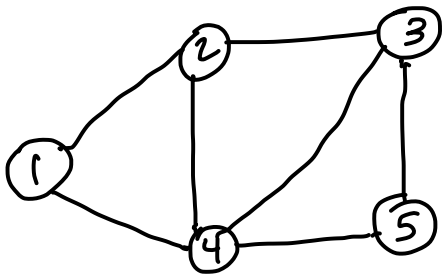
GRAPHS

① Adjacency Matrix } Most Popular
② Adjacency List



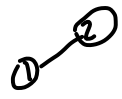
→ Adjacency Matrix

n x n matrix = 5 x 5 → # of vertices

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

$O(n^2)$
preferred when graph is dense: $E = O(n^2)$

* no loop at all = diagonals is 0's
* this is undirected graph so is 1 to 2, 2 to 1

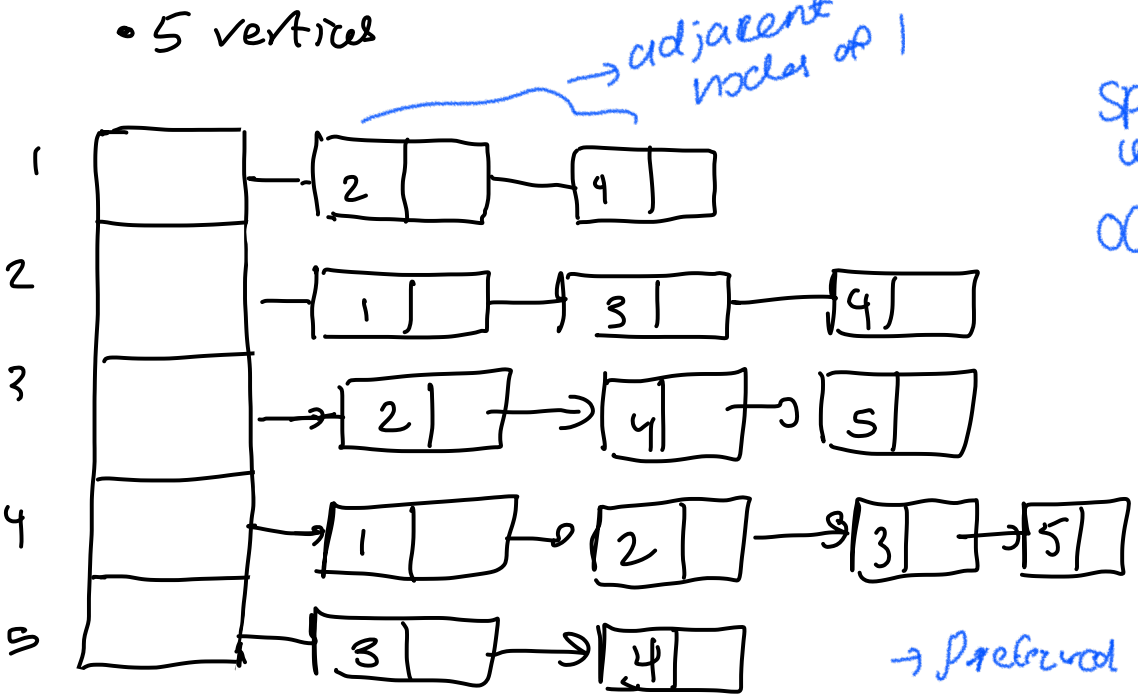


Adjacency Matrix:
A matrix A[n][n] when n = # of vertices → quickly determine edge between 2 vertices

$$a[i][j] = \begin{cases} 1 & \text{if } i \text{ \& } j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases}$$

→ Adjacency List

- 5 vertices

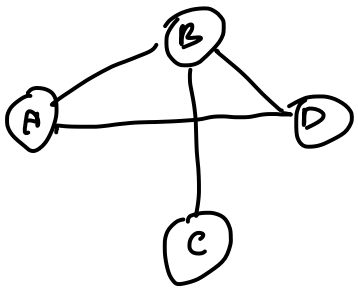


Space comp:
$O(n + 2e)$
↓
2 x edge

→ Preferred when for sparse graph: $E = O(V)$

→ Undirected Graph

- edges have no direction
- order of vertics is not important



V(Graph 1) = {A, B, C, D}

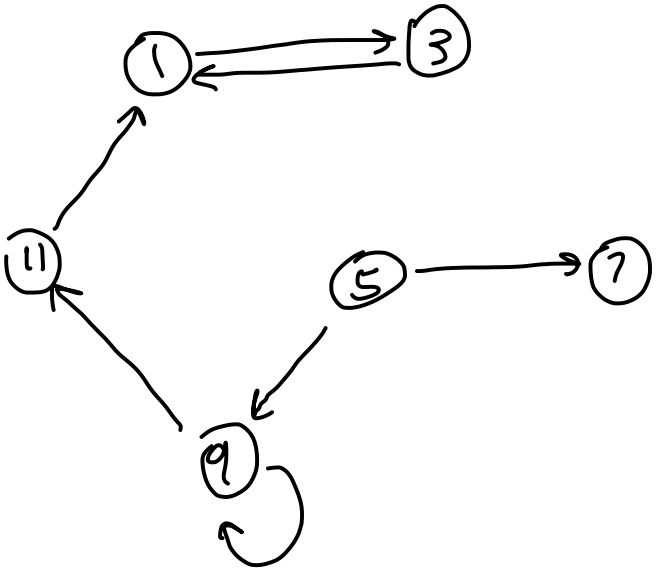
E(Graph 1) = {(A, B), (A, D), (B, C), (B, D)}

→ Directed Graph

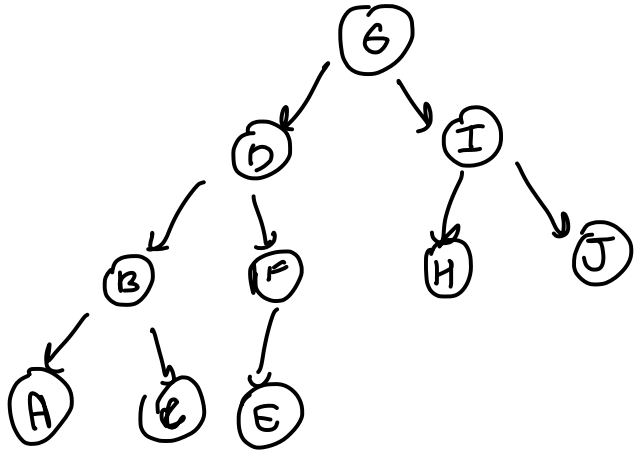
- order matters of vertices

V = {1, 3, 5, 7, 9, 11}

E = {(1, 3), (3, 1), (5, 7), (5, 9), (9, 11), (9, 9), (11, 1)}



→ Trees are a special type of graphs

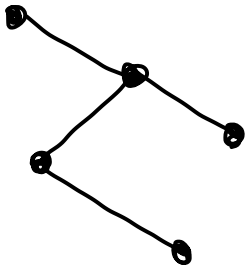


V = {A, B, C, D, E, F, G, H, I, J}

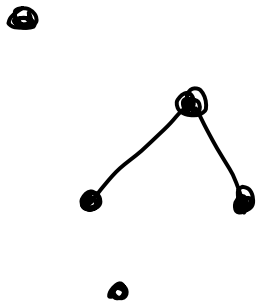
E = {(G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

→ **Connected Graph**

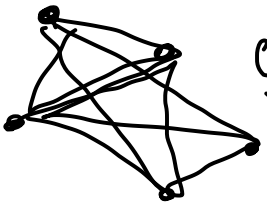
- each pair of distinct vertices has a path between them



Connected



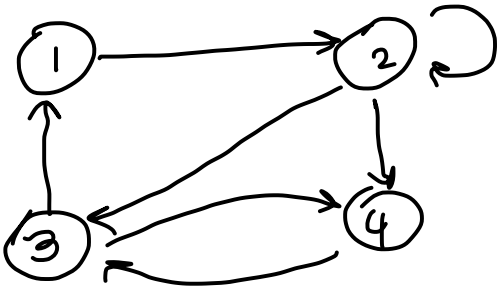
Disconnected



complete

→ **Path**: sequence of vertices that connects 2 nodes in graph

→ **Length**: length of a path is # of edges in the path



path from 1 to 4:

<1, 2, 3, 4>

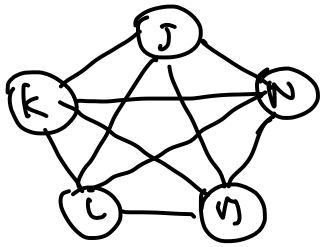
→ **Simple path**: pasas through vertex only once

→ **Cycle**: a path that begins + ends at same vertex

→ **Multigraph**: allows multiple edges between vertices

→ **weighted graph**: numeric values

→ # of edges in a complete undirected graph:



$E = V * (V-1)/2 \Rightarrow 5 * (5-1)/2$

or $O(V^2)$ $\quad 5 * 2 = \underline{10}$

→ # of edges in a complete directed graph: $E = V^*(V-1)$

Graph Traversals

1) BFS (level-order)

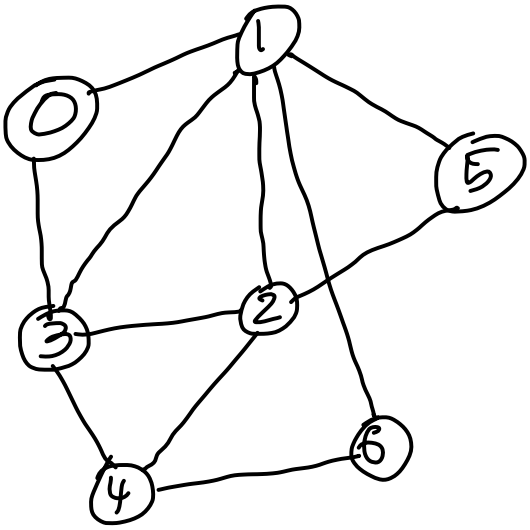
2) DFS

→ BFS (level-order)

- can take any node as start (let's say 0)
- Queue → any order

- Result:

0 1 3 2 5 6 4



1) visit vertices as close to root n

2) then all adj vertices + unvisited vertices of ↓

0, 1, 3, 2, 5, 6

↓

unvisited

3) of 3

~~2~~, ~~0~~, ~~1~~, 4

4) of 2: 1, 3, 4, 5

↓

all visited, nothing to insert

→ DFS

- Stack
- any node as root node, suppose 0
- 1, 3 → adj vertices of 0

↳ take only one of those initially

so take 1

= look at adj vertices: pick one

5, 2, 3, 6

4, 2, 0

4, 3, 4, 5

4, 3, 4, 0

↳ none

now backtrack

pop

pop & backtrack

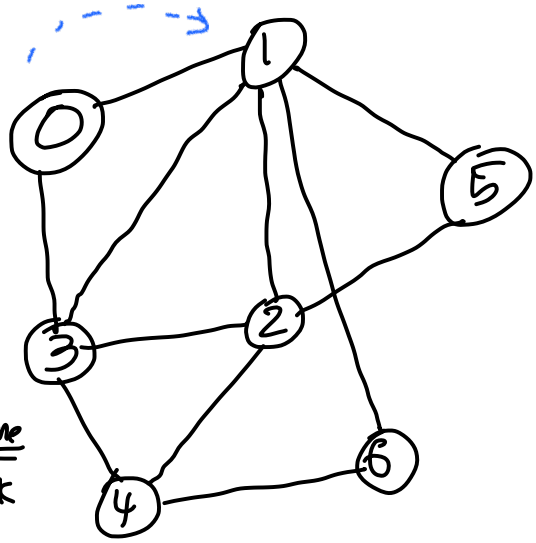
has no unvisited so pop

5 is unvisited, pop 5 cuz no unvisited, backtrack to 2

no unvisited

no unvisited

no unvisited



Result: 0 1 3 2 4 6 5

→ Code for DFS

```
Function DFS(graph, start):
    stack = empty stack
    visited = empty set

    stack.push(start)

    while(stack != empty) {
        vertex = stack.pop()
        if vertex not visited{
            visited.add(vertex)
            print(vertex)

            for each neighbor in graph[vertex]:
                if neighbor is not in visited:
                    stack.push(neighbor)
    }
```

Total:
$O(V + E)$

Breadth-First Search

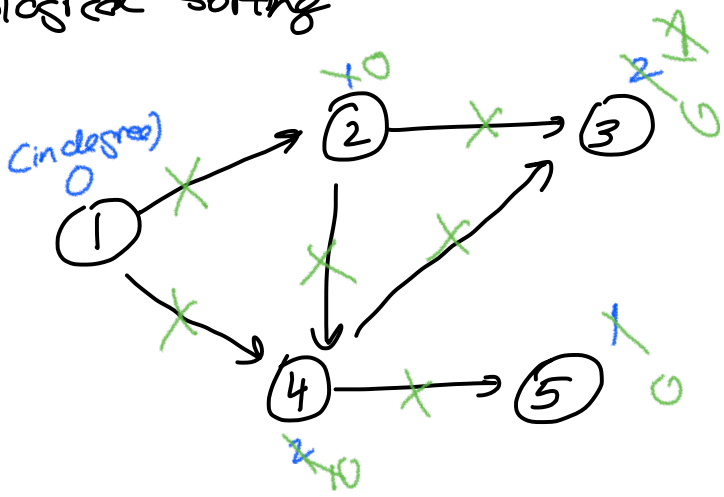
procedure *BreadthFirstSearch*(**graph** G, **vertex** v):
{Breadth-first search of G starting at v}
    foreach vertex w **in** G **do** Encountered(w) ← **false**
    {Q is a queue storing encountered but unvisited vertices}
    Q ← *MakeEmptyQueue*( )
    Encountered(v) ← **true**
    Enqueue(v, Q)
    until *IsEmptyQueue*(Q) **do**
        {Process the next vertex}
        w ← *Dequeue*(Q)
        Visit(w)
        foreach neighbor w' **of** w **do**
            if not Encountered(w') **then**
                Encountered(w') ← **true**
                Enqueue(w', Q)

Algorithm 12.1 Breadth-first search. The procedure *Visit* is to be executed on each vertex reachable from v.

Topological Sorting

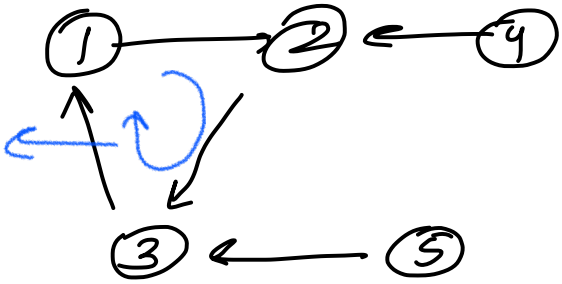
- Graph must be directed acyclic graph
- every DAG will have ≥1 topological sorting

→ in degree: coming to that node

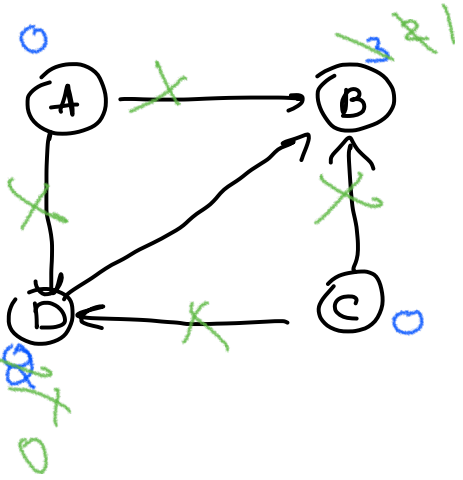


- choose vertex with indegree of 0 & delete node 1 & all outer arrows
- then choose node 2, delete..

2 possible { 1 2 4 3 5 or
1 2 4 5 3



cycle
so
can't find
topological
ordering



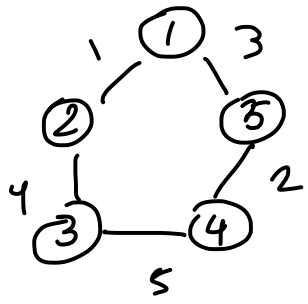
Case 1: select A

A C D B

Case 2: Select C

C A D B

Spanning Tree



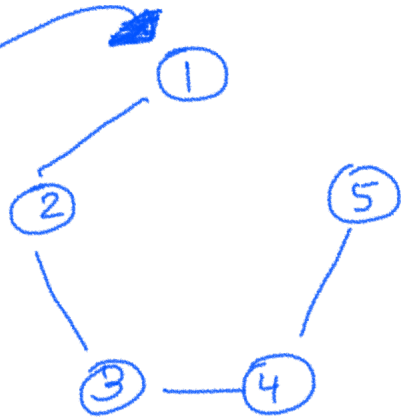
→ Spanning Tree

$G(V, E)$

$G'(V', E')$

$V' = V$ ✓ $\quad E' \subset E$ or $E' = |V| - 1$ → 4

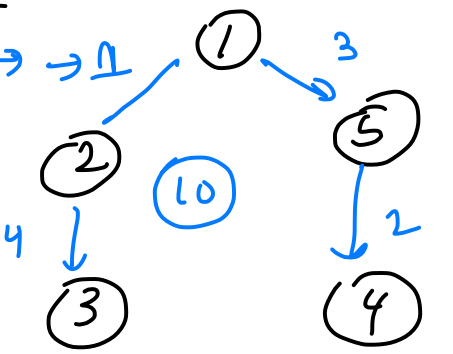
5 ✓



Minimum ST

→ To construct start with min weight? → → 1

→ no cycle & no disconnect for spanning tree



4 edges because $E' = |V| - 1$

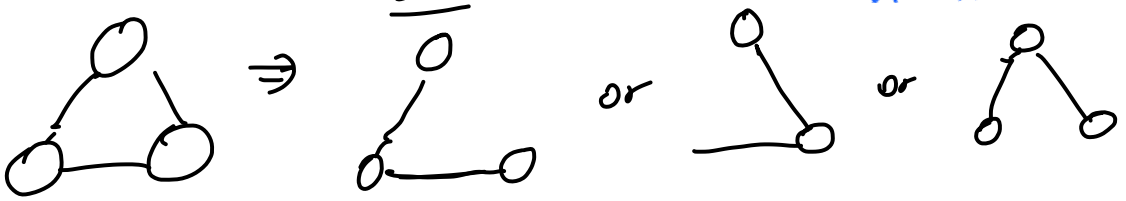
→ Properties of ST

1) remove 1 edge = disconnected
2) adding 1 edge = creates a loop
3) If each edge has a diff weight → only 1 unique MST
4) a complete undirected graph can have n^{n-2} no. of ST
5) every connected & undirected graph has at least one ST
6) disconnected graph = no ST
7) from a complete graph by removing max $(e - n + 1)$ edges we can construct a ST

each vertex is connected to another vertex

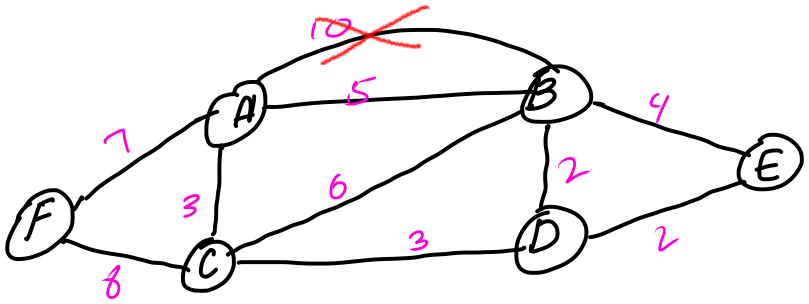
n = # of vertices

4 ⇒ STs



$(e - n + 1) = 3 - 3 + 1 = 1$ → max can remove 1 edge

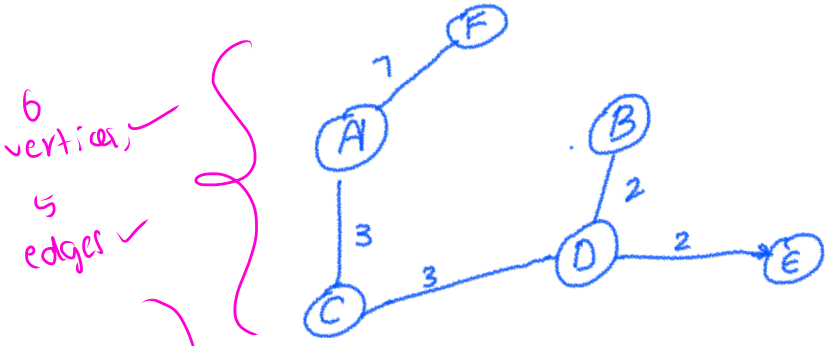
Kruskal's Algorithm To Find MST



1) remove all loops + parallel edges ($A \to B$)
   ↳ keep the edge with min weight

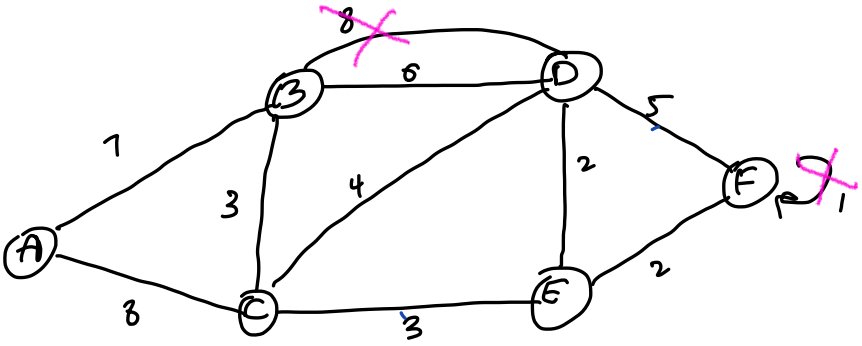
2) Increasing weight
   - → BD = 2 } min edge weight
   - → DE = 2
   - → CD = 3
   - → AC = 3
   - ✗ BE = 4 → can't connect because forms a cycle
   - ✗ AB = 5
   - ✗ BC = 6
   - → AF = 7
   - ✗ FC = 8 → cycle

MST



* no cycle
* same # vertices, n-1 edges
* total weight would be minimum

→ Prim's Algorithm

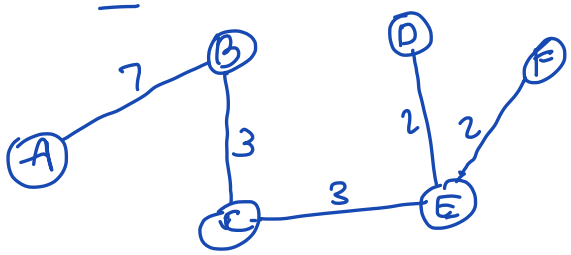


1) remove all loops & parallel edges
   ↳ min weight should be kept

2) choose any node as root node

→ Suppose A is chosen
→ check all outgoing edges from A
→ choose edge with min weight
→ now ~~check~~ check edges from B & A not just B
   ↳ 6, 3, 8
   ↳ pick min
→ check B, A, C → ~~3~~ 4, 3
   6 ↵ ↳ done ✓

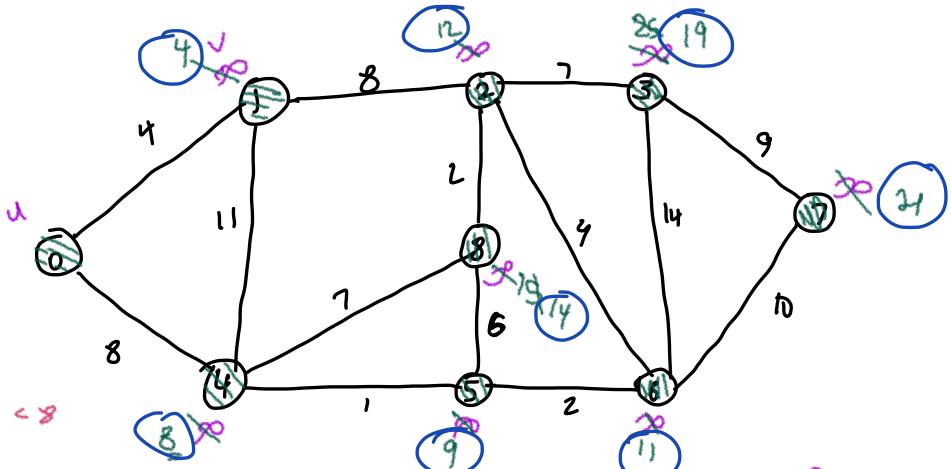
MST



E' # of edges: V − 1 = 6 − 1 = 5

V' # of vertices: same as V

Dijkstra Algorithm



- distance from source vertex to Source vertex = 0

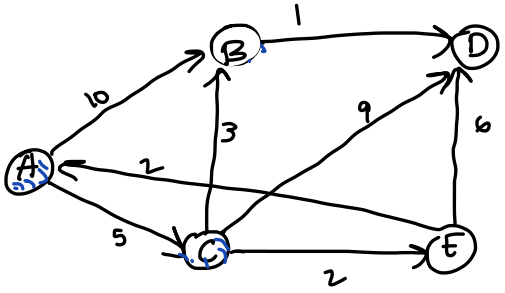
if (d(u) + c(u,v)) < d(v)) {distance(v) = d(u) + c(u,v)}

0 + 4 ∞

4 < 8

- after done with source vertex, select vertex with shortest distance (vertex 1)
- once a vertex is visited, don't update its distance

SV = A



Source vertex must be first

Visited	A	B	C	D	E
A	0	∞	∞	∞	∞
C	0	10	5	∞	∞
E	0	8		14	7
B	0	8		13	
D				9	

→ Shortest Distance from A to D = 9

→ Find Path A to D

DBCA → so reverse:

ACBD = 9

→ start at last so D
so 9 then go backward so it's B which is diff from 9 so see which node was selected so B = 8 then backwards up row & it's 8 = 8 so just go backwards normally

A C B D
5 + 3 + 1 = 9

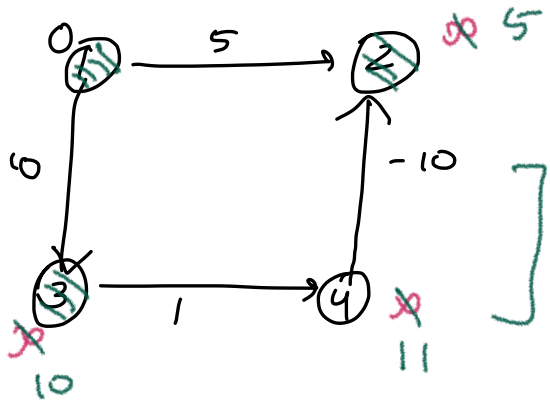
- → Path A to B

⇒ 8 (distance)

Pointer to 8

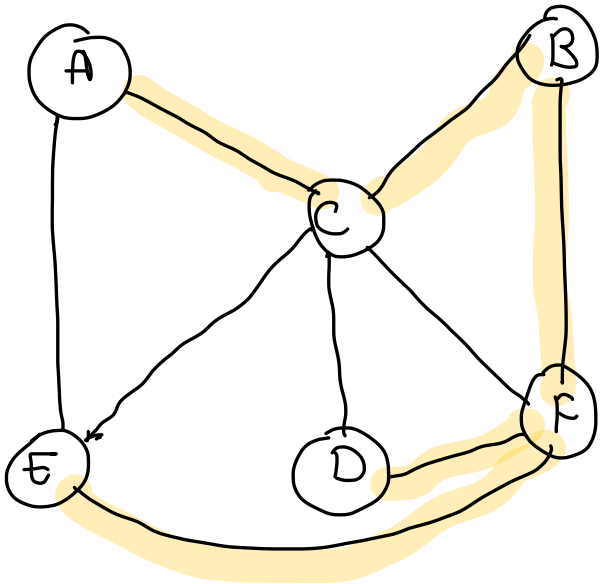
BCA → ACB

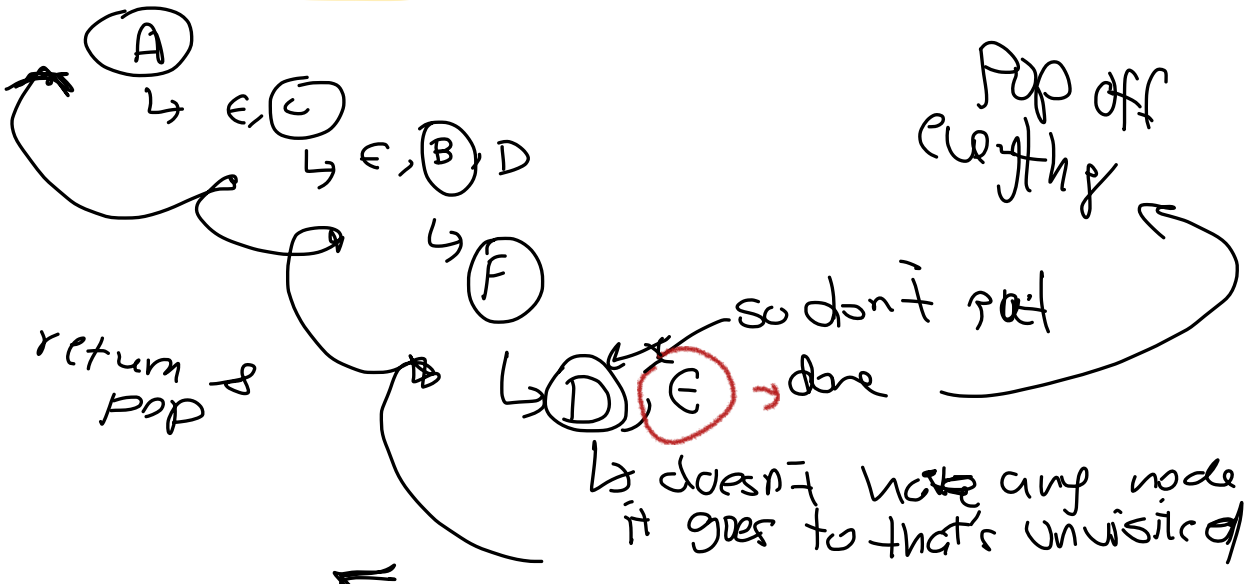
5 + 3 ≠ 8



DFS

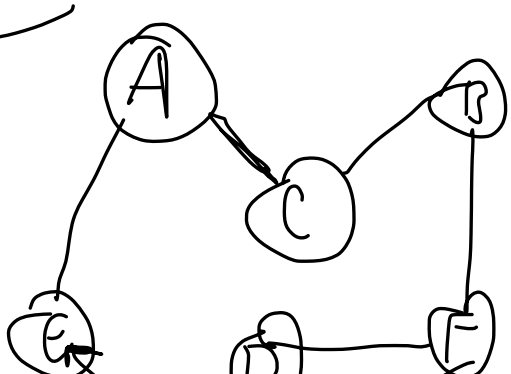
→call DFS on A

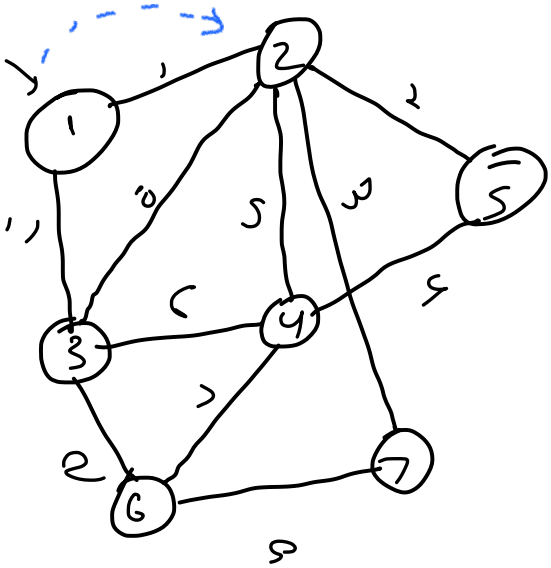


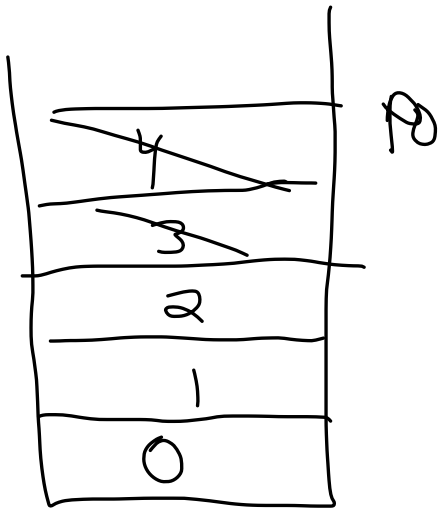


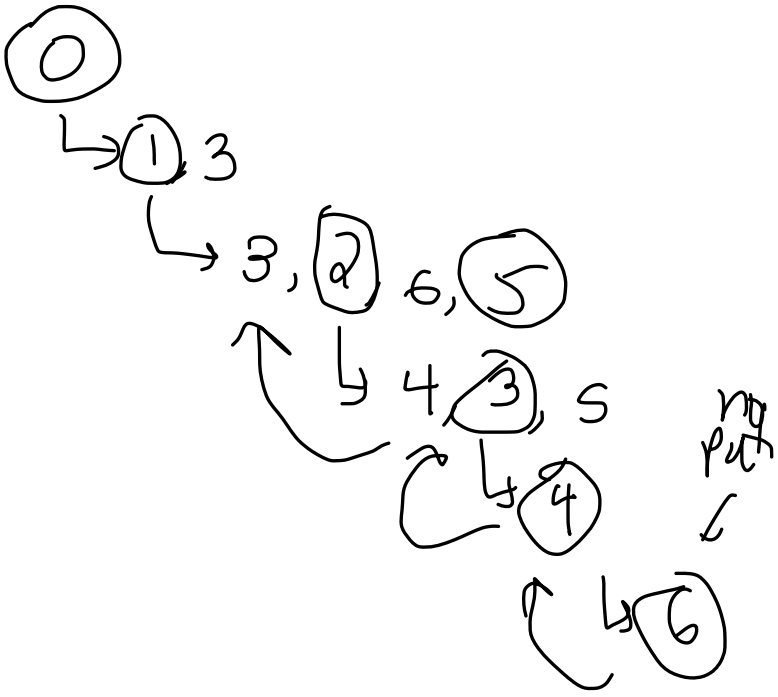
↳ doesn't have any node that it goes to that's unvisited

⇒ [PATH]

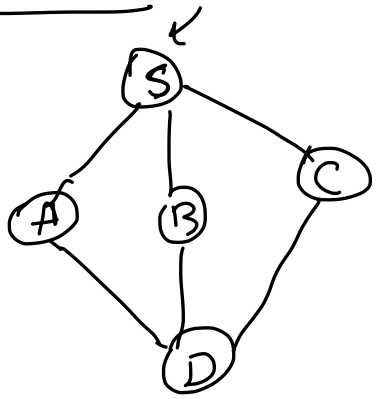



1
2
3
4
5
6
7


4
3
2
1
0


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

BFS

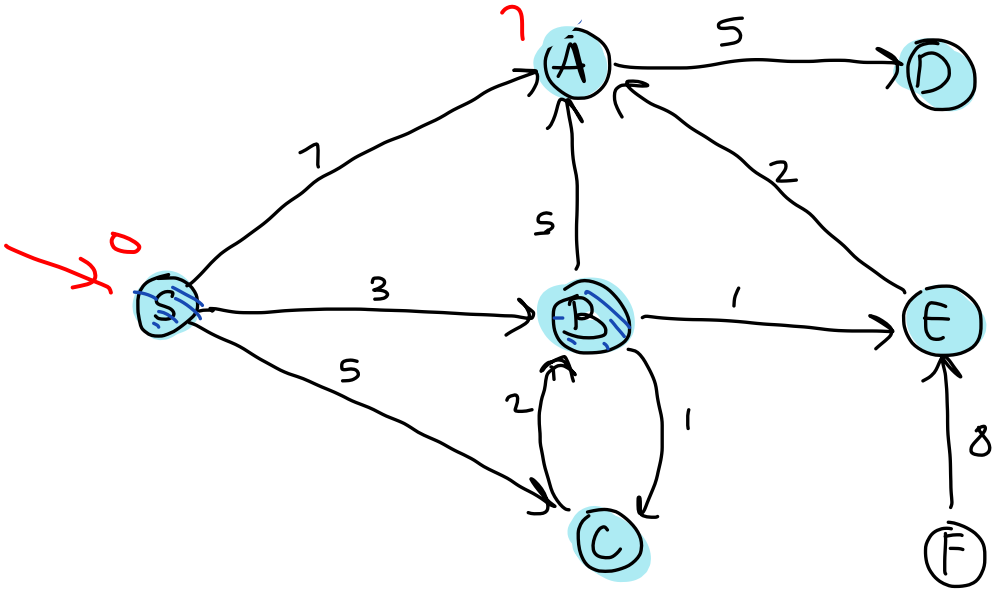


Queue

visited : S

↳ A

P
C
P
C
P
C
P
C
fork



unvisited

nodes: {S, A, B, C, D, E, F}

S: 0

A: ∞ ~~7~~ ~~6~~

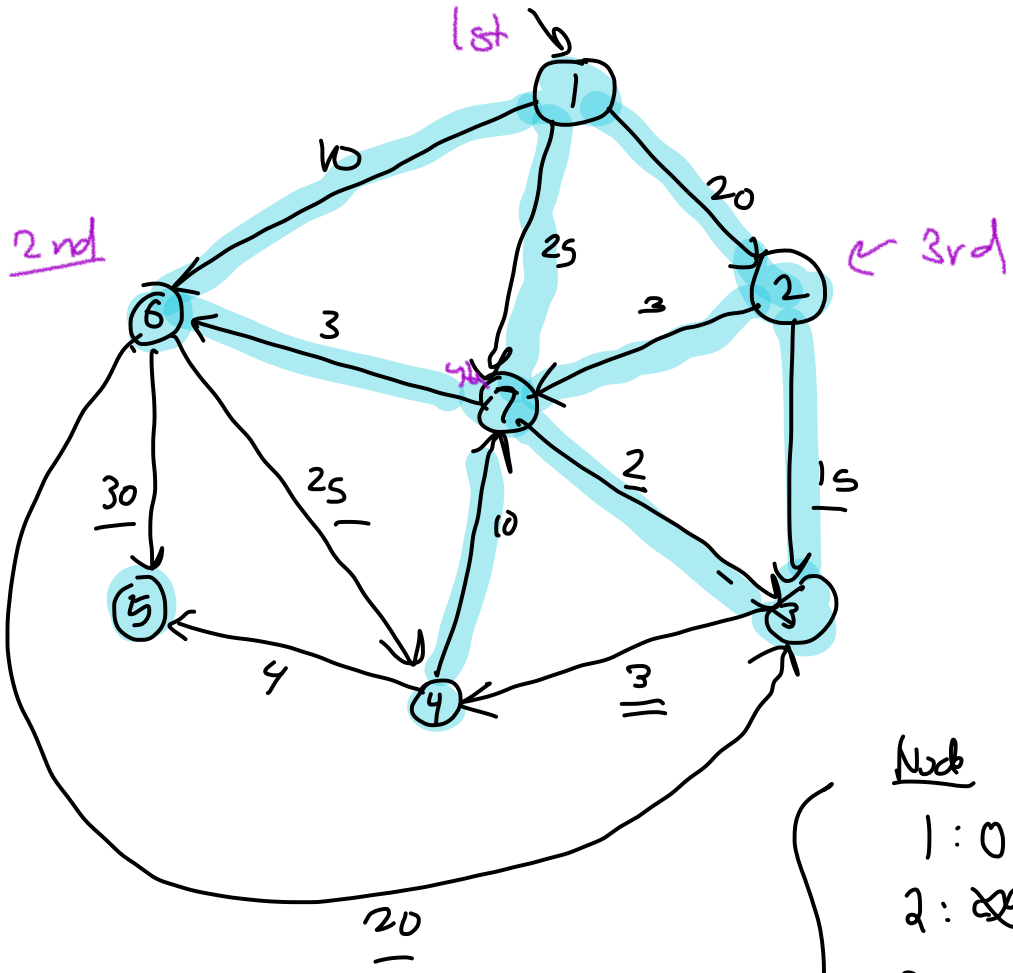
B: ~~∞~~ 3

C: ~~∞~~ ~~5~~ 4

D: ~~∞~~ 11

E: ~~∞~~ 4

F: ∞



Unvisited Nodes

{1, 2, 3, 4, 5, 6, 7}

Node

1 : 0

2 : ~~∞~~ 20

3 : ~~∞~~ ~~30~~ ~~35~~ 25

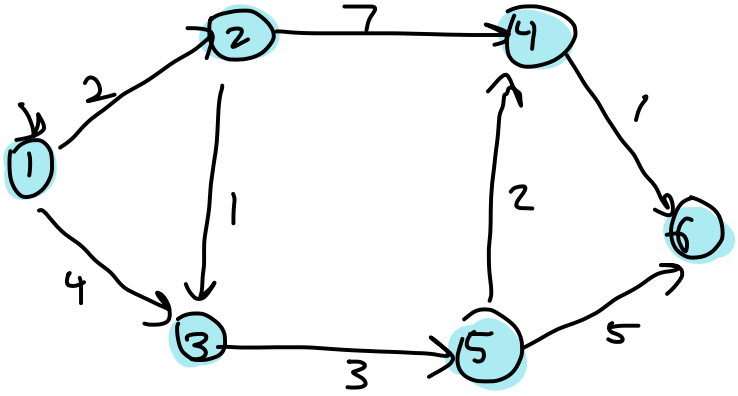
4 : ~~∞~~ ~~35~~ 28

5 : ~~∞~~ ~~40~~ 32

6 : ~~∞~~ 10

7 : ~~∞~~ ~~25~~ 23

	Shortest Path from 1	Cost
1	1	0
2	1 → 2	20 = 20
3	1 → 2 → 7 → 3	20 + 3 + 2 = 25
4	1 → 2 → 7 → 3 → 4	20 + 3 + 2 + 3 = 28
5	..... → 5	... + 4 = 32
6	1 → 6	10
7	1 → 2 → 7	20 + 3 = 23



Unvisited Nodes:

~~{1, 2, 3, 4, 5, 6}~~

1: 0
2: ~~∞~~ 2
3: ~~∞~~ ~~4~~ 3
4: ~~∞~~ ~~9~~ 8
5: ~~∞~~ 6
6: ~~∞~~ ~~11~~ 9

finally