$G(V, E)$

① $V = \{A, B, C, D\}$

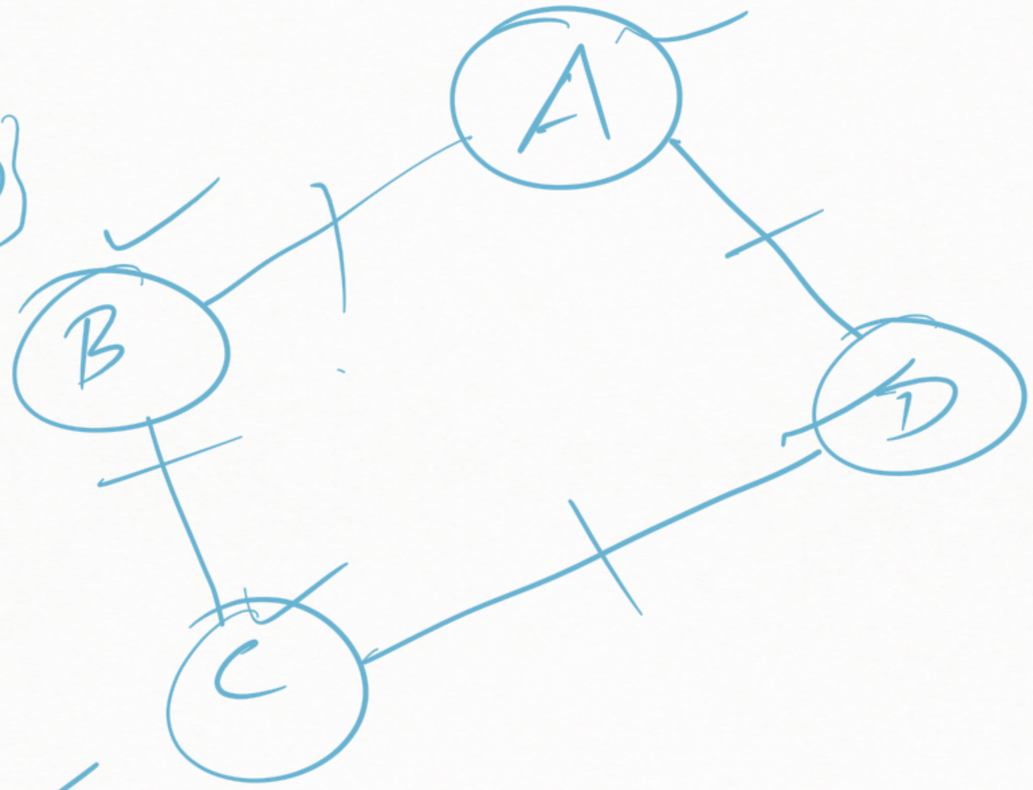
② $E = \{AB\ BC\ CD\ DA\}$

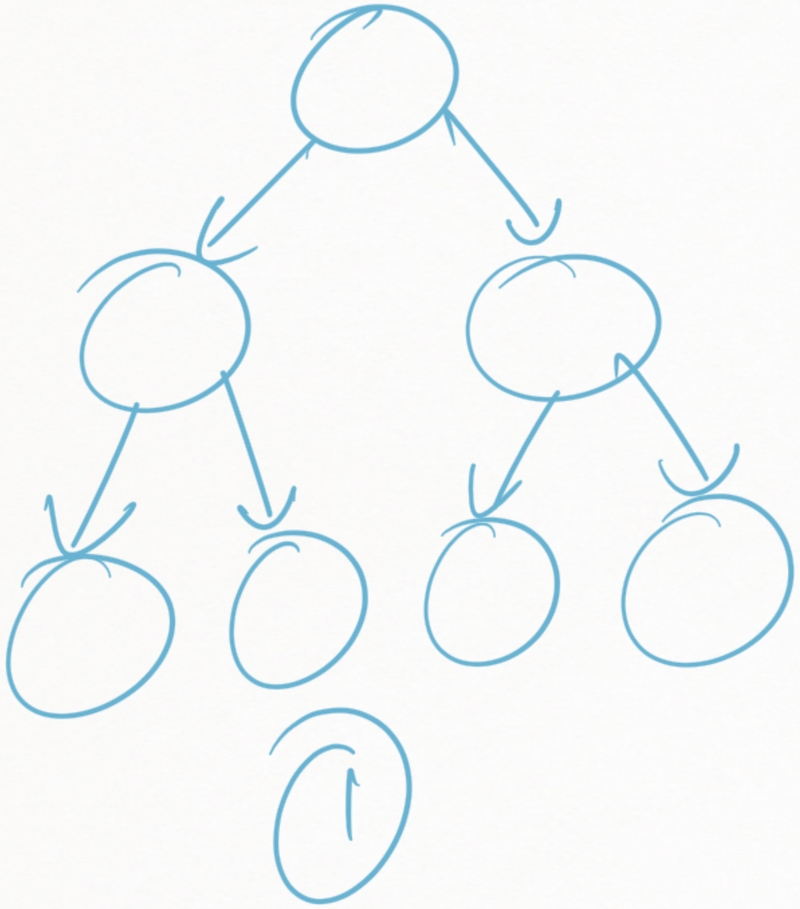
③ $|V| = 4$

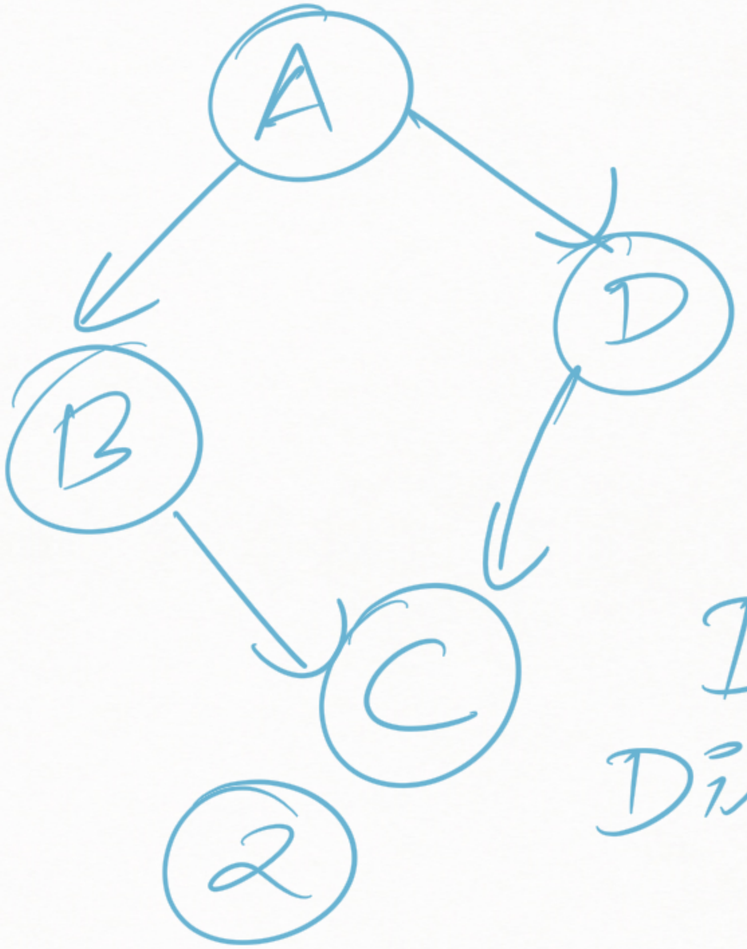
④ $|E| = 4$

⑤ D / N

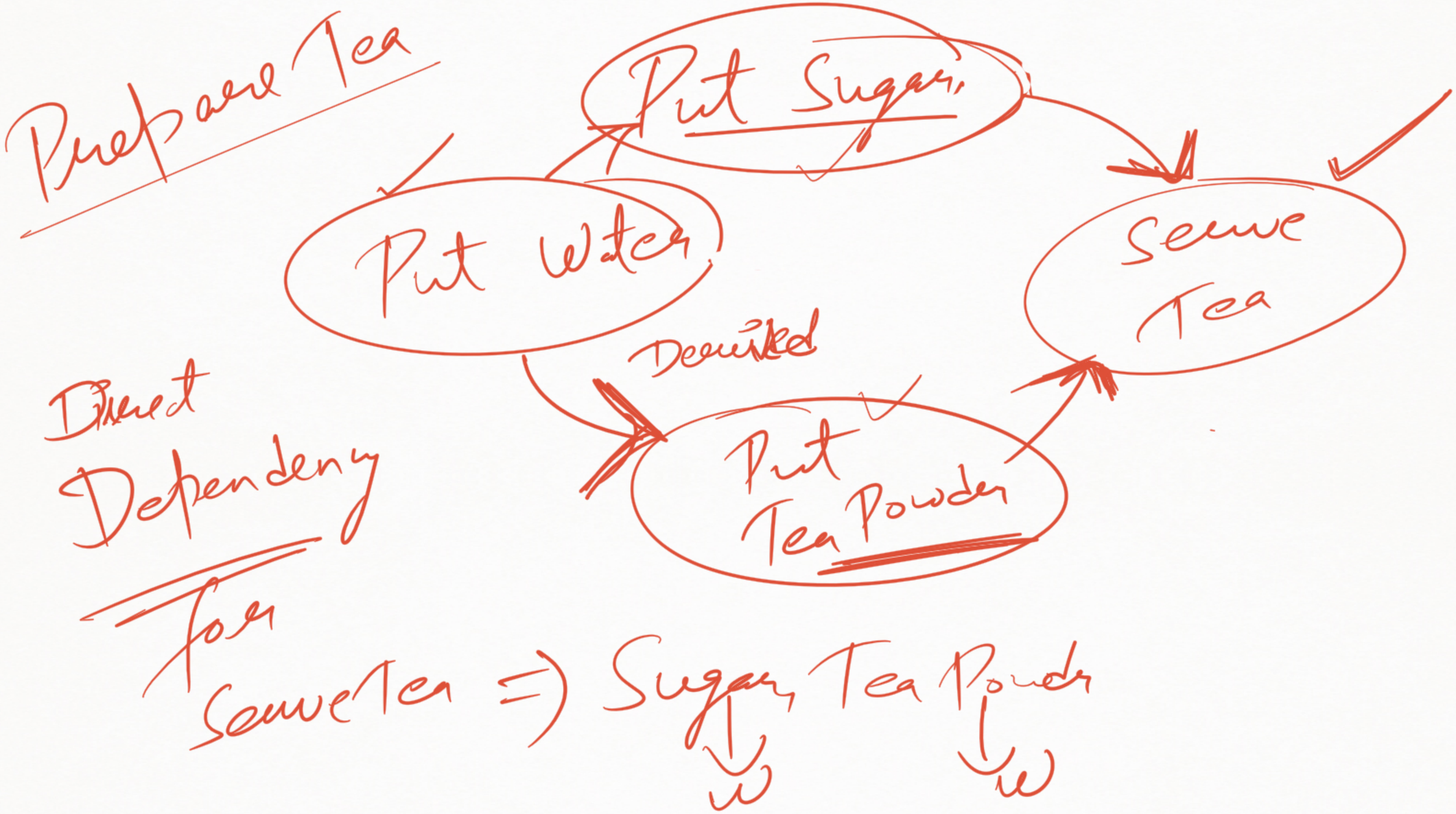
⑥ Č / A







DAG
Directed Acyclic
Graph


Prepare Tea
Put Sugar
Put Water
Serve Tea
Boiled
Put Tea Powder
Direct Dependency for Serve Tea => Sugar, Tea Powder

Graph algo runtime depends on |V| and |E|

→ $O(|V| + |E|)$ → linear time

→ $O(|V||E|)$, $O(|V|^2)$, $O(|V| \log |V| + |E|)$

→ ?) $O(|V|^{3/2})$ ①, $O(|E|)$ ②

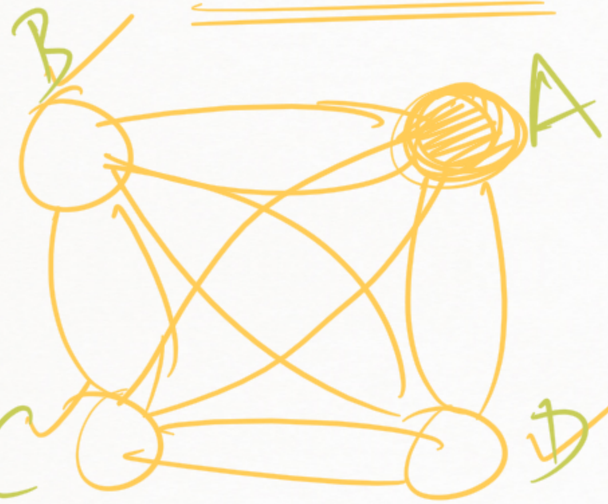
which one is faster?

I don't know.

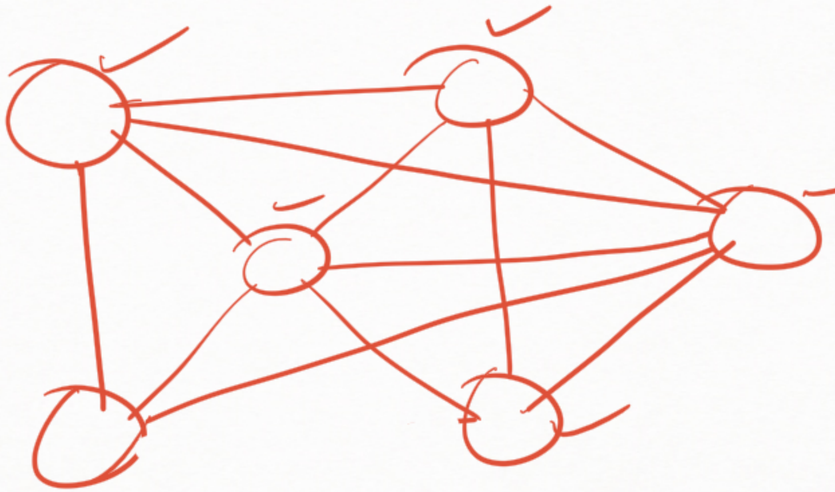
→ Depends on graph
→ Depends on density of graph
→ How many edges are present b/w those vertices.

Dense Graphs

$$|E| \simeq |V|^2$$



Almost all of vertices are connected to each other



So many edges

Which representation is good efficient for dense graphs?

Ans => ② Adjacency Matrix

① Edge List

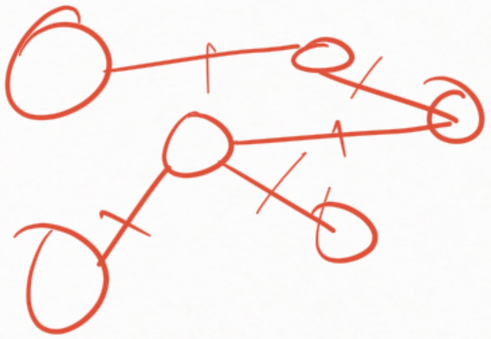
② Adjacency Matrix

③ Adjacency List

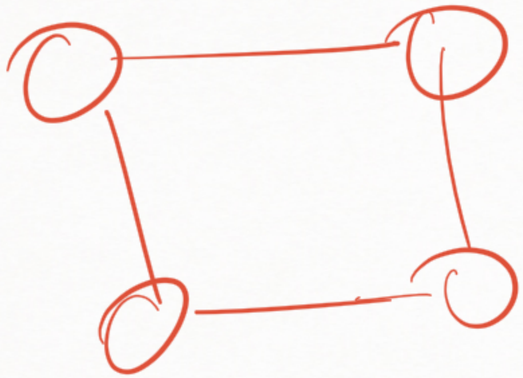
Sparse Graphs

$|E| \simeq |V|$

Each Vertex has only few edges



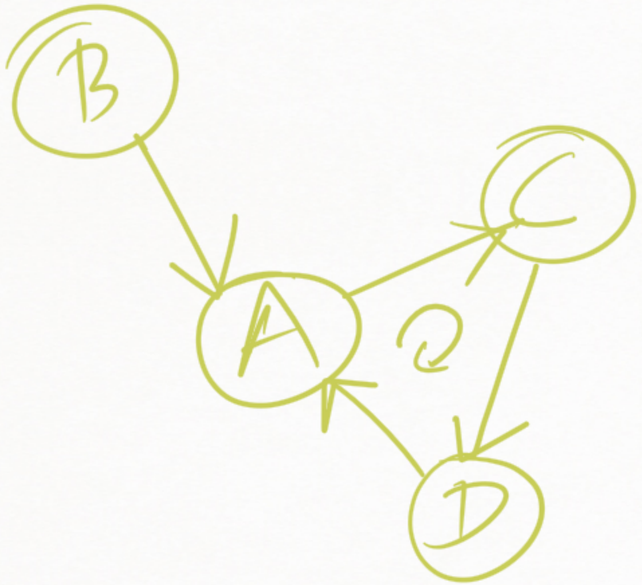
$5 \simeq 6$



$4 \simeq 4$

Exploring Graphs

Whether or not vertex of a graph can be reachable from another vertex



Can we reach B from D // A/C

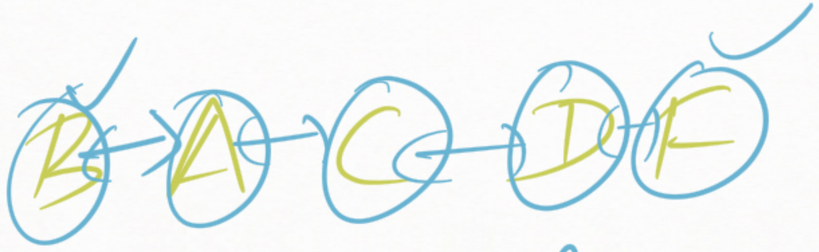
=) No

Application

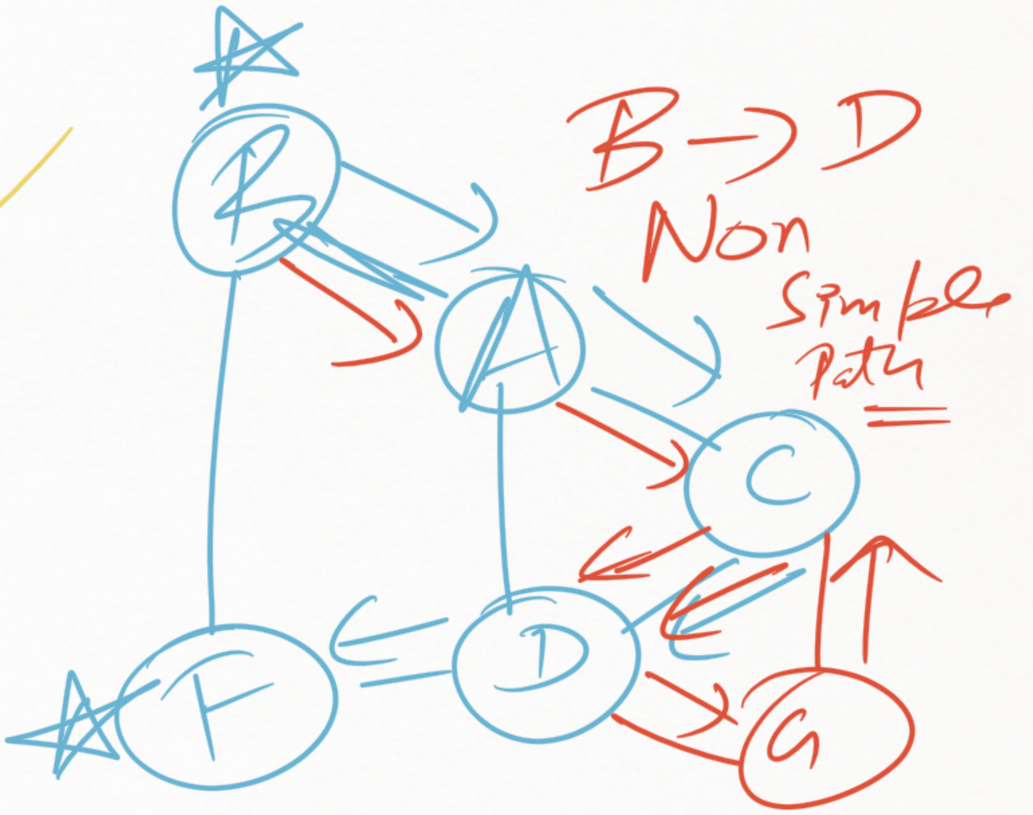
1. Find routes (path to destination)
2. Ensuring Connectivity
3. Solving puzzles & mazes

Path: In $G(V, E)$ a sequence of vertices where each adjacent vertex is dependent on edge

path =>



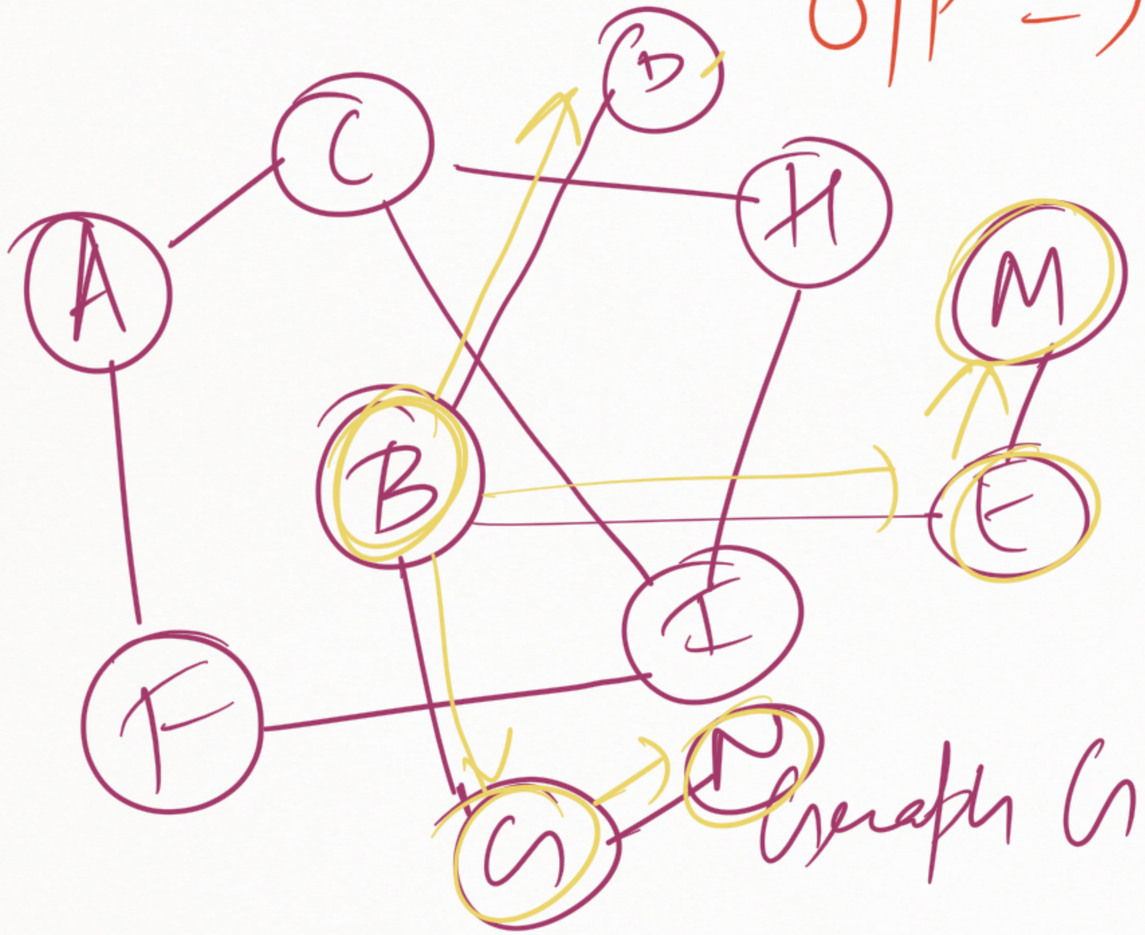
Simple path: No Vertex Repeats



Reachability:

I/p => Graph G & Vertex s

O/p => Collection of vertices "V" of G, so that there is a path from s to V.



Graph G(V, E)

I/p => G, B

O/p => D, E
G', M, N

1 Vertex to another

DFO =) Depth First Order

1 2 3 4 7 6 5 0

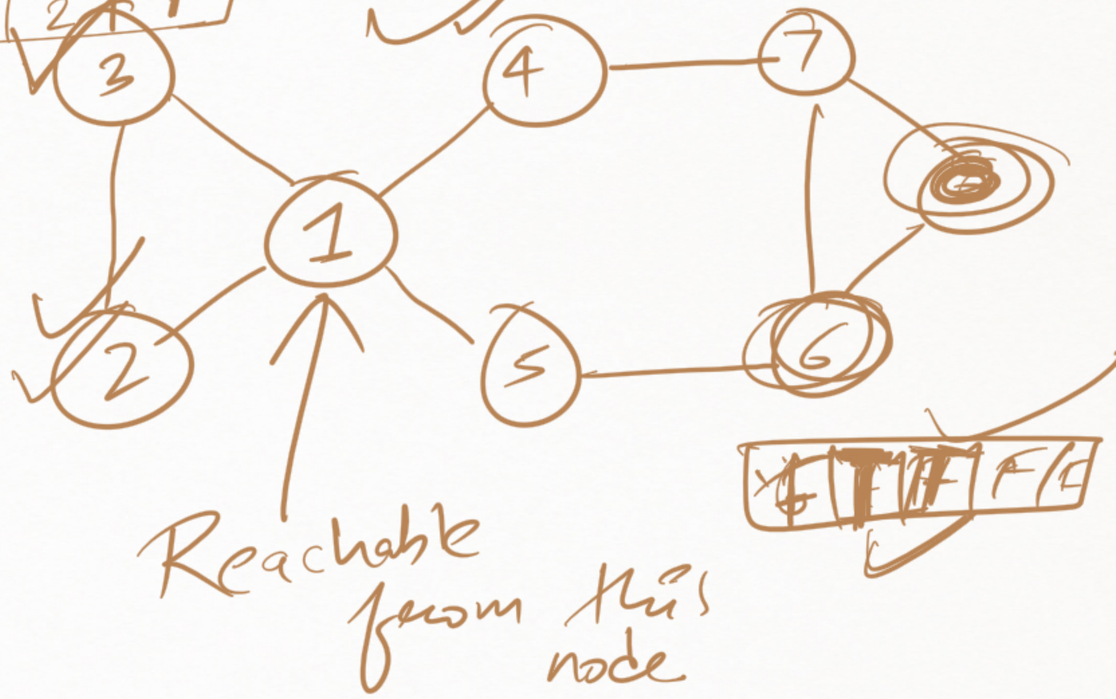
Visited is an bool array which contains info if some vertex is visited.

Explore(v) DFO(v)

```
visited(v) ← true
for (v, w) ∈ E:
  if not visited(w):
    Explore(w)
```

return

Need adjacency list representation!

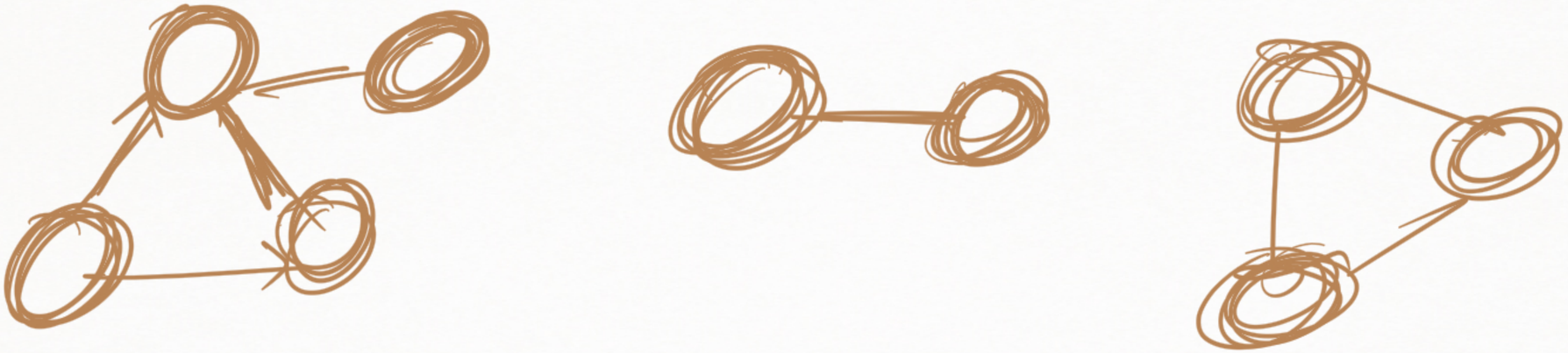


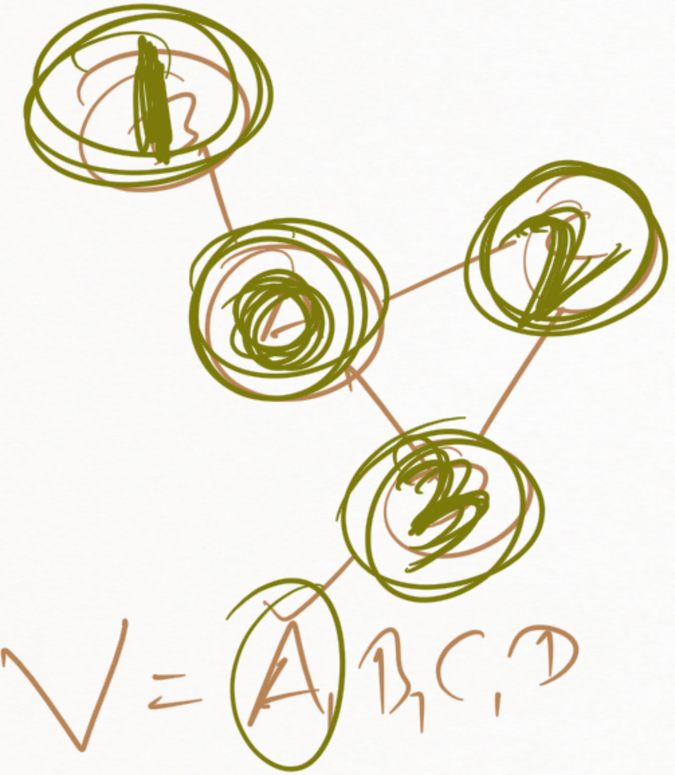
Reachable from this node

Depth First Search

=) Find all vertices of G.

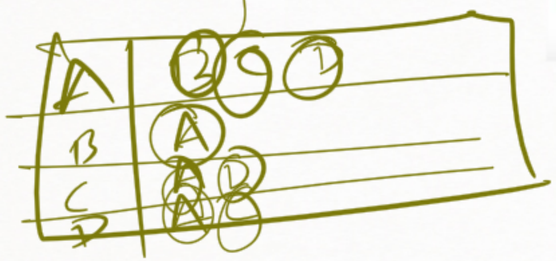
→ Mark everything as not visited





$V = \{A, B, C, D\}$

$E = \{AB, AC, AD, DC\}$



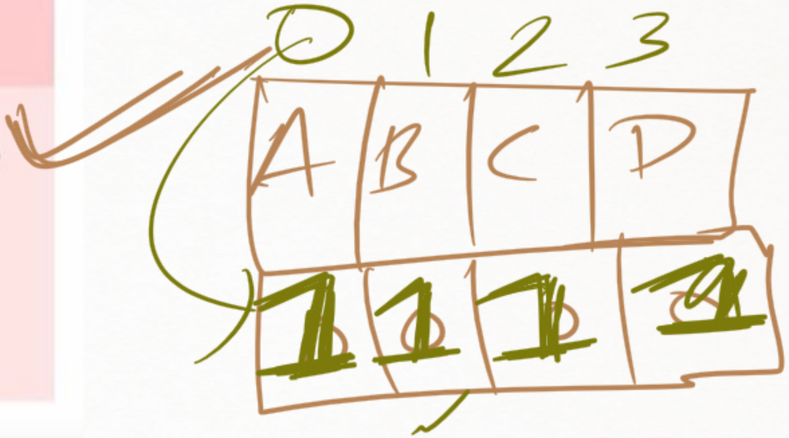
Explore(v)

```
visited(v) ← true
for (v, w) ∈ E:
  if not visited(w):
    Explore(w)
```

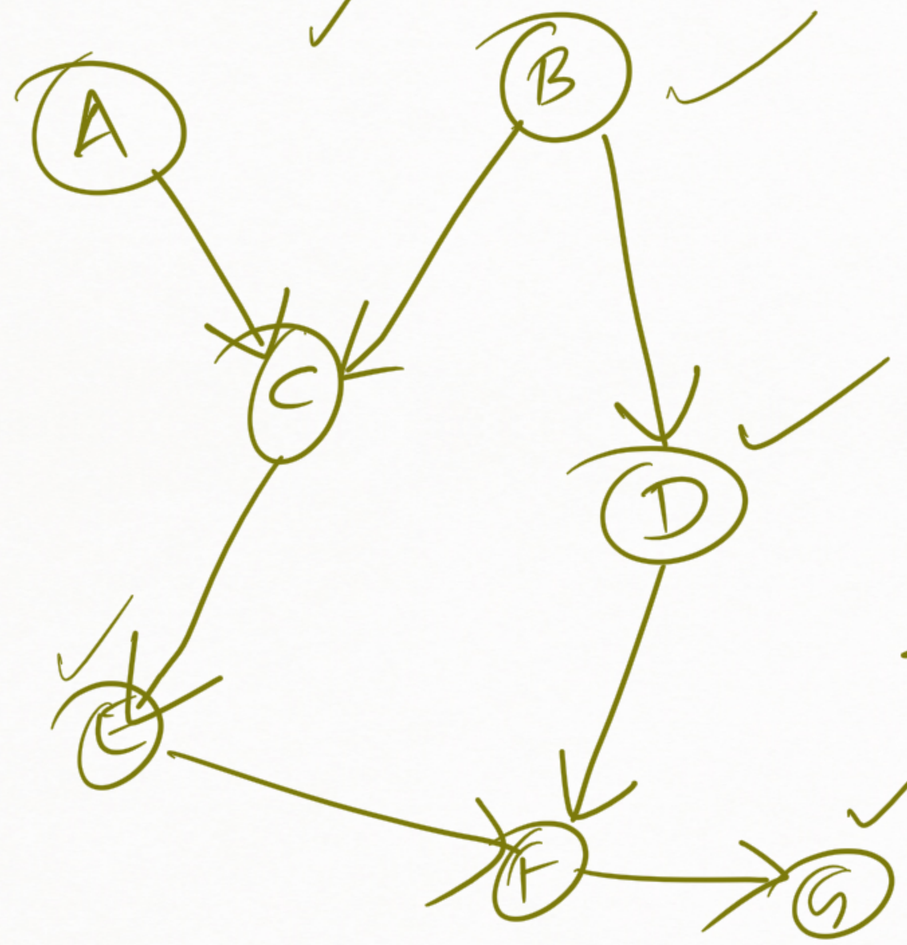
Need adjacency list representation!

DFS(G)

```
for all v ∈ V:    mark v unvisited
for v ∈ V:
  if not visited(v):
    Explore(v)
```



Topological Sorting



→ We take dependency into Account

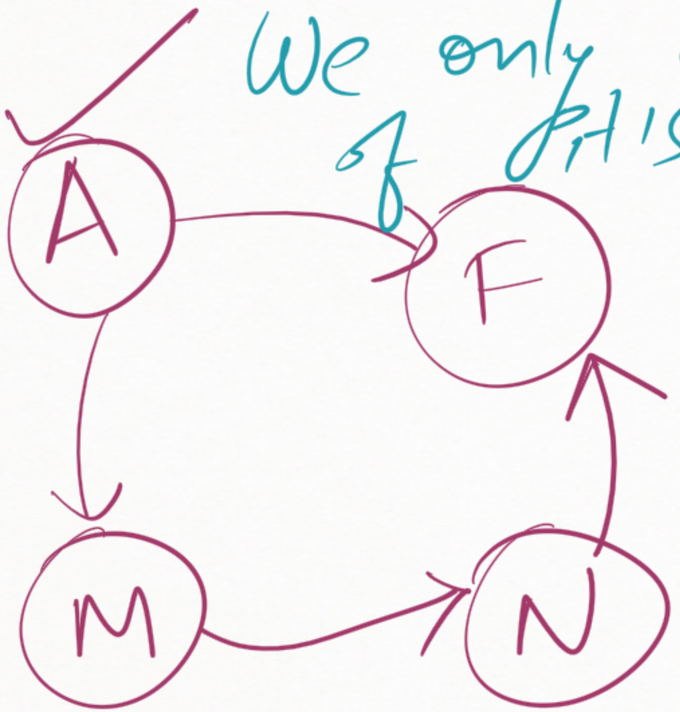
→ Mark everything unvisited

→ Many T.S orders for this graph

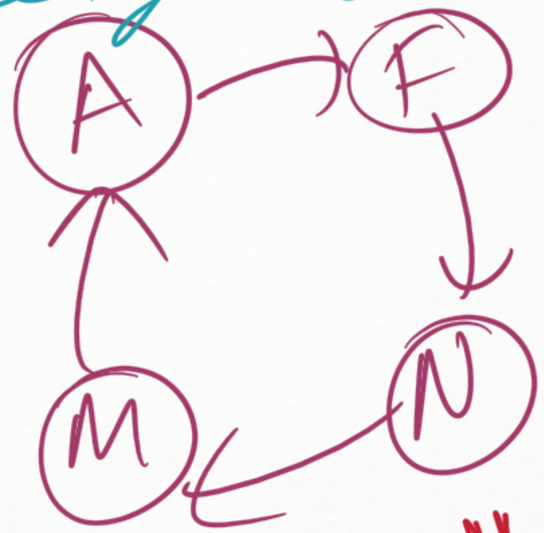
→ A B C E D F G

→ B D A C E F G

We only cover a node if all of its dependency are visited



T.S => AMNF



T.S =>

Cyclic

Start with a node which does not have dependency


0