* Shortest Path in an Undirected graph

Simple BFS and updating distance would work here

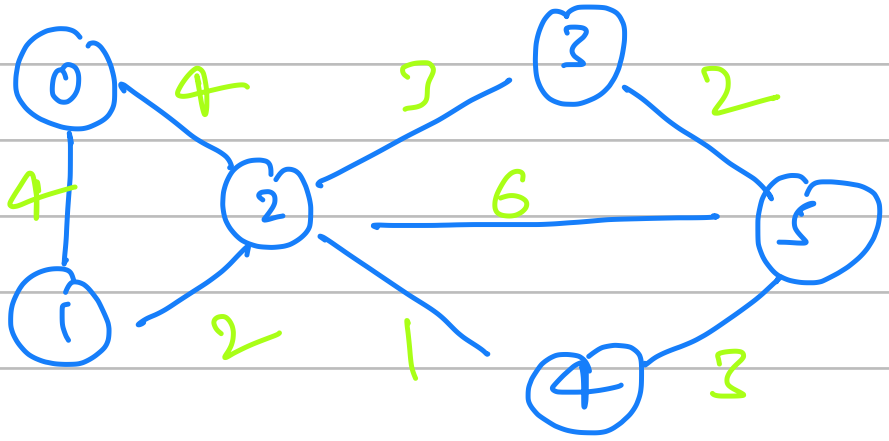
↳ This would not work for a DAG

↳ you will need topo-order

This is a perfect ()

* Dijkstra Algorithm

-> Undirected graph



Path from 0 — 1 is

0 → 2 → 1 Cost = 4 + 2 = 6

0 → 1 cost = 4

Cost = 4 + 3 + 2 + 6 + 2

(A) Statement

Given a source node find the min distance to every node

3 methods to do that

- Queue (bad)
- Priority Queue
- Set

(B) Priority Queue approach

Initial Configuration

- Create a min-heap data structure that stores ≡ {node, distance}
  - distance = criteria for min heap
- distance array ≡ [ ]
- Initialize the dist[src] = 0
- put that into min-heap = [[src, 0]]

→ Iteration

① take out the node from priority queue

node, dist = pq.pop()

for every neighbour of node:

if dist[node] + edgeweight is better:

dist[neighbour] = dist[node] + edge_weight-neighbor

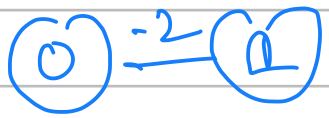
pq.add([neighbour, dist(neighbour)])

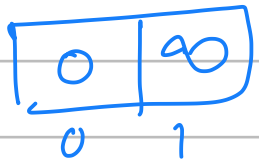
② Return the distance array

```
from heapq import heapify, heappush, heappop
class Solution:

    #Function to find the shortest distance of all the vertices
    #from the source vertex S.
    def dijkstra(self, V, adj, S):
        # Initial configuration
        pq = []
        distance = [float('inf')] * V
        distance[S] = 0
        heappush(pq, (0, S))

        # Iteration
        while len(pq) > 0:
            d, u = heappop(pq)
            for neighbor in adj[u]:
                v, edge_wieght = neighbor
                if distance[v] > distance[u] + edge_wieght:
                    distance[v] = distance[u] + edge_wieght
                    heappush(pq, (distance[v], v))
        return distance
```

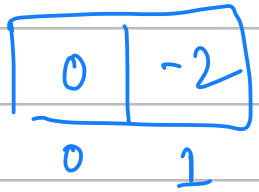
Note → This would not work for negative weights on cycle that results in negative

Example

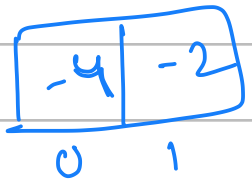




when going from 0 → 1 dist = -



when going from 1 → 0 dist = -2-2 = -4

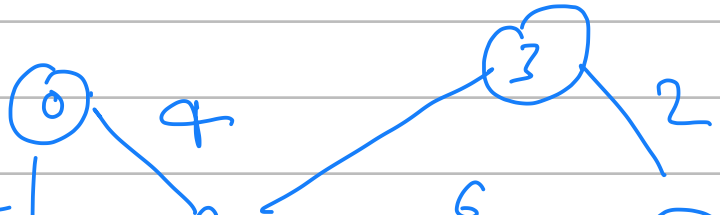


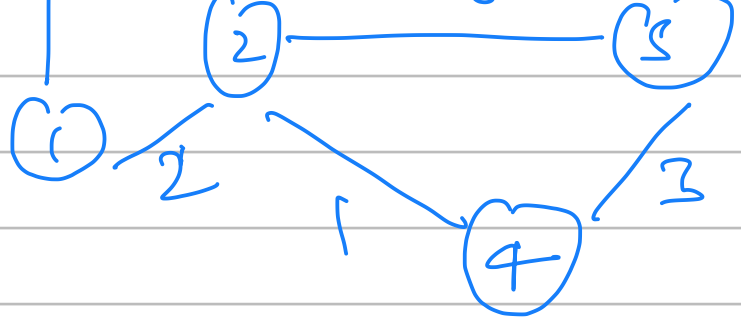
See the logic, Algorithm Stuck in a loop

(A) Dijkstra using Set data structure

→ Everything is same as priority queue except

Consider this snap shot





Distance = | 0 | 4 | 4 | 7 | 5 | 10 |

0 1 2 3 4 5

Set = { (10,5) (7,3) (5,4) }

5

(4) — checks for neighbor (5)

node current

it sees that someone has already reached (5) with a distance of 10 but it has better value of (5+3) = 8 < 10

remove the (10,5) entry from the set

and continue

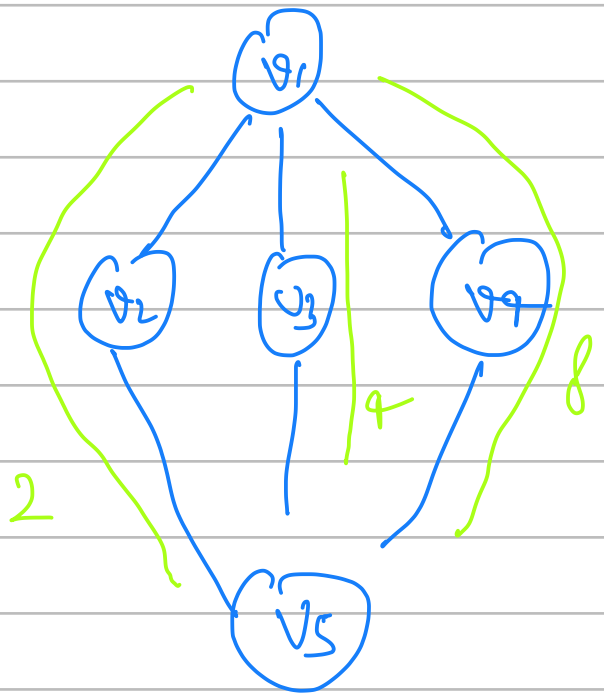
Intuition

→ Why a Priority Queue instead of a Queue

→ Queue will also work

* Simple using Queue and doing a BFS and updating distance as you encounter is just a brute force in which you consider all the paths

†



$(8, v_5)$
$(4, v_5)$
$(2, v_5)$

To reach v_5 →

v_5 can be reached in three ways and with simple queue all the paths will be explored

≡ To eliminate this Dijkstra said why not use priority queue and consider the path with min distance first

i.e. $(2, v_5)$ first which will

calculate all the paths with min time and will nullify future entries

≡ Using Set idea → In set if you found a better path remove the entries from set

Time complexity ≡ E·log V

worst case Scenario

* Tell the shortest path b/w src and target
  - Apply dijkstra but memorize where you come from
  - maintain a parent array

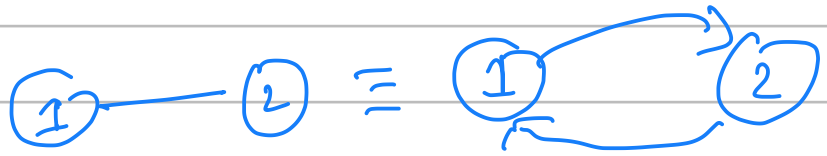
```python
class Solution:
    def shortestPath(self,n:int, m:int, edges:List[List[int]] )->List[int]:
        adj = []
        for i in range(n+1):
            adj.append([])
            # construct the adjacency list
        for egde in edges:
            u, v, w = egde
            adj[u].append([v, w])
            adj[v].append([u, w])
        # Initial configuraton
        distance = [float('inf')] * (n+1)
        parent = [-1] * (n+1)
        queue = []

        distance[1] = 0
        heappush(queue, (0, 1))

        # Iterate
        while len(queue) > 0:
            d, u = heappop(queue)
            for neighbor in adj[u]:
                v, w = neighbor
                if distance[v] > distance[u] + w:
                    distance[v] = distance[u] + w
                    heappush(queue, (distance[v], v))
                    parent[v] = u
        if distance[n] == float('inf'):
            return [-1]
        result = []
        p = n
        while p != -1:
            result.append(p)
            p = parent[p]
        result.append(distance[n])
        result.reverse()
        return result
```

(*) Bellman Ford Algorithm

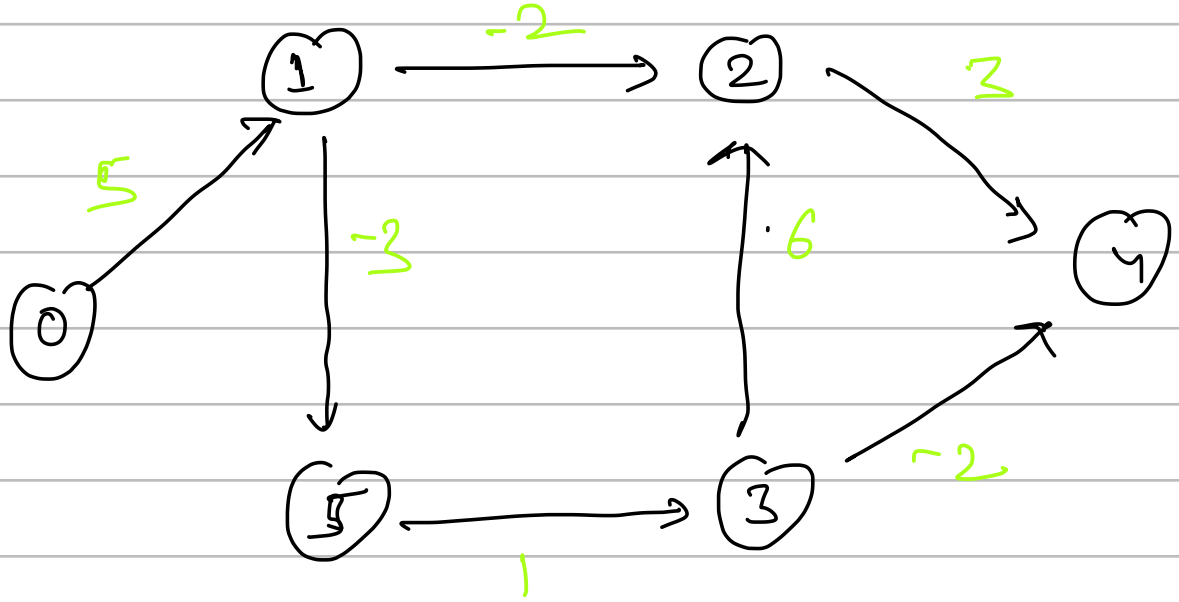
* Shortest path from src to all nodes
* Dijkstra fails with negative edges and negative cycles

* Applicable only on Directed graphs

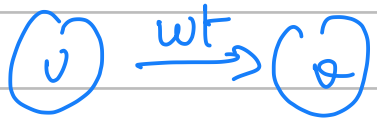
↳ to make it work on Undirected ≡ convert it to Directed



* Single source shortest path algo



(1) Relax all edges n-1 times sequentially

Relax ≡ 

if distance[u] + wt < distance[v]:

// got a better path

distance[v] = distance[u] + wt

(2) initial configuration

distance = [∞] * n

distance[src] = 0

(3) So, there are (n-1) iterations or (n-1) iteration needs to be done

On each iteration you have to go over all the edges each time

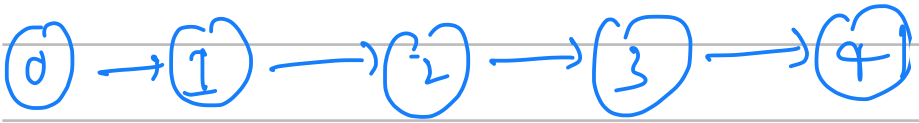
* If you haven't reached a node in an iteration ignore it

(4) Intuition

We know the thought process but the question is why (n-1) iterations — will it update all the nodes's distance

* Edges can be given in any order

Consider this



dist = [0 ∞ ∞ ∞ ∞]

1st iteration

Edges

3 4 1

2 3 1

1 2 1

0 1 1

dist[3] + 1 < dist[4] — nothing

dist[2] + 1 < dist[3] — nothing

dist[1] + 1 < dist[2] — nothing

dist[0] + 1 < dist[1] ≡ [0 1 ∞ ∞ ∞]

2nd iteration

dist[3] + 1 < dist[4] ✗

dist[2] + 1 < dist[3] ✗

dist[1] + 1 < dist[2] ≡ [0 1 2 ∞ ∞]

dist[0] + 1 < dist[1] ✗

⋮

⋮ → and so on

If the order of edges have been given in below order

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

→ } On a single iteration all the min dist would have been calculated

This is the intuition of running iterations (n-1) times

(*) How to detect a negative cycle

we know (n-1) iteration will update the distance completely

if on nth iteration distance array updates for any node then there is a negative cycle.

```
class Solution:
    # Function to construct and return cost of MST for a graph
    # represented using adjacency matrix representation
    '''
    V: nodes in graph
    edges: adjacency list for the graph
    S: Source
    '''
    def bellman_ford(self, V, edges, S):
        inf = 10**8
        # Initial configuration
        distance = [inf] * V
        distance[S] = 0
        # Perform V-1 iterations
        for _ in range(V-1):
            # Go Through each edge
            for edge in edges:
                u, v, wt = edge
                if distance[u] != inf and distance[u] + wt < distance[v]:
                    distance[v] = distance[u] + wt
        # check for negative cycle by running Vth iteration
        for edge in edges:
                u, v, wt = edge
                if distance[u] != inf and distance[u] + wt < distance[v]:
                    return [-1]
        return distance
```

(*) Floyd Warshall Algorithm

Bellman ford and Dijkstra were single source shortest path algo

-> This is different

→ multi source shortest path also
→ helps detect negative cycles well

Thought process →

Go via every node & calculate

→ Use adjacency matrix to represent the graph

→ initialize a cost matrix of the same size

*Initial configuration

(1) Store every edge into cost matrix

(2) (i,i) ≡ Store 0 as cost of itself will always be 0

(3) iterate through every vertex 0 .... n-1 and do

```
via = 0  to via = n-1 :
    for i = 0 to n-1 :
        for j = 0 to n-1 :
```

loop through cost matrix

cost[i][j] = min(cost[i][j],

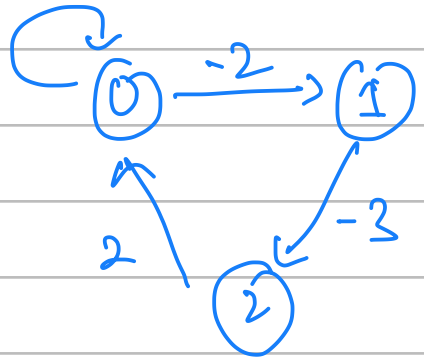
cost[i][via] + cost[via][j])

(4) The resultant cost matrix will give shortest path from every vertex to another

The algorithm is not much intuitive as the other ones. It is more of a brute force, where all combination of paths have been tried to get the shortest paths.

Nothing to be panic much on the intuition, it is a simple brute on all paths. Focus on the three for loops.

(4) How to detect a negative cycle



Cost of each vertex to itself should always be 0 but for negative cycles the cost to itself will be calculated as -ve

for i = 0 to n :

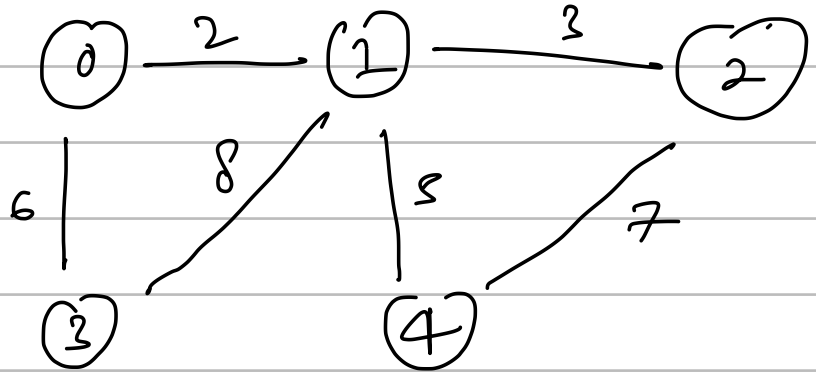
if cost[i][i] < 0 :

cycle detected

```python
class Solution:
    def shortest_distance(self, matrix):
        n = len(matrix)
        # Initial configuration
        # Initialize a cost matrix
        cost = []
        for _ in range(n):
            temp = [float('inf')]*n
            cost.append(temp)
        # u->u should be 0 and direct edges cost u->v in adj matrix
        for i in range(n):
            for j in range(n):
                if matrix[i][j] == -1: # no edge
                    cost[i][j] = float('inf')
                else:
                    cost[i][j] = matrix[i][j]

        # Iterate through every vertex 'via' and calcuate the costs
        for via in range(n):
            # loop through the cost matrix and update
                for i in range(n):
                    for j in range(n):
                        cost[i][j] = min(cost[i][j], cost[i][via] + cost[via][j])
        # update -1 for v which are not reachable from u
        for i in range(n):
          for j in range(n):
              if cost[i][j] == float('inf'):
                  cost[i][j] = -1

        return cost
```

* Minimum Spanning tree (MST)

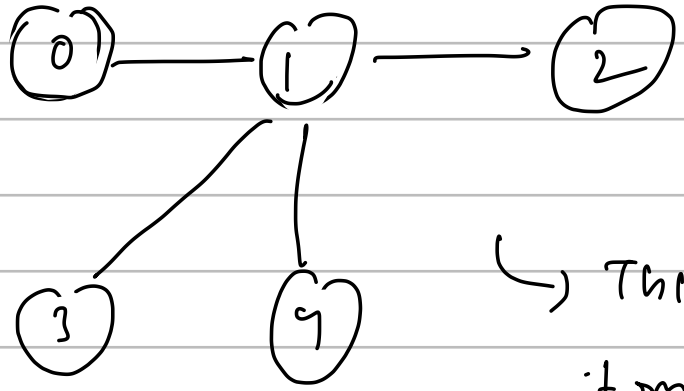
undirected graph



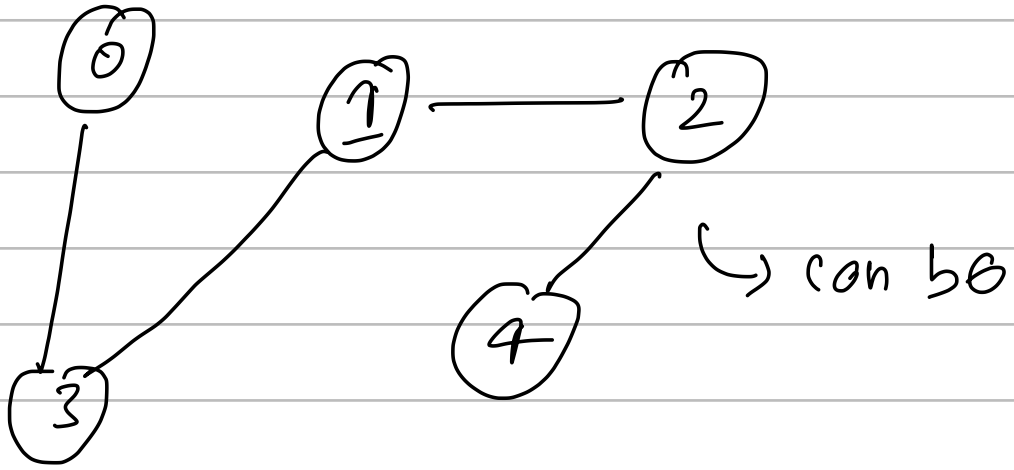
A tree in which we have n nodes and n-1

edges = no cycles
all nodes are reachable

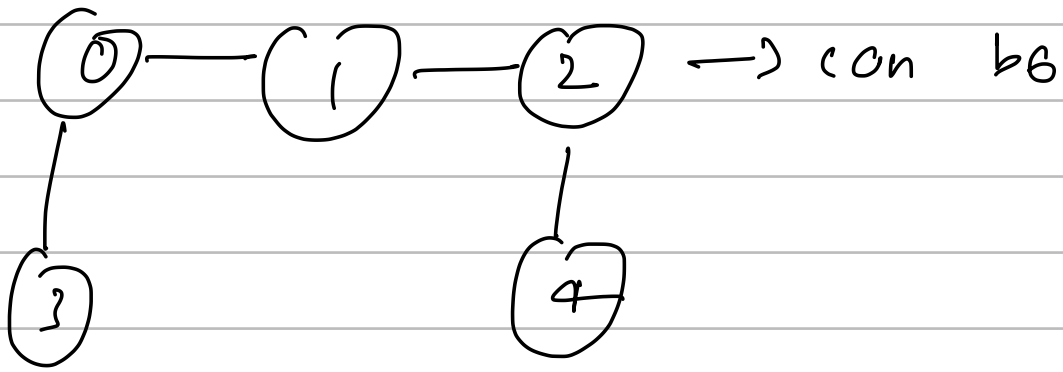
If a graph contains n vertices and n-1 edges and is connected it cannot have any cycles



↳ This can be a spanning tree



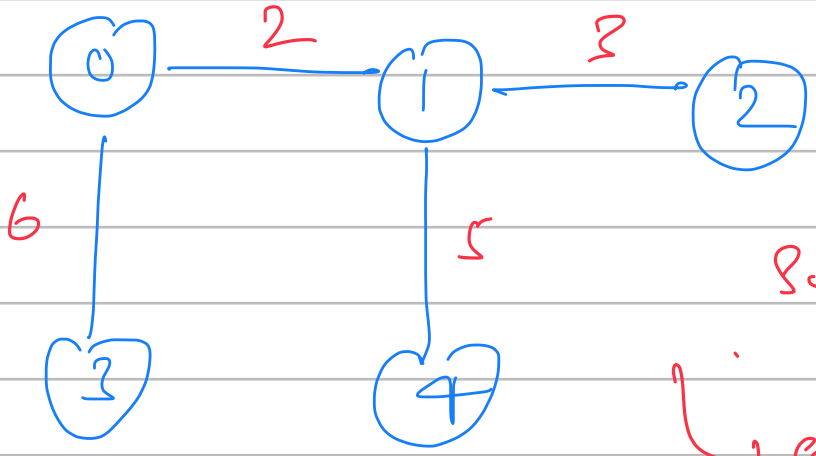
↳ can be



→ can be

These are all spanning trees

(*) The spanning tree which has the least or the minimum sum will be MST of a given graph



Sum = 16

↳ on MST

There can also be multiple MSTs for a graph

(*) Prim's Algorithm

helps to find MST for a graph

Required 2 things

- Priority queue
- visited array

} Initial configuration

- MST list = [ ]

* Initial configuration

priority queue = (wt, node, parent)

visited ≡ [ ]

MST ≡ [ ]

(*) Can start with any node

Iteration

(1) Take the element out of pqueue.

(wt, u, v) ≡ pq.pop()

(2) if (v) is already visited continue

(3) if (v) is not visited :

(a) mark (v) visited

(b) put (u – v) edge in MST

(c) Update the Sum = Sum + wt

(d) iterate through children of (v)

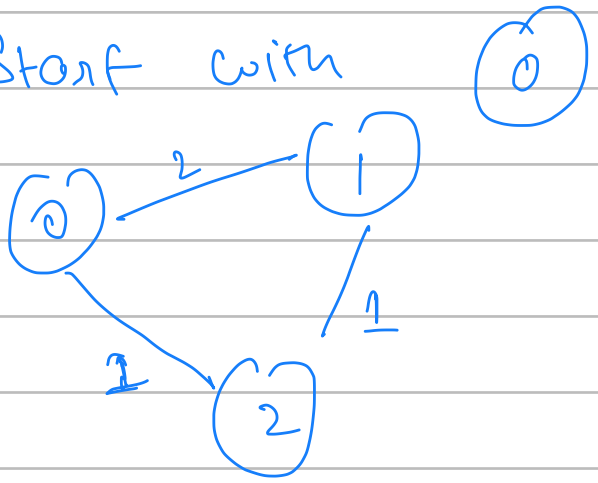
* if neighbor is not visited

* push them to the queue

```python
from heapq import heapify, heappush, heappop
class Solution:

    #Function to find sum of weights of edges of the Minimum Spanning Tree.
    def spanningTree(self, V, adj):
        # Initial configuration
        visited = [False] * V
        MST = []
        pque = [] # tupe (wt, u, parent)
        heappush(pque, (0, 0, -1))
        sum = 0

        # Iteration
        while len(pque) > 0:
            wt, u, parent = heappop(pque)
            if visited[u] is True:
                continue
            visited[u] = True
            sum += wt
            MST.append((u, parent))
            for neighbor in adj[u]:
                v, wt = neighbor
                if visited[v] is False:
                    heappush(pque, (wt, v, u))
        return sum
```

* Intuition

Start with (0)



(0) pushed (1) and (2) in pqueue and in next iteration it took min of the 2 edges which is (0) — (2)

now as soon as you take out (2) from pq

it pushed (2) —1— (1) which resulted in not considering (0) — (1)

(*) greedy is the intuition here

(*) TC → O(E log E)

(*) Disjoint Set Data structure

→ Traversing through DFS or BFS costs O(V+E)

→ for dynamic graphs use DS

Two methods
- find Parent → Path compression
- union → Rank
  - → size

Initially

Every node is alone

union will connect two vertices

(*) union by rank

rank = [ ]

parent = [ ]

UNION (u, v):

(1) find parent of u and v = p1 and p2

(2) find rank of p1 and p2 = r1 and r2

(3) Attach the smaller rank to larger rank

FIND-PARENT (u):

if parent[u] == u:

return u

parent[u] = FIND-PARENT (parent[u])

return parent[u]

TC = O(4x)

PATH compression technique

* union by size technique

Size = [1, 1, 1, 1 ...]

parent = [0, 1, 2, 3 ....]

Consider edges

(1, 2)

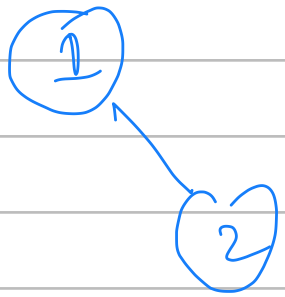
(2, 3)

(4, 5)

Union (1, 2)

size = [2, 1, 1, 1 ...]

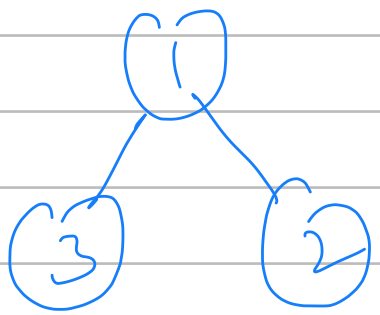
parent = [1, 1, 3, 4, 5 ....]



union (2, 3)

size = [3, 1, 1, 1 ....]

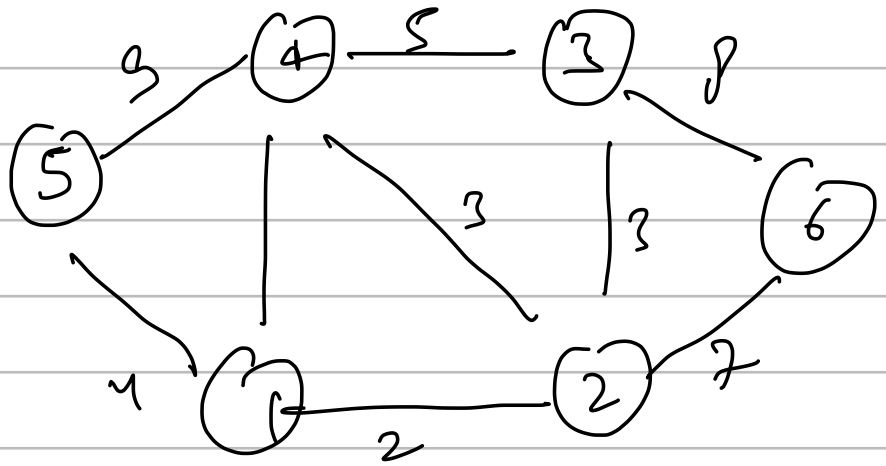
parent = [1, 1, 1, 4, 5 ...]



* Kruskal's algorithm (Disjoint Set)

(1) Sort all the edges according to weights

(2) Initialize a disjoint set datastructure with the given nodes



DS = (1) (2) (3) (4) (5) (6)

Edges Sorted

wt	u	v
1	1	4
2	1	2
3	2	3
3	2	4
4	1	5
5	3	4
7	2	6
8	3	6
9	4	5

(3) Pick the edge ones by one and add it to

the disjoint set and calculate the total weight

↳ if (u) (v) already connected then ignore the edge as this will only create a cycle.

* Disjoint Set implementation

```
class DisjointSet:
    def __init__(self, n):
        self.parent = []
        self.rank = []
        for i in range(n+1):
            self.parent.append(i)
            self.rank.append(0)

    def getParent(self, u):
        if self.parent[u] == u:
            return u
        self.parent[u] = self.getParent(self.parent[u])
        return self.parent[u]
    def union(self, u, v):
        p_u = self.getParent(u)
        p_v = self.getParent(v)

        if p_u == p_v:
            return
        if self.rank[p_u] > self.rank[p_v]:
            self.parent[p_v] = p_u
        elif self.rank[p_u] < self.rank[p_v]:
            self.parent[p_u] = p_v
        else:
            self.parent[p_v] = p_u
            self.rank[p_u] += 1
    def connected(self, u, v):
        if self.getParent(u) == self.getParent(v):
            return True
        return False
```

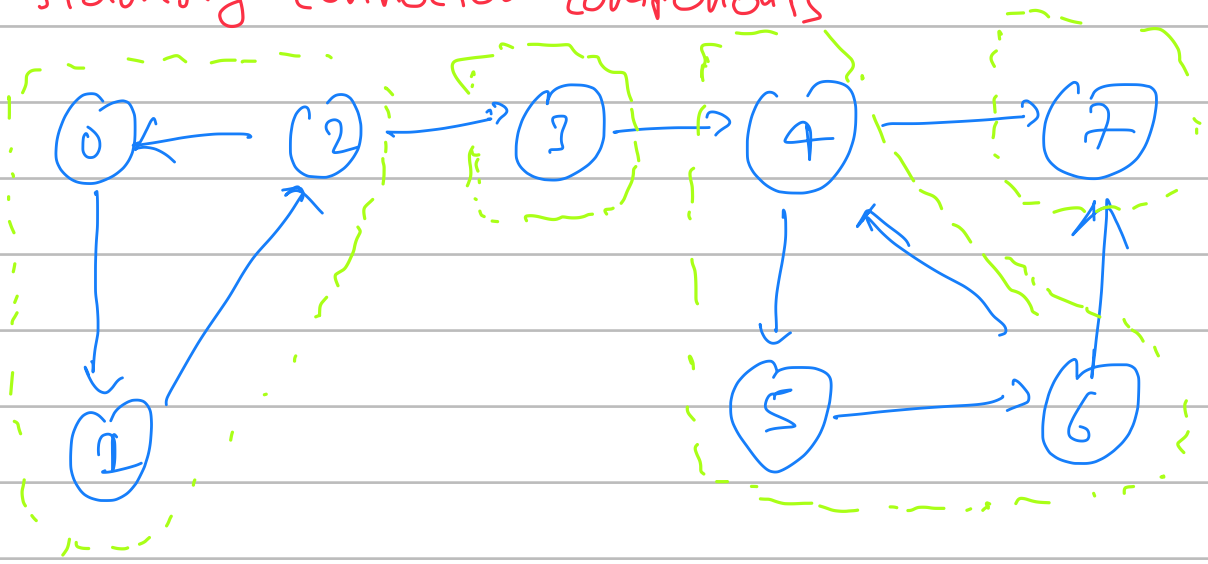
* Kruskal's implementation

```python
class Solution:
    #Function to find sum of weights of edges of the Minimum Spanning Tree.
    def spanningTree(self, V, adj):
        # Initial configuration
        # 1. Get the edges
        edges = []
        for i in range(len(adj)):
            for neighbor in adj[i]:
                u = i
                v, wt = neighbor
                edges.append([wt, u, v])
        # 2. Sort the edges
        sorted_edges = sorted(edges, key=lambda x: x[0])
        # 3. Create a disjoint set data structure
        ds = DisjointSet(V)

        weight = 0
        # Go through every edge
        for edge in sorted_edges:
            wt, u, v = edge
            if ds.connected(u, v) == False:
                weight += wt
                ds.union(u, v)
        return weight
```

Kosaraju's Algorithm

Note -> Strongly connected comps are only valid for Directed Graphs

-> Strongly Connected Components



Consider a (u) and (v) if in a graph you can reach from u → v and you can also reach from v → u then that set of vertices are known as strongly connected Component

→ 0 1 2

→ 3

→ 4 5 6

→ 7

* Basic intuition

if you apply DFS it will visit all nodes and that we don't require.

we can divide the graph as follows

SCC 1 ⟶ SCC 2 ⟶ SCC 3 ⟶ SCC 4

* Now if you reverse every edge SCC won't change you will still be able to traverse it but you won't be able to go to SCC 2 from SCC 1

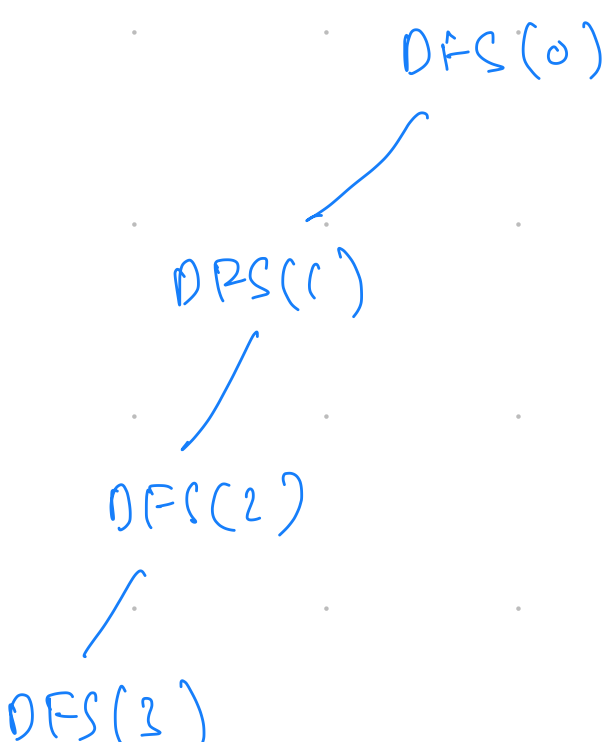
* Kosaraju steps

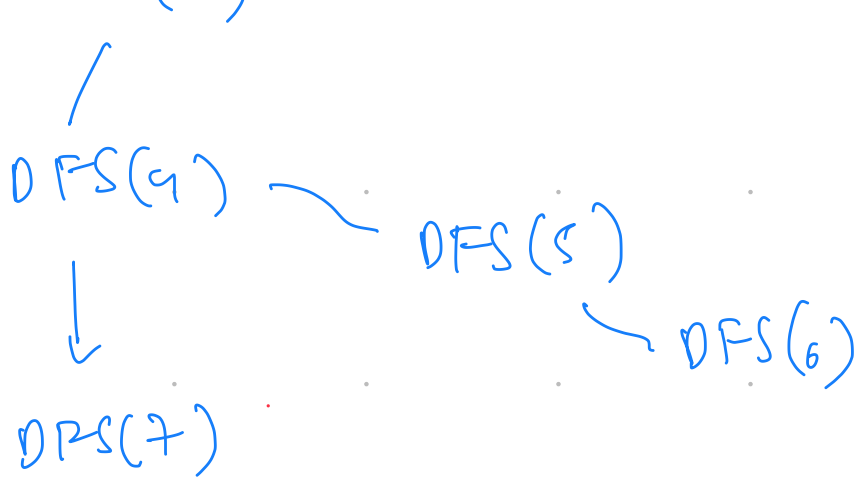
1 ⟶ Sort all the edges according to finishing time

2 ⟶ Reverse the graph

3 ⟶ Do a DFS

Step - 1



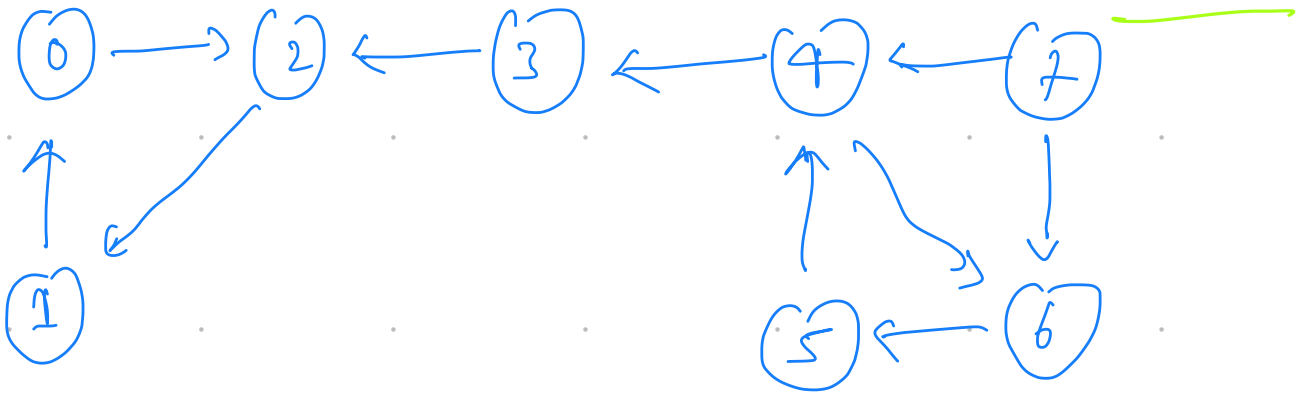


{7 is the first guy and is the last portion in the SCC}

Store the vertices when you back track

7 6 5 4 3 2 1 0 order of finishing time

Step 2

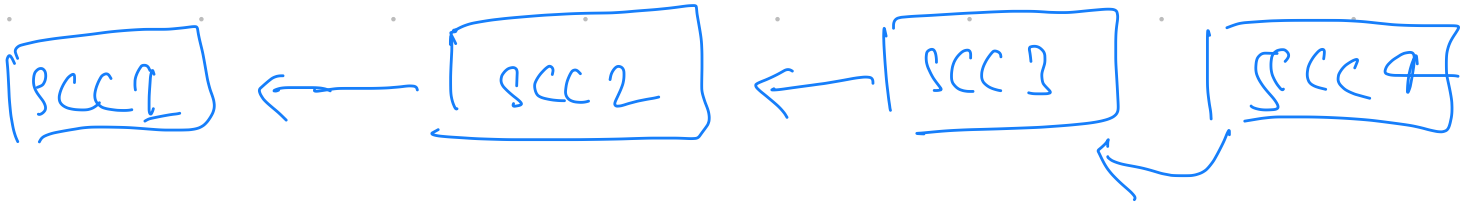


Step -3

Initially if you did a DFS on the original

graph you will do something

* Now since we have traversed all the edges



* We want to do a DFS of SCC 1 first as doing a DFS of any other SCC would reach the previous one

* This order we can get from the finishing time stack that we got in Step 1

```
def DFS_1(self, u, V, adj, visited, order):
    visited[u] = True
    for v in adj[u]:
        if visited[v] == False:
            self.DFS_1(v, V, adj, visited, order)
    order.append(u)
def DFS_3(self, u, adj, visited, scc):
    visited[u] = True
    scc.append(u)
    for neighbor in adj[u]:
        if visited[neighbor] == False:
            self.DFS_3(neighbor, adj, visited, scc)
```

```python
class Solution:

    #Function to find number of strongly connected components in the graph.
    def kosaraju(self, V, adj):
        visited = [False] * V

        # Step-1 sort all the veritces w.r.t finishing order
        order = []
        for i in range(V):
            if visited[i] == False:
                self.DFS_1(i, V, adj, visited, order)

        # Step-2 Reverese all the edges
        adjT = []
        for _ in range(V):
            adjT.append([])
        for u in range(V):
            visited[u] = False
            for v in adj[u]:
                adjT[v].append(u)

        # Step-3 Perform DFS on finishing order
        scc_s = [] # strongly connected comps
        while len(order) > 0:
            u = order.pop()
            if visited[u] == False:
                scc = []
                self.DFS_3(u, adjT, visited, scc)
                scc_s.append(scc)
        return len(scc_s)
```

* Bridges in graph

Any edge whose removal brokes down the graph into two components or more components is a Bridge