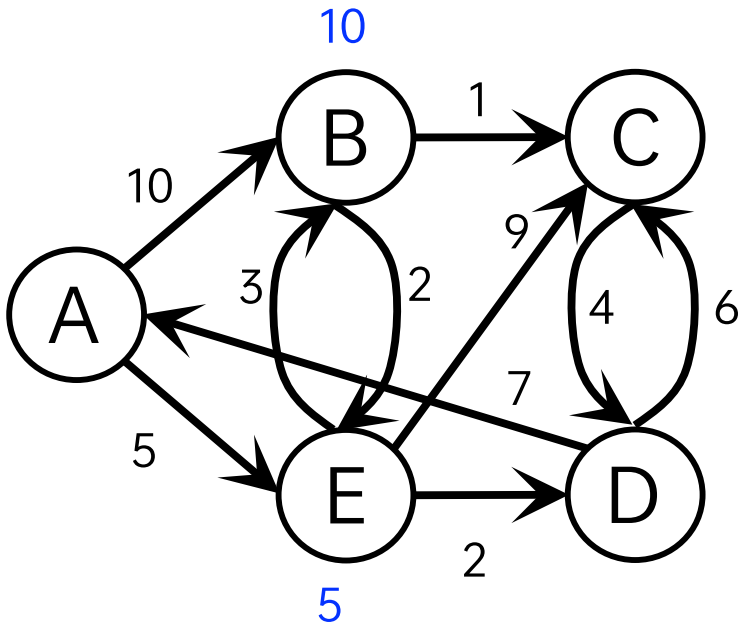
Image source: www.azquotes.com/quote/754978

Single Source Shortest Paths



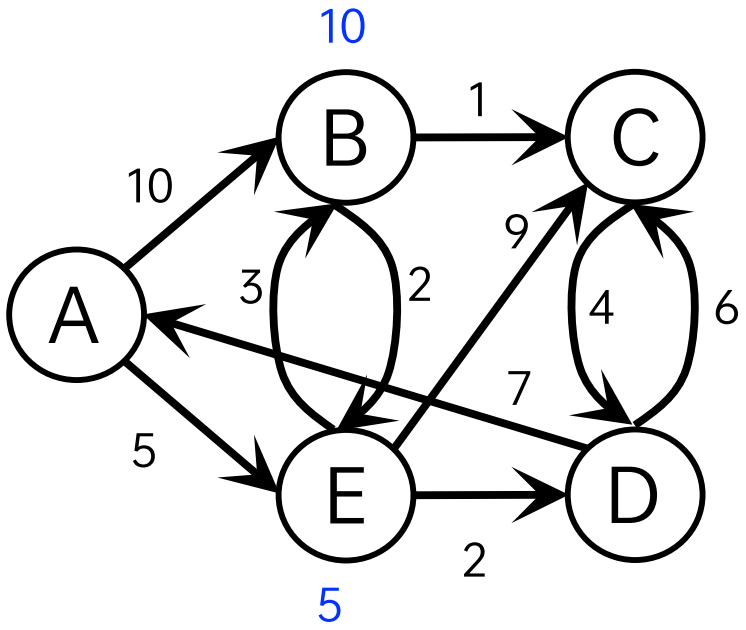
	d	
A	0	Null
B		Null
C		Null
D		Null
E		Null

Single Source Shortest Paths



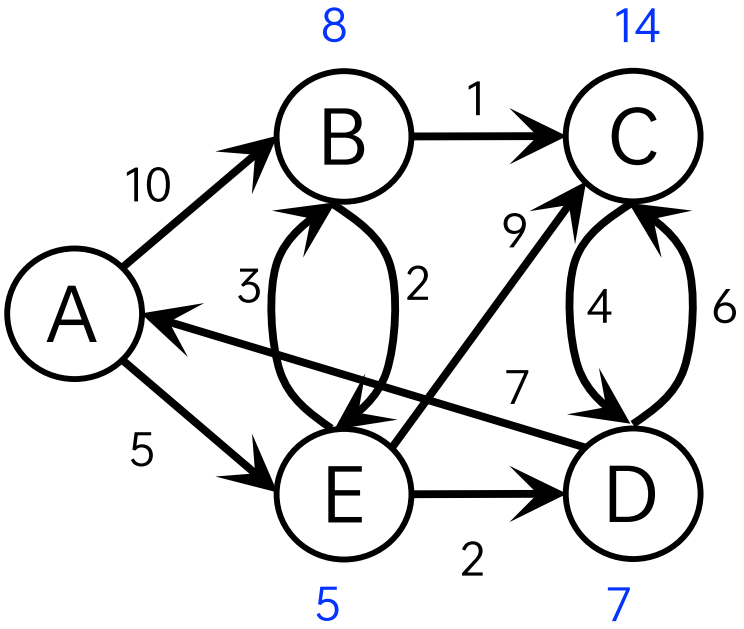
	d	
A	0	Null
B	10	~~Null~~ A
C		Null
D		Null
E	5	~~Null~~ A

Single Source Shortest Paths



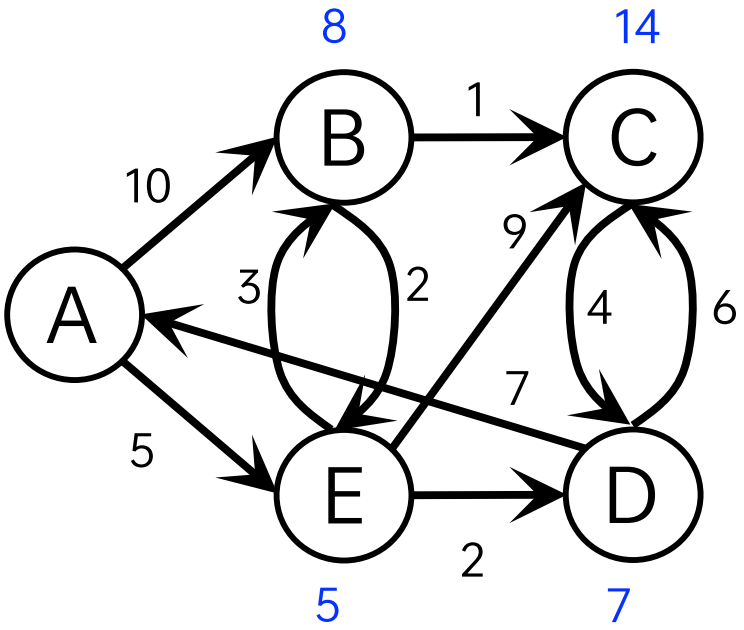
	d	
A	0	Null
B	10	~~Null~~ A
C		Null
D		Null
E	5	~~Null~~ A

Single Source Shortest Paths



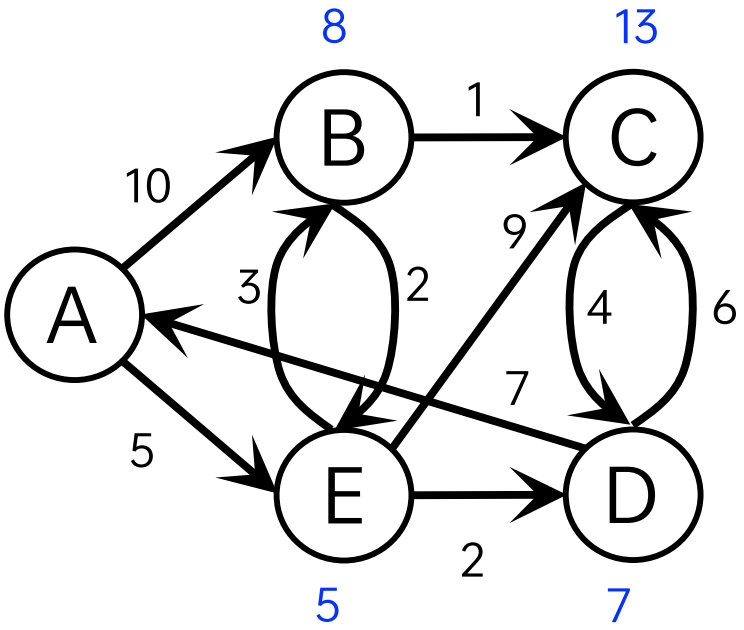
	d	
A	0	Null
B	~~10~~ 8	~~Null~~ ~~A~~ E
C	14	~~Null~~ E
D	7	~~Null~~ E
E	5	~~Null~~ A

Single Source Shortest Paths



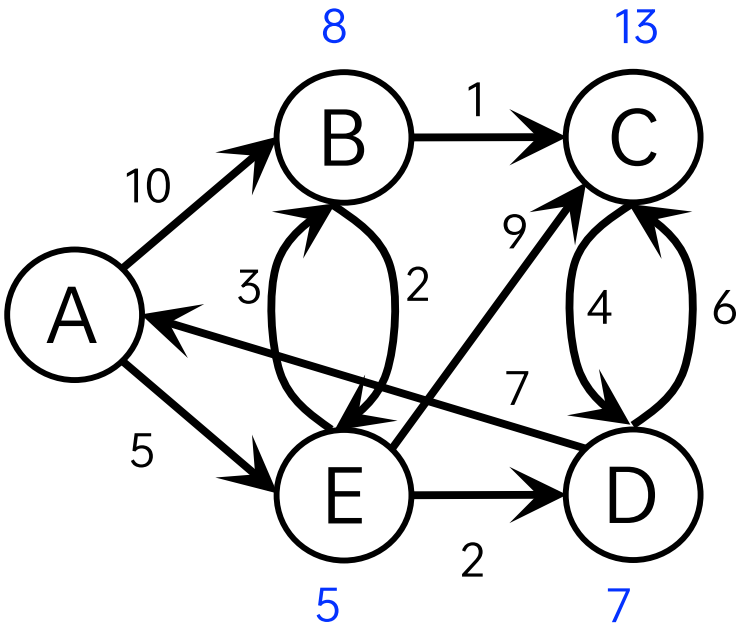
	d	
✅A	0	Null
B	~~10~~ 8	~~Null~~ ~~A~~ E
C	14	~~Null~~ E
D	7	~~Null~~ E
✅E	5	~~Null~~ A

Single Source Shortest Paths



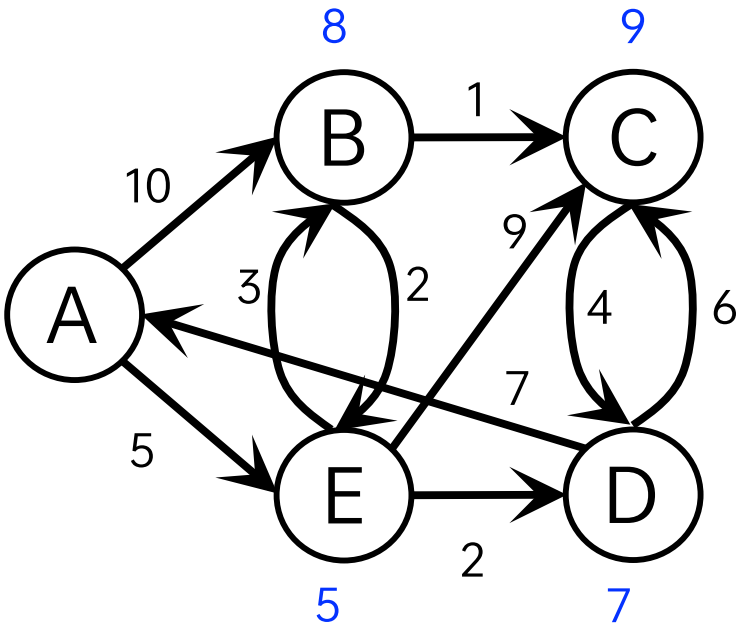
	d	
✅A	0	Null
B	~~10~~ 8	~~Null~~ ~~A~~ E
C	~~14~~ 13	~~Null~~ ~~E~~ D
D	7	~~Null~~ E
✅E	5	~~Null~~ A

Single Source Shortest Paths



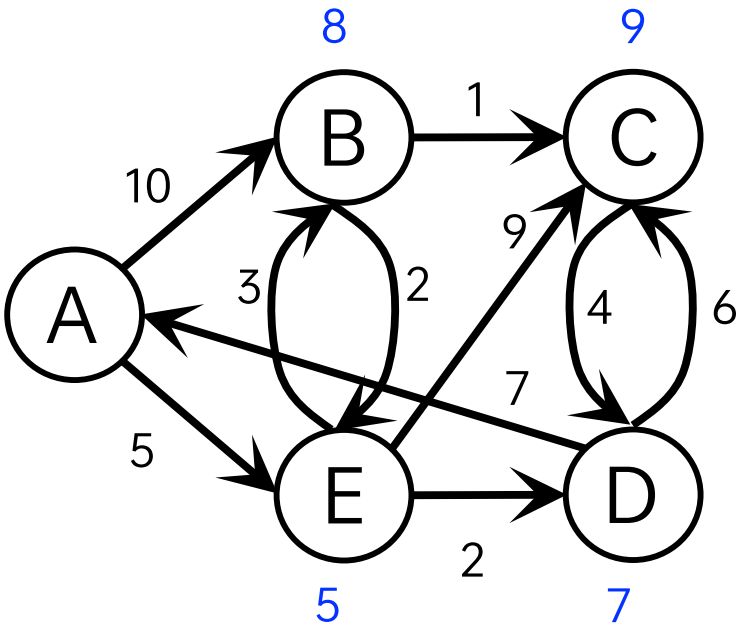
	d	
✅A	0	Null
B	~~10~~ 8	~~Null~~ ~~A~~ E
C	~~14~~ 13	~~Null~~ ~~E~~ D
✅D	7	~~Null~~ E
✅E	5	~~Null~~ A

Single Source Shortest Paths



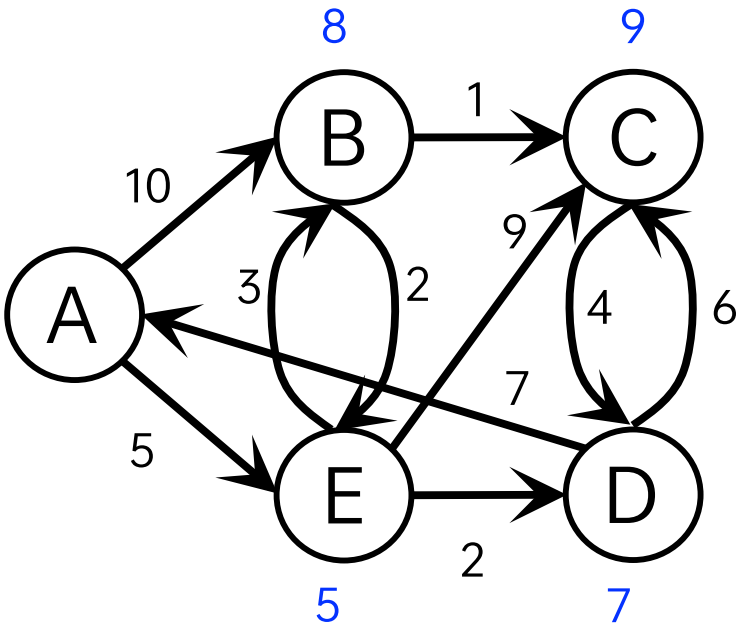
	d	
✅A	0	Null
B	~~10~~ 8	~~Null~~ ~~A~~ E
C	~~14~~ ~~13~~ 9	~~Null~~ ~~E~~ ~~D~~ B
✅D	7	~~Null~~ E
✅E	5	~~Null~~ A

Single Source Shortest Paths



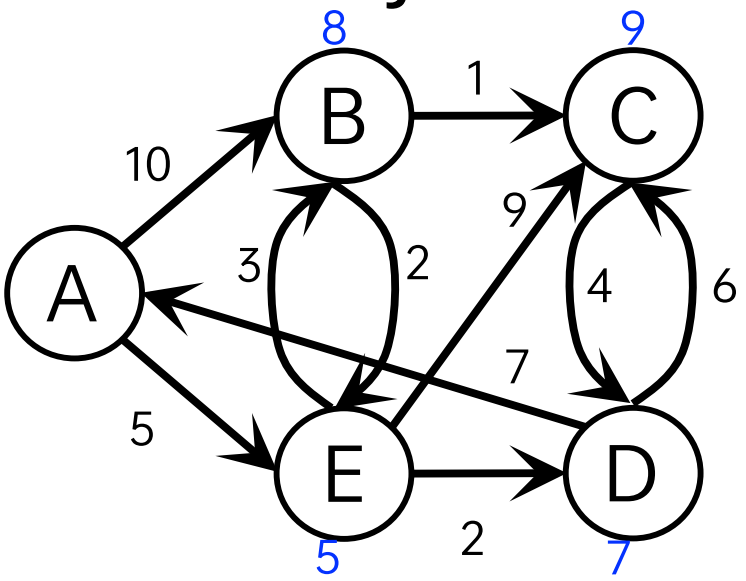
	d	
✅A	0	Null
✅B	~~10~~ 8	~~Null~~ ~~A~~ E
C	~~14~~ ~~13~~ 9	~~Null~~ ~~E~~ ~~D~~ B
✅D	7	~~Null~~ E
✅E	5	~~Null~~ A

Single Source Shortest Paths



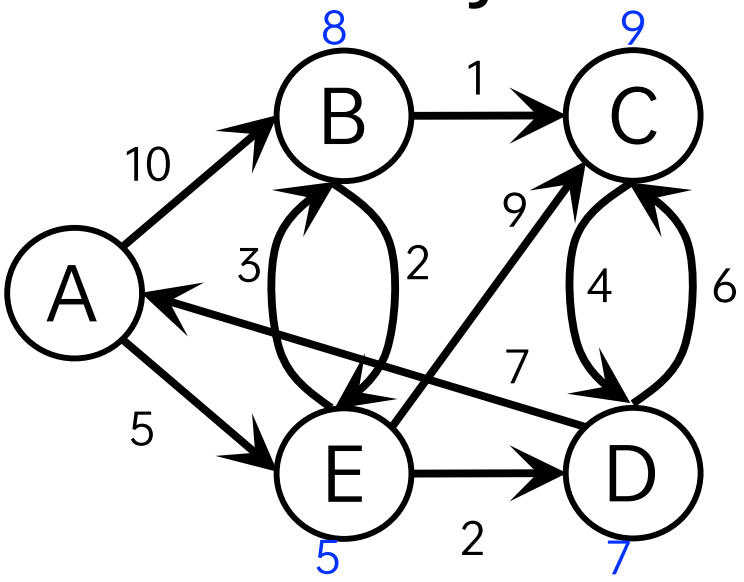
	d	
✅A	0	Null
✅B	~~10~~ 8	~~Null~~ ~~A~~ E
✅C	~~14~~ ~~13~~ 9	~~Null~~ ~~E~~ ~~D~~ B
✅D	7	~~Null~~ E
✅E	5	~~Null~~ A

Dijkstra's Algorithm - Pseudocode

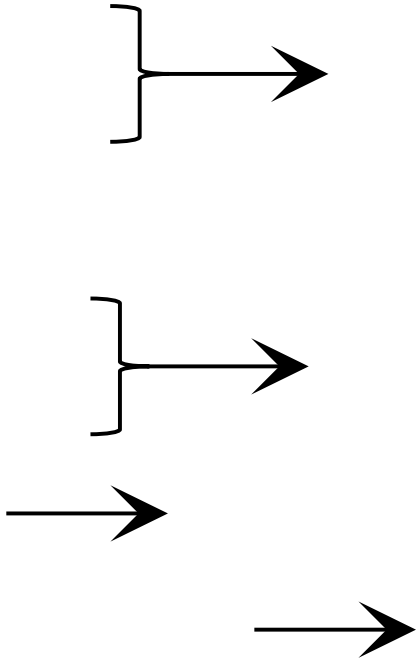


	d	
✅A	0	Null
✅B	~~10~~ 8	~~Null~~ ~~A~~ E
✅C	~~14~~ ~~13~~ 9	~~Null~~ ~~E~~ ~~D~~ B
✅D	7	~~Null~~ E
✅E	5	~~Null~~ A

Dijkstra's Algorithm - Analysis



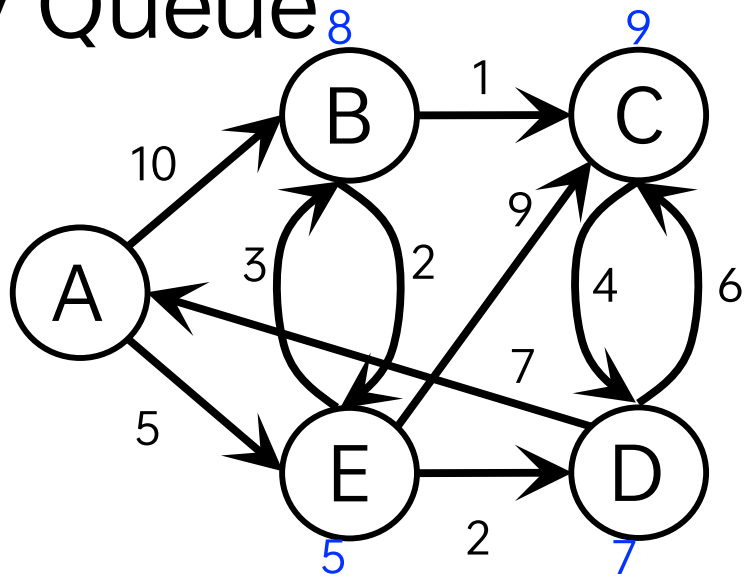
	d	
✅A	0	Null
✅B	~~10~~ 8	~~Null~~ ~~A~~ E
✅C	~~14~~ ~~13~~ 9	~~Null~~ ~~E~~ ~~D~~ B
✅D	7	~~Null~~ E
✅E	5	~~Null~~ A



As we are using priority queue, there are costs associated with inserts, extracts and change priorities

Remembering Priority Queue

```
Dijkstra(G, s)
  for each vertex u ∈ G.V                    } → Θ(V)
      u.d = ∞, u.π = Null
  s.d = 0
  // Initialize a data structure for the vertices
  Q = ϕ // Priority queue
  for each vertex u ∈ G.V                    } → V Inserts
      Q.Insert(u)
  while Q ≠ ϕ
      u = Q.getMin()  → V Extracts
      for each neighbor v ∈ G.Adj[u]
          if u.d + w(u,v) < v.d  → E checks
              v.d = u.d + w(u,v)
              v.π = u
```



	Heap	Unsorted Array
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(n)$
DecreaseKey	$O(\log n)$	$O(1)$

Dijkstra's Algorithm - Analysis

Dijkstra(G, s)
  for each vertex $u \in G.V$
    $u.d = \infty, u.\pi = Null$ → $\Theta(V)$
  $s.d = 0$
  // Initialize a data structure for the vertices
  $Q = \phi$ // Priority queue
  for each vertex $u \in G.V$
    $Q.Insert(u)$ → V Inserts
  while $Q \neq \phi$
    $u = Q.getMin()$ → V Extracts
    for each neighbor $v \in G.Adj[u]$
      if $u.d + w(u, v) < v.d$ → E checks
        $v.d = u.d + w(u, v)$
        $v.\pi = u$

	Heap	Unsorted Array
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(n)$
DecreaseKey	$O(\log n)$	$O(1)$

With a Heap

$$T(V, E) = O(V \log V) + O(V \log V) + O(E \log V)$$
$$= O(V \log V) + O(E \log V)$$

For a connected graph, number of edges is of the order of V i.e., $E = \Omega(V)$.

$$T(V, E) = O(E \log V)$$

Dijkstra's Algorithm - Analysis

```
Dijkstra(G, s)
  for each vertex u ∈ G.V
      u.d = ∞, u.π = Null          } → Θ(V)
  s.d = 0
  // Initialize a data structure for the vertices
  Q = φ // Priority queue
  for each vertex u ∈ G.V          } → V Inserts
      Q.Insert(u)
  while Q ≠ φ
      u = Q.getMin()               → V Extracts
      for each neighbor v ∈ G.Adj[u]
          if u.d + w(u,v) < v.d    → E checks
              v.d = u.d + w(u,v)
              v.π = u
```

With a Heap

$$T(V,E) = O(V \log V) + O(V \log V) + O(E \log V)$$
$$= O(V \log V) + O(E \log V)$$
$$= O(E \log V)$$, If connected graph

With an unsorted array

$$T(V,E) = O(V) + O(V^2) + O(E)$$
$$= O(V^2)$$ Can we write this?

	Heap	Unsorted Array
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(n)$
DecreaseKey	$O(\log n)$	$O(1)$

Dijkstra's Algorithm - Analysis

	Heap	Unsorted Array
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(n)$
DecreaseKey	$O(\log n)$	$O(1)$

With a Heap

$$T(V,E) = O(V \log V) + O(V \log V) + O(E \log V)$$
$$= O(V \log V) + O(E \log V)$$
$$= O(E \log V)$$, If connected graph

With an unsorted array

$$T(V,E) = O(V) + O(V^2) + O(E)$$
$$= O(V^2)$$ Can we write this?

For a dense graph, extreme case is complete graph and $E = \theta(V^2)$

With an unsorted array

$$T(V,E) = O(V^2)$$

With a Heap

$$T(V,E) = O(V^2 \log V)$$

For a sparse graph, $E = \theta(V)$

With an unsorted array

$$T(V,E) = O(V^2)$$

With a Heap

$$T(V,E) = O(V \log V)$$

Thank You