11 Introduction to graphs

Def :- In this course we'll study about finite graphs.

Graph is defined as a set Gr, which is itself a set of set V and set E.

i.e. Gr(V,E) is called a graph.

where V is any finite set, which is why we call it finite graph.

& E is any subset of (V×V) i.e. E is relation on V.

As we already know that

$$V \times V = \{(a,b) \mid a,b \in V\}$$

∵ V is a finite set, hence we can assume that it consists of n elements. These elements are known as the nodes or vertices of the graph.
Without loss of generality we may assume

$$V = \{0, 1, 2, \dots n-1\}$$

It is so b/c, even if V is collection of some other n numbers, then we know that there is a bijection b/w 2 sets with same cardinality.

so, then $\phi(0), \phi(1) \dots \phi(n-1)$
will give us the set of nodes in the graph

n = cardinality of V

m = cardinality of E

Date ___/___/___
Page ___

The elements of E are known as edges of the graph.

Let m denote the no. of edges in the graph.

As can be observed $0 \leq m \leq n^2$.

When $(i, j) = e$ is an edge in the graph.
we say that :-

a.) i & j are adjacent nodes
b.) i is adjacent to j
c.) i & j are neighbors
d.) i & j are incident on edge e.

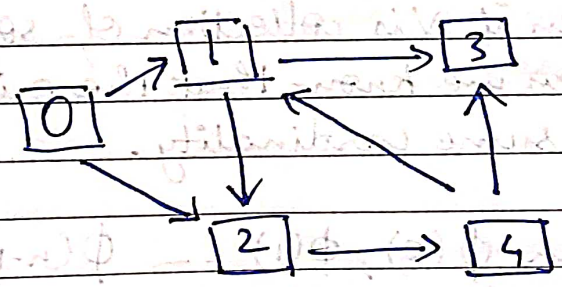
The other way to represent the graph is pictorially.
We plot all i's & j's.
each (i & j) ~~is~~ are connected by an arrow, tail of ~~arrow being~~ being at i, while head of " " " j.

$(i_0, j_0), (i_1, j_1), (i_2, j_2)$

For ex:- $E = \{(0,1), (0,2), (1,2), (1,3), (2,4), (4,1), (4,3)\}$

$n = 5$ & $m = 7$

1.) DIRECTED GRAPH



The edge relation is not symmetric. (a, b) is an edge does not mean (b, a) is an edge. i.e. direction of the arrow matters

Read

⇒ life Examples of such graph can be visualised as:-

i) each ~~node~~ node is a place say A, B ---
& there is one way traffic defined b/w each place.

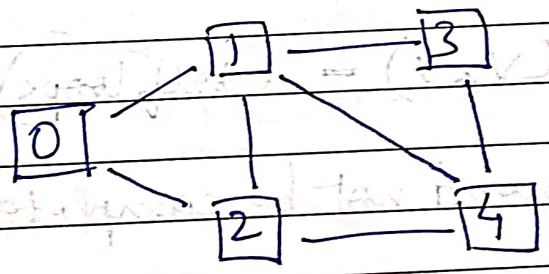
ii) relationships b/w persons:-
one way relationships such as
I know the PM of India but PM of India does not know me.

—x—

2.) UNDIRECTED GRAPHS

$E = \{(0,1), (0,2), (1,2), (1,3), (2,4), (4,1), (4,3)\}$
$n = 5$ & $m = 7$

The edge relation is symmetric. (a, b) is an edge means that (b, a) is an edge.
⇒ NO arrows needed.



Real life example:-

facebook friends means 'if 'a' is a friend of 'b'. implies 'b' is a friend of 'a'.

⇒ **Simple Graphs**

We'll study simple graphs i.e. graphs without self loops.
i.e. (a,a) is not an edge.

∴ In :-

1.) undirected graphs : $0 \leq m \leq \frac{n(n-1)}{2}$ ($^nC_2$)

2.) directed graphs : $0 \leq m \leq n(n-1)$

The ranges for m given in ppt are WRONG.

↳ The ranges given here are CORRECT.

Graphs whose m are $O(n^2)$ are called as dense graphs.

Complete Graph :- ~~undirected graph~~.

Graphs for which :

$$E = (V \times V) - (\text{self loops})$$

∴ dense graphs need not be complete.

Graphs with m as $O(n)$ are called sparse graphs.

(*) Weighted graph → $(W: E \to R)$

Graphs whose edges are correspondingly associated with a weight (weighted edges).

Real life example

if all the nodes correspond to some customer or a bank and the weight is defined as the amount of money deposited by a customer.

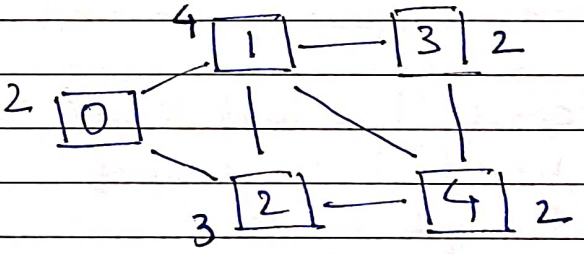
Then the weight becomes positive for that (customer, bank) edge when the customer deposits money and negative when the customer withdraws money or either takes a loan.

Hence, Weight can be NEGATIVE.

—x—

(*) Degree of a node → The no. of edges incident on the node.

⇒ Undirected Graphs

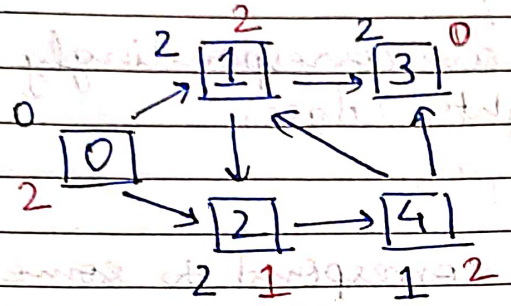


The no. written beside each node is its degree.

Observe–

→ (Sum of degree of nodes in the graph) = 2 × (no. of edges in the graph)

⇒ Directed Graphs



~~Incoming~~

In degree

Out degree

→ In-degree of a node is the number of incoming edges to the node.

→ Out-degree of a node is the number of outgoing edges to the node.

Observe

** Sum of in-degree nodes = Sum of out degree nodes = m

m → no. of edges

(#) Data Structure to store graphs

We'll mainly discuss the following types of data structures:-

a.) Adjacency matrix representation
b.) Adjacency list representation.
c.) Edge list representation

—x—

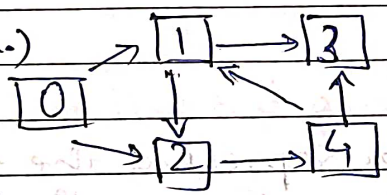
1.) Adjacency matrix

Assuming n number of vertices in the graph, we'll take (n×n) matrix i.e. A[n][n]

This will be a boolean matrix.

A[i][j] = true iff (i, j) is an edge in the graph.

Ex:- a.)



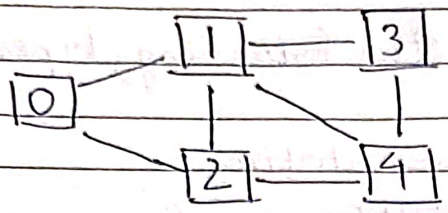
DIRECTED GRAPH

i \ j	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	1	0
2	0	0	0	0	1
3	0	0	0	0	0
4	0	1	0	1	0

(*) ∵ We'll deal with simple directed & undirected graphs ⇒ (i, i) will not be an edge

⇒ The digonal entries will be zero.

b.) Undirected Graphs



i \ j	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	1	1
2	1	1	0	0	1
3	0	1	0	0	1
4	0	1	1	1	0

Observe that in undirected graphs the adjacency matrix is a symmetric matrix.

—x—

The disadvantage with this kind of data structure is that it needs $\Theta(n^2)$ space memory even though m is $\Theta(n)$.

—x—

⇒ Real life example

Assuming that the vertices represent the facebook users ⇒ $n \approx 10^9$ ⇒ space needed $\approx 10^{18}$
even a super computer won't have that much memory. However, most of the entries will be 0 b/c you won't know most of the facebook users.

—x—

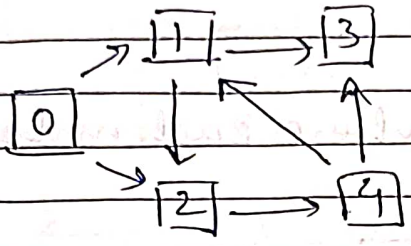
Hence, when the graph is dense this may be one of the ways of representing but when the graph is sparse this is not an efficient way of representing it.

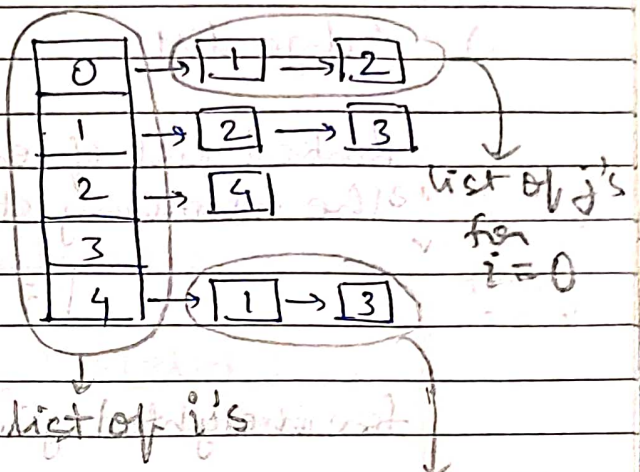
→ For weighted graph, instead of storing the boolean value, we'll store the corresponding weight of the edge.

2.) Adjacency list

In order to overcome the problem mentioned in the prev. pg we have adjacency list representation.

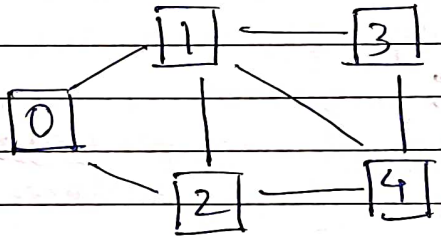
a.) DIRECTED GRAPH

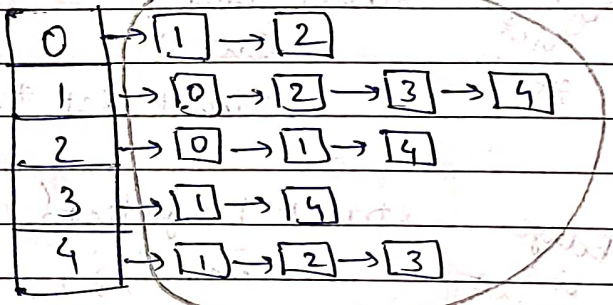




spaces needed to store a graph is $\Theta(n+m)$

b.) Undirected graph





space needed = $n+2m = \Theta(n+m)$

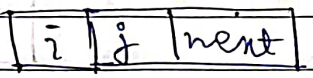
for weighted graph each of these nodes will contain an additional parameter i.e. the weight of the corresponding edge.

The disadvantage with this data structure is that if $m << n$, even then you've to declare an array of size n with most of the entries as NULL.

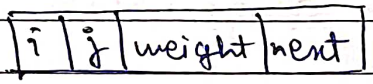
—x—

3.) Edge list

Linked list of edges, where each node will have the following struct:-



for weighted graph the struct will be



Space needed = $\Theta(m)$

—x—x—

Adjacency list struct

```
struct List{
    int i;
    struct List *next; };
```

Edge list struct

```
struct Edgelist{
    int i, j;
    struct Edgelist *next; };
```

—x—

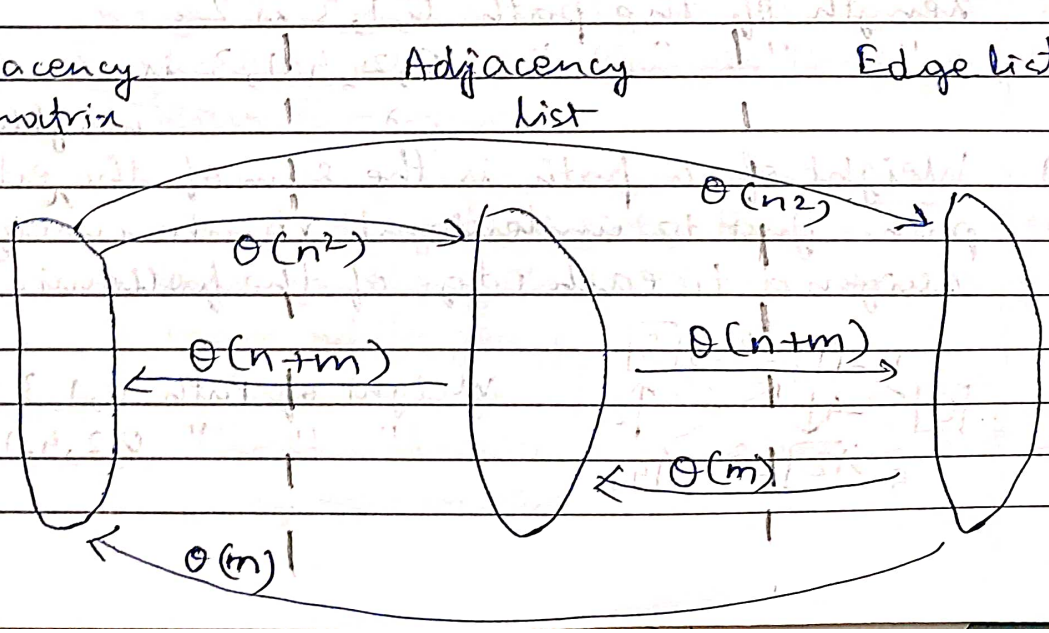
⇒ Code to generate random Graph

```
→ for (i=0; i<n; ++i)
  {
    for (k=0; k<n/10; ++k)
    {  j = rand()%n;
       if (!A[i][j]) {
Adjacency matrix ←——  A[i][j] = true;
Adjacency list ←——  list[i] = newedge(list[i], j);
Edge list ←——  elist = newedgelist(elist, i, j);
                ++m; } }
  }
```

—x—

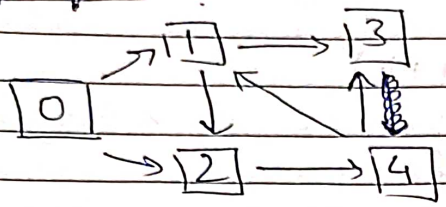
→ Time complexity for conversion from

data str A to data str B
= order of space needed to store in data str A.



Shortest Path Problem

(*) Def of Path



Path is a sequence of nodes $v_1, v_2, v_3 \dots v_k$ such that if we take any 2 adjacent nodes (v_i, v_{i+1}) then it should be an edge in the graph.

0, 1, 3 is a path
0, 2, 4, 1, 3 is a path.

—x—

(*) Cycle is a path, in which the first node is same as the last node.

1, 2, 4, 1 is a cycle.

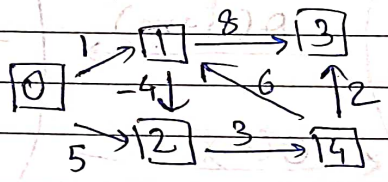
—x—

(*) Length of a path is the no. of edges in the path.

Length of the path 0, 1, 3 is 2.
" " " " 0, 2, 4, 1, 3 is 4.

—x—

(*) Weight of a path is the sum of the weights of edges in the path. Just to understand, random weights are assigned to each edge of the following graph.



Weight of Path 0, 1, 3 = 1 + 8 = 9
" " " 0, 2, 4, 1, 3 = 5 + 3 + 6 + 8
= 22

exist for both directed and undirected graphs.

(*) Shortest Path Problem

Given 2 nodes s & t, find the shortest weight path

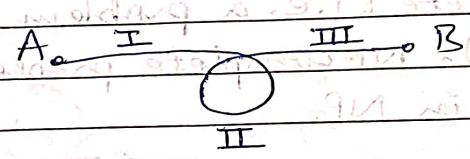
Among all the paths b/w s & t, find a path with minimum weight.

Shortest path is always a ~~simple~~ noncyclic path

path which does not contain a cycle.

Q. → Why?

Ans:- Lets see through an example. Let the path be as follows with 3 parts, part II being a cycle



Case 1 weight of cycle is +ve, then to choose the shortest path, it is obvious that you'll leave the cycle

Case 2 weight of cycle is -ve, then you can go through the cycle as many times as you want while going from A to B

⇒ weight of path can be $-\infty$.

Hence, shortest path problem is not well defined in this case.

—x—

(*) Longest path problem

Among all the paths b/w s & t, find a path with maximum weight.

Shortest path problem is solvable in polynomial time, but the longest path problem is NP-Hard.

Length of the longest path is n-1 iff there is a Hamiltonian path in the graph.

↓

a path in an undirected or directed graph that visits each vertex exactly once. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete (i.e. a problem which is both NP and NP-hard). NP-complete problems represent the hardest problems in NP.

—x—

⇒ There is no polynomial time brute force method to solve the shortest path problem.

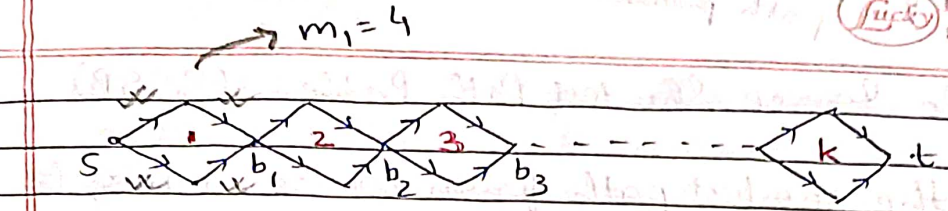
It's so b/c the number of paths in a graph can be exponential in number of nodes and number of edges in the graph.

↓

To travel along each path, find its weight & find the smallest value among them

ex in next pg.

Ex:-

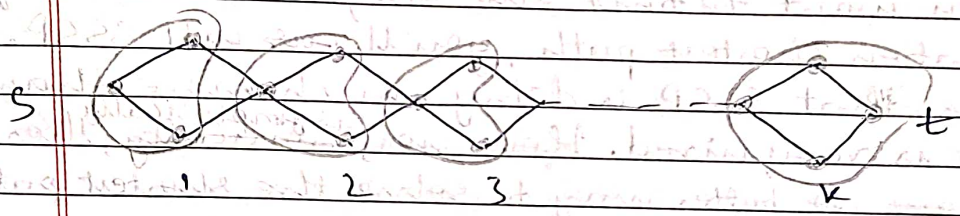


blocks

There are 2 possible paths to reach node b_1
2^2 " " " " b_2
2^3 " " " " b_3
- - - - - - - - -
2^k possible paths to reach t

for block 1 $m_1 = 4$
" " 2 $m_2 = 4$
" " 3 $m_3 = 4$
- - - - - - - - - -
for block k $m_k = 4$

$m = m_1 + m_2 + \dots + m_k = 4k$



for block 1 $n_1 = 3$
" " 2 $n_2 = 3$
" " 3 $n_3 = 3$
- - - - - - - - - -
for block k $n_k = 3 + 1$

$n = n_1 + n_2 + \dots + n_k = 3k + 1$

→ Two kinds of ~~ato~~ algo to solve the shortest path problem.

Date ___/___/___
Page ___

(*) Single Source Shortest Path Problem (SSSP)

Find the shortest path from source nodes, to every other node in the graph.

(*) All Pair shortest Path (APSP)

Find the shortest path b/w every pair of nodes in the graph.

→ So, if you ~~kno~~ known the solution to SSSP, one way to solve APSP is to call the SSSP for all the 'n' nodes varying over graph. But we'll see that calling APSP is much more efficient than calling SSSP n number of times in some cases.

→ If you want to find the single source, single destination shortest path, you'll ~~cas~~ call SSSP. But we see that SSSP is doing much more than what is required. However, asymptotically, there is ~~no~~ ~~asy~~ no better way to solve the shortest path problem. You'll have to call SSSP for that

—x—

Now, we'll see 3 algorithms, 2 for the SSSP & 1 for the APSP. ~~It's difficult to get a feeling, as to why these algo work correctly. So, the correctness of the algo will be dealt with at some later point of time.~~

In this course, we'll study in detail only one of the 2 SSSP. The rest of the 2 algo are easy to implement and

it's also easy to compute their complexity. However, for these 2 algo, it's a difficult to get a feeling, as to why these algo work correctly. So, the correctness of these 2 algo will be dealt with at some later point of time. (after we've learnt dynamic programming)

—x—

⇒ SSSP

1.) Dijkstar's Algorithm → most efficient when
- works only when all the weights are +ve.
- complexity - $O((n+m)\log n)$
- Uses Adjacency list

2.) Bellman - Ford Algo
- complexity - $O(nm)$
- uses edge list

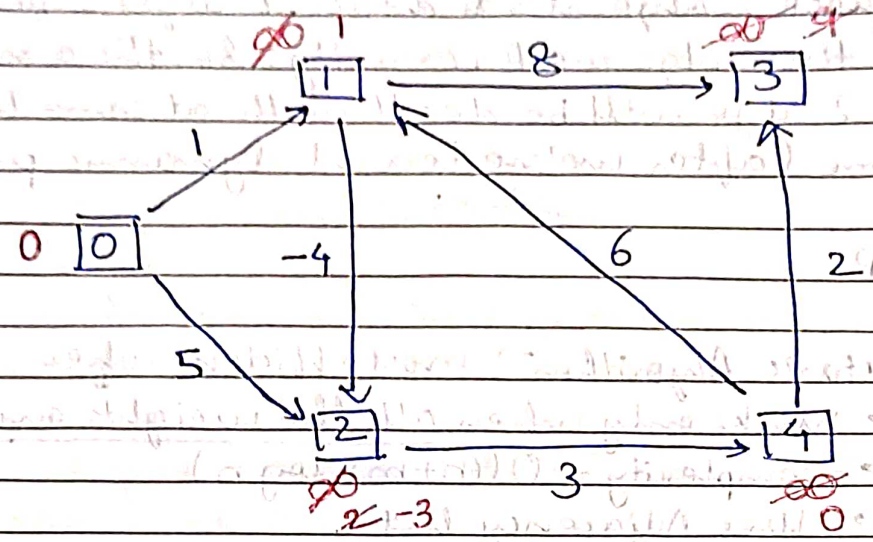
⇒ APSP

1.) Floyd - Warshall Algo
- complexity - $O(n^3)$
- Uses adjacency matrix.

—x—

→ struct Edgelist{
edge ← [int i, j;
weight of edge ← [float w;
struct Edgelist *next; };

How Bellman-Ford Algo works



Make an array SD[ ] of size n (no. of nodes)

(will store the shortest distance of corresp. node from source S.)

lets assume that given 'S' is 0

∵ source 'S' is 0

⇒ initialize SD[0] = 0

& " rest all indices to ∞

⇒ SD = {0, ∞, ∞, ∞, ∞}

Edges → (0,1) (0,2) (1,3) (1,2) (2,4) (4,3) (4,1)

We'll run a loop 5 (i.e. n) times for each edge. What we'll do is that

we'll see the current shortest distance of the j of each edge node from the source by using the array SD.

If current_shortest_dist of j > current_shortest_dist of i + weight of edge

then update

current_shortest_dist. of j = current_shortest_dist of i + weight

So,

We start with (0,1)

$\infty > 0 + 1$

∞: current_shortest_dist of 1; 0: current_shortest_dist of 0; 1: weight of (0,1)

⇒ We update current_shortest_dist of 1 to 0+1=1

∴ Now, SD = {0, 1, ∞, ∞, ∞}

Now, for (0,2)

$\infty > 0 + 2$

∞: current_shortest_dist. of 2; 0: current_shortest_dist of 0; 2: weight of (0,2)

⇒ We update current shortest dist. of 2 to 0+2=2

Now SD = {0, 1, 2, ∞, ∞}

Similarly for (1,3) (1,2) --- (4,1)

↓

5 times, we do this

Finally SD = {0, 1, -3, 2, 0}

⇒ shortest dist. of 1 from 0 = SD[1] = 1

" " " 2 " 0 = SD[2] = -3

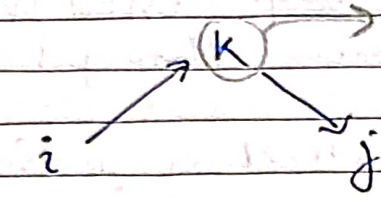
" " " 3 " 0 = SD[3] & so on

⇒ Code for Bellman-Ford

→
```
void BellmanFord (struct Edgelist *elist,
                  float SD, int s, int n)
{ struct Edgelist *temp; int k;
  for(k=0; k<n; ++k)
  {SD[k] = INT_MAX;}
    SD[s] = 0;

  for (k=0; k<n; ++k){
      temp = elist;
      while (temp){
          if (SD[temp->j] > SD[temp->i]
                          + temp->w)
          {SD[temp->j]
                = SD[temp->i]+temp->w;}
          temp = temp->next;
      }}
}
```

→ See Abdul Bari Vid.

(*) Floyd - Warshall Algorithm



If (weight of shortest path from i to j is shorter than > weight of shortest path from i to k + weight of shortest path from k to j)

then update

weight of shortest path from i to j = weight of shortest path from i to k + weight of shortest path from k to j

→ void FloydWarshall (float A[][1000], float D[][1000], int n)

```
{ int i, j, k;
  for(i=0; i<n; i++)
      for(j=0; j<n; j++)
          D[i][j] = A[i][j];
  for(i=0; i<n; i++)
      D[i][i] = 0;
  for (k=0; k<n; k++)
      for(i=0; i<n; i++)
          for (j=0; j<n; j++)
              if (D[i][j] > D[i][k] + D[k][j])
                  D[i][j] = D[i][k] + D[k][j];
}
```

→ This looks similar to matrix multiplication.

Q.) We know that for matrix multiplication, we can put that 3 ~~loop~~ nested loops in any order. Can we do the same thing here?

Soln NO, we can't interchange the loops. The approach of the algo is dynamic programming. This means that during computation only partial sol^n are determined. Here (in the code in prev. pg), k is an important parameter in the algo. When dealing with k, the shortest paths are computed that are only allowed to pass vertices 1 upto k, the other vertices can only be initial or final. This means that k has a different position than that of i & j.

SUMMARY

We compute the weight of the shortest paths in these algo.

If you want to find the shortest paths, it can be easily done by adding few more details to these algo., we shall skip these details now, but you'll learn the trick in the due course of time.

Minimum / Maximum Spanning Tree (MST)

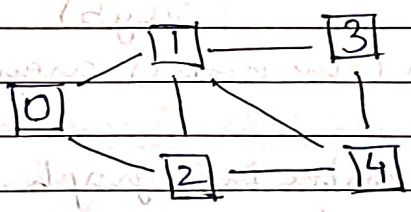
The shortest path problem was defined for both directed and undirected graphs. However, the MST is defined only for undirected graphs.

→

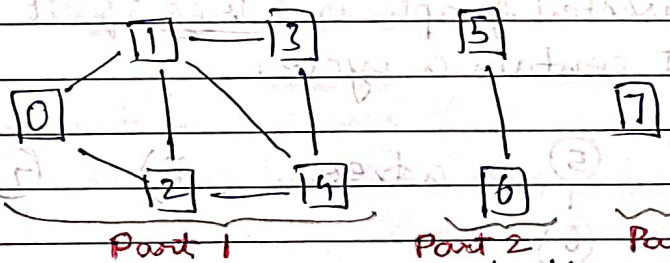
* 2 nodes are connected if there is a path b/w them.

An undirected graph is connected if there is a path b/w every pair of nodes in the graph. In this chapter, whenever we mention graph, we'll mean undirected graph.

Example of a connected graph



Example of a non-connected graph



When the graph is not connected, then we talk about connected component of a graph

* A subgraph S of a graph G is a graph whose vertex set V(S) is a subset of the vertex set V(G) [i.e. V(S) ⊆ V(G)], and whose edge set E(S) is a subset of the edge set E(G) [i.e. E(S) ⊆ E(G)].

Date ___/___/___
Page ___

(*) A connected component of G is a maximal connected subgraph of G.

In the graph (in prev. pg), there are 3 connected components (depicted as part1, part2 & part3).

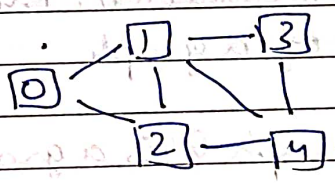
The feature of 2 different connected components of a graph is that if you take any node from one component and any node from the other component, you won't be able to find any path that connects those 2 nodes.

Ex:- take any node (say 3) from part 1
" " " (say 5) " " 2
They ~~are~~ (the nodes) aren't connected.

(*) The entire set of nodes in a graph is the disjoint union of set of ~~nodes~~ nodes in the connected components.

(*) An undirected graph is a tree if it is connected and does NOT contain a cycle.

Ex:- i) (5) is a tree (6) ii) (7) is a tree

iii) 

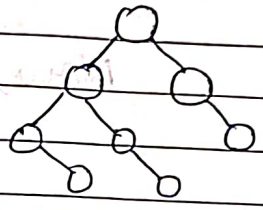
is not a tree.

(*) Forest - It's a collection of trees, ~~that~~ does not contain a cycle, but may not be connected.

Ex:- set of { (5)-(6) , (7) } is a forest.

→ The binary trees that we have studied in the course are special kind of Trees/Graphs.

(x, x→parent) is an edge



—x—

Theorem 1 - Let G be an undirected graph with n nodes. Any two of the following statements will imply the third.

1. G is connected
2. G does not contain a cycle
3. G has n-1 edges

—x—

(*) By observing Th^m 1 we can say that, every tree will contain (n-1) edges. It's so b/c the very def. of a tree contains statement 1 & 2 of Th^m 1. Hence, statement 3 should be true.

→ Before coming to the proof of Th^m 1, lets look at some observations:-

- Let G be an undirected graph with n nodes. Let us start with only these isolated nodes and start adding the edges in any order

Note:- The graph has n connected components as we've not added any edges at first.

- If we add an edge going across, the connected components, then the number of connected components decreases by 1.
- If we add an edge within the connected components, then we'll introduce a cycle in the graph.

Now, we can see the proof.

Proof of Th^m 1

Case 1 — (1 & 2 implies 3)

∵ G does not contain a cycle
⇒ each time we add an edge across (& NOT within) the connected components.
⇒ after adding k edges, we'll have n−k connected components.

But state. 1 says, G is connected. So after adding all the edges the graph should be connected

i.e. $n-k=1 \Rightarrow k=n-1$ (proved)

Case 2- (2 & 3 implies 1)

∵ G does not contain a cycle
⇒ after adding k edges we'll have n-k connected components.

But state. 3 says that $k = n-1$
⇒ finally there are $n-k = n-(n-1) = 1$ connected components.
⇒ graph is connected (proved)

Case 3 - (1 & 3 implies 2)

Lets assume that out of n, only k edges had gone across (& NOT within) the connected components then we'll have n-k connected components.

But graph is connected $\Rightarrow n-k=1 \Rightarrow k = n-1$

∴ All edges have gone across the connected components, hence no cyclic paths (proved.)

—x—

(*) Spanning Tree

Spanning tree T of a graph G is a subgraph of G, which is a tree with n nodes.

The nodes of the spanning tree will be same as the nodes of the graph, but will have only n-1 edges of the graph.

(*) Weight of a spanning tree is defined as the sum of the weights of the edges of the spanning tree.

—x—

(*) MST

Minimum Spanning Tree (MST) is a spanning tree with minimum weight.

Maximum spanning tree (MST) is a spanning tree with maximum weight.

Both the problems are solvable in polynomial time.

—x—

We'll learn 2 algo. namely:—

a.) Prim's algo.
b.) Kruskal's algo.

Both algo. have same complexity as that of Dijkstra's i.e. $O((n+m)\log n)$.

We won't look at their proofs. We'll just see their implementations & how they work.

Whenever we mention Graph, we assume it to be an undirected graph

Cayley's Theorem

In this chapter, we may not look at a very formal proof of this theo. but we'll try to get a feeling of it as much as we can.

—x—

Cayley's Th^m:- The number of distinct spanning trees in a complete graph with n nodes is n^{n-2}.

—x—

Before we look at the proof of Cayley's th^m, we'll look at another th^m which is required for the proof of Cayley's th^m.

—x—

Theo 1- A tree with n nodes ($n>1$) will have at least 2 nodes with degree 1.

—x—

To prove th^m 1, lets first look at a simpler th^m

Th^m 2- A tree with n nodes ($n>1$) will have atleast one node with degree 1

Proof of Th^m 2- Let us assume that the theorem is not true.

$\Rightarrow$ if i is a node then $\deg(i) > 1$

$\Rightarrow \deg(i) \geq 2$; for any i

$$\sum_{i=1}^{n} \deg(i) \quad (\forall i) \geq 2n$$

When we studied the very def. of deg of a node we saw that [sum of deg $= 2m$] (m: no. of edges)

$\Rightarrow \sum deg(i) = 2m = 2(n-1) = 2n-2$

$\Rightarrow 2n-2 \geq 2n \Rightarrow -2 \geq 0$, a contradiction

Hence proved.

—x—

Proof of Th^m 1 Let us assume that the theo. is not true. → Th^m 2

If j be a node with degree 1 and all other nodes have deg > 1

$\left(\sum deg(i) \ (\forall i; i \neq j)\right) + deg(j) \geq 2(n-1)+1$

But sum of all degrees $= 2m = 2(n-1)$

$\Rightarrow 2(n-1) \geq 2n-2+1$

$\Rightarrow 2n-2 \geq 2n-2+1 \Rightarrow 0 \geq 1$,

a contradiction.

Hence proved

—x—

⇒ Just out of curiosity lets see, if,
If A tree with n nodes ($n>1$) will have at least 3 nodes with degree 1?

→ Proof :- Let i & j be the only nodes having deg > 1.

$\Rightarrow 2(n-1) = \sum_{\substack{k \neq j \\ \& \\ k \neq i}} deg(k) + deg(i) + deg(j) \geq \underbrace{2(n-2)+2}_{2(n-1)}$

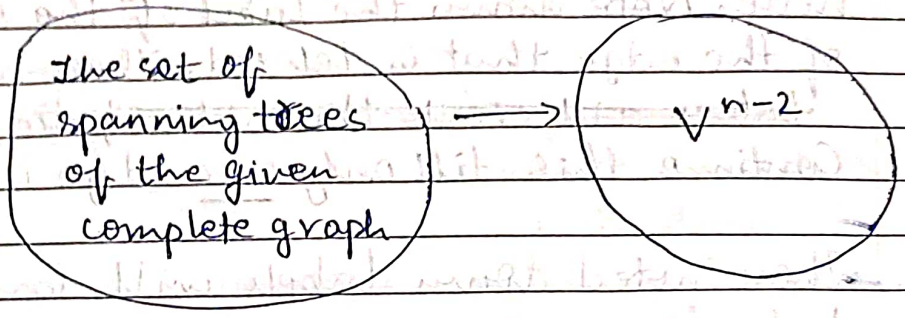
NOT a contradiction.

Cardinality of V is n
" " V^2 is n^2
" " V^{n-2} is n^{n-2}

In fact, a degenerate tree has exactly 2 nodes with degree one.
↓
leaf & root nodes.

—x—

Cayley's Th^m:- The number of distinct spanning trees in a complete graph with n nodes is n^{n-2}.



Showing a bijection b/w the sets will prove the th^m.

It's so b/c cardinality of $V^{n-2} = n^{n-2}$
$\Rightarrow$ no. of spanning trees $= n^{n-2}$

—x—

→ So, all that is left to do is given a set of spanning trees of a given complete graph, we need to find a fu^n that maps to a vector of dim n-2 & a reverse fu^n that maps that vector of dim n-2 to a spanning tree.

→ So, we'll give the algo to produce the mapping & the functions. However we may not prove that the generated fu^n is infact a bijection.

—x—

(*) Now, lets see that given a spanning tree with n nodes, how to map it to a vector of ~~length~~ dim n-2.

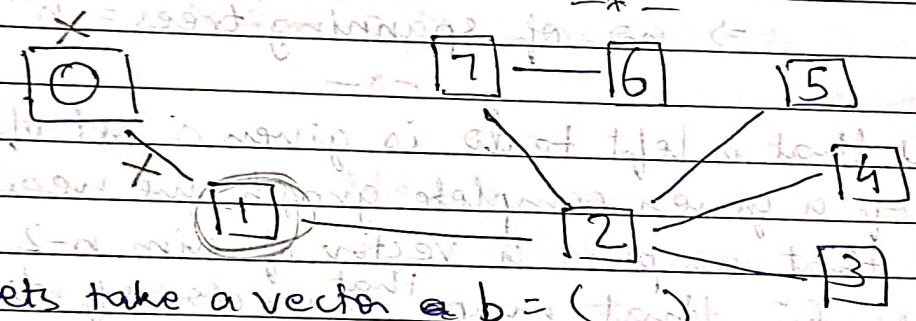
So, pick a degeee one node with least lable (as we know every node is labled from 0 to n-1). Delete this node and the edge incident on the node. Note down the label of the other end point of the edge that is deleted. ~~These noted down labels will create the vector of dim~~

Continue this till only one edge is left. (2 nodes left at last)

These noted down labels will form the vetor of ~~be~~ dim n-2.

⇒ n-2 labels noted

Ex:-



lets take a vector a b = ( )

look at the deg 1 nodes; i.e. the leaf nodes.

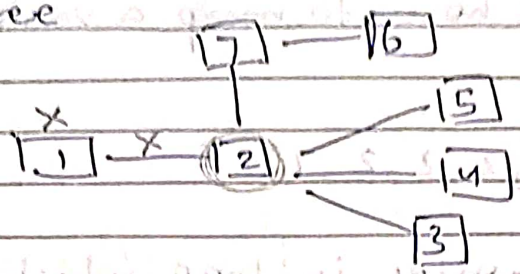
those are 0, 3, 4, 5, 6

least label 0 ⇒ delete 0 & the edge

note down the label of other end of edge i.e. 1.

⇒ b = (1,)

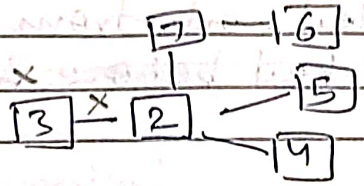
New tree



leaf nodes → 1, 3, 4, 5, 6

delete 1 & its edge. Note down 2

update b = (1, 2)



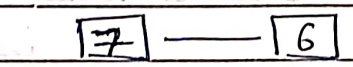
leaf nodes → 3, 4, 5, 6

delete 3 & its edge

Note down 2.

update b = (1, 2, 2)

Keep on doing this. Finally you'll get



b = (1, 2, 2, 2, 2, 7) → the vector corresponding to the spanning tree that we took.

n-2 = 6. No need to be noted ~~the last node to be noted~~

(*) In this algo, ~~the~~ ~~if you could~~ be noted is n-1 (as ~~could~~ can also be seen in above ex.).

~~wrong~~.

It's b/c the highest labeled node will never be deleted.

(*) Now lets see how to map a vector to a spanning tree

$b = (1, 2, 2, 2, 2, 7)$

list of nodes in tree, which we'll create

$l = (0, 1, 2, 3, 4, 5, 6, 7)$

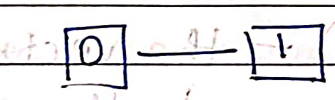
So, pick the least labeled node from l that ~~is has to not been picked before & which~~ is not in vector b.

Here it is 0

↳ update $l = (1, 2, 3, 4, 5, 6, 7)$

↳ ∵ 0 has been picked

Now, connect 0 to ~~least labeled~~ 1st node in vector b which is 1 & delete that node from b



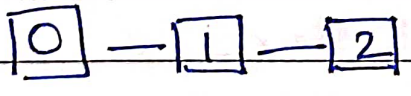
update $b = (2, 2, 2, 2, 7)$

again choose the least labelled node in l which is NOT in b ⇒ 3 !

update $l = (2, 3, 4, 5, 6, 7)$

connect 1 to ~~least labeled~~ 1st node in b (current)

which is 2.

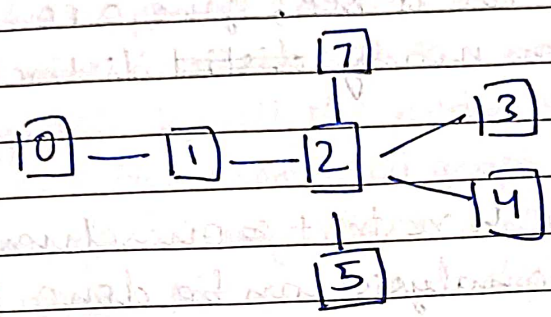


update $b = (2, 2, 2, 7)$

Continue this until b is empty.

$l = (2, 3, 4, 5, 6, 7)$ $b = (2, 2, 2, 7)$

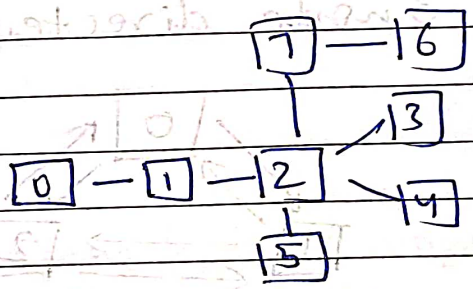
when b becomes empty



l will ~~have~~ be left with nodes n-1 & n-2 i.e 7 & 6

connected n-2 to n-1

Finally



—x—

Hence, we've got a bijection from set of spanning trees & to set of vectors of dim n-2. As mentioned earlier we'll skip the proof for now.

Lucky Date / /
Page

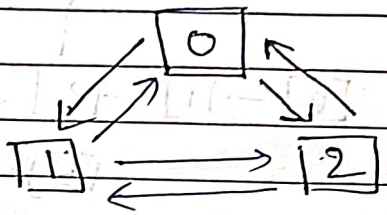
(#)

Number of Paths

In this chapter we'll see; given a complete graph of n nodes how many ~~distid~~ distinct ~~shortest~~ simple paths are there.

Even though, we'll restrict ourselves to directed graphs, similar analysis can be done for undirected graphs as well.

Ex of a 3 node directed complete graph.



i.e. each node will have (n-1) outgoing & (n-1) incoming edges

Let n_l be the number of paths of length l.

$\Rightarrow n_1 = n_{C_1} \times (n-1)_{C_1}$

$n_2 = n_{C_1} \times (n-1)_{C_1} \times (n-2)_{C_1}$

$n_3 = n_{C_1} \times (n-1)_{C_1} \times (n-2)_{C_1} \times (n-3)_{C_1}$

$\therefore n_{n-1} = n_{C_1} \times (n-1)_{C_1} \times (n-3)_{C_1} \text{ ---- } \times 1_{C_1}$

Hence, the total no. of ~~distinct~~ distinct simple paths in a complete graph with n node

$$= \sum_{i=1}^{n-1} n_i = n! \sum_{i=2}^{n-1} \frac{1}{(n-i)!}$$

#

Lucky

Graph Traversal

→ So, what we mean by traversal is, given a node s, we want to visit all the nodes, reachable from s in the graph. If all the nodes of the graph are not reachable from s. ⇒ NOT a connected graph ⇒ apply same algo to all its connected component.

→ In case of binary trees we knew that given the root node, there were 4 different types of traversals namely:-

a.) Preorder traversal
b.) inorder "
c.) Postorder "
d.) Levelorder "

Similarly we'll see few different ways of traversing a graph. Our focus is going to be on the connected graphs.

The algo that we'll implement will be applied only if the graph is in the form of adjacency list. However, this algo will be indistinct to directed and undirected graphs.

Now, in this chapter we'll only deal with a generic overview of how to do the traversal, given the graph in the form of an adjacency list. Do not panick if the overview seems vague. You don't need to understand it completely. Just take a superfluous look at it. This overview will get completely clear after you've read the next 2 chapters.

Before starting the overview, there're some things to consider. Just like we used an additional data structure (stack & queue) to perform the traversals in Binary tree, similarly we'll use an additional data structure (DS) to perform graph traversals.

↓ can be stacks, queues etc.

We'll have a Delete fun^n which will work A/T the DS we use.
For ex if DS is a stack the 'Delete' will delete the last input, whereas it will delete the 1st input if DS is a queue.
Add(DS, s) will add s to DS.
And also there'll be a phi array phi[ ]. Its vth index will contain an info corresponding to the vth labeled node in the graph.

Now, after we're done traversing the graph, we could make a tree whose edges will be (phi[v], v) for various v's.

For ex when you perform a Breadth first search (BFS) traversal, you can make a BFS tree using the phi array.
edge (u, v)
For every Every node is deleted at exactly once from DS. You'll consider all outgoing edges of the deleted nodes, after you delete the node (from DS, NOT from graph). So every edge will be considered at exactly once.

→ source node

Phi[s] will always be -1

We're ready to see the pseudo code now.

—x—

→ Phi[s] = -1; Add(DS, s);

while(DS) → while DS is not empty

{

I [u = Delete(DS);

II [For every edge (u, v)
1. if v is already deleted - do nothing
2. if v is not added - add v
3. if v is already in DS - update

III [Phi[v] = u;

}

—x—

In the II part for every edge (u, v)
if v is already in DS - update

(a.) if DS is queue - do nothing → FIFO
b.) if DS is stack - Add to stack → LIFO
c.) if DS is priority queue
→ Min - do decreasekey
→ Max - do increasekey

—x—

Names of traversals based on the DS used.

1.) DS = Queue, then Breadth first Search (BFS)
2.) DS = stack, then Depth first search (DFS)
3.) Min Priority Queue = DS, then Dijkstra
4.) Priority Queue = DS, then Prims → Min → Min spanning tree
→ Max → Max spanning tree

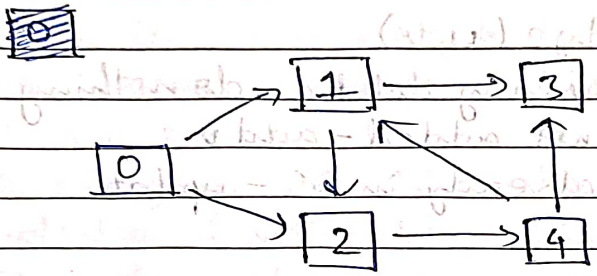
Breadth First Search (BFS)

using Queue

Let us see how BFS works by taking an example. So, we'll maintain a queue (Q), Phi array & an array V[], which will store a flag of whether a node has been deleted from Q or not. Remember for Q it is first in first out. We'll see BFS on directed graphs

—x—

Ex:-



Let the given node s be 0

We initalize all the indices of Phi except index s to be -2; Phi[s] = Phi[0] = -1

indices	0	1	2	3	4
Initial values Phi	-1	-2	-2	-2	-2
Final values Phi	-1	0	0	1	2

indices	0	1	2	3	4
Initial values V	false	false	false	false	false
Final values V	true	true	true	true	true

Initially Q is empty

∵ s is 0, we add 0 to Q

⇒ Q = (0)

then we delete 0 ⇒ V[0] = true;

& ~~look at~~ add the outgoing nodes of 0 i.e 1 & 2

⇒ Q = (1, 2)

We ~~got~~ added 1 by deleting 0 ⇒ Phi[1] = 0

" " 2 " " 0 ⇒ Phi[2] = 0

Now, again we do Delete(Q)

i.e. we delete 1 ⇒ V[1] = true;

& look at the outgoing nodes of 1 i.e. 3 & 2

~~V[2] true~~

~~Phi[3] = 2~~ but ~~Phi[2] ≠ 2~~ V[3] = false

⇒ 2 is already ~~in Q~~ picked but 3 is not

∵ we'll only add 3 to Q ⇒ Q = (2, 3)

We added 3 by deleting 1 ⇒ Phi[3] = 1

Now, again we do Delete(Q) i.e. we delete 2

[⇒ V[2] = true ⇒ Q = (3)]

& look at the outgoing nodes of 2 i.e. 4

V[4] = false, 4 is NOT ~~in Q~~ picked.

⇒ we'll add 4 to Q ⇒ Q = (3, 4)

We added 4 by deleting 2 ⇒ Phi[4] = 2

Again, we delete 3 ⇒ V[3] = true ; Q = (4,)

& there are no outgoing nodes from 3

so do nothing

Then we delete 4 ⇒ V[4] = true; Q = ( )

& look at the outgoing nodes from 4 i.e. 1 & 3

V[1] = true ~~Phi[1] ≠ 2 & Phi[3] ≠ 2~~

& V[3] = true

⇒ 1 & 3 have already been picked once.

⇒ do nothing

Now Q is empty so stop.

—x—x—

Now Phi = (-1, 0, 0, 1, 2)

we can form a BFS tree

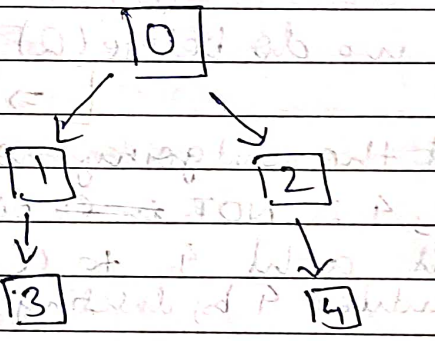
∵ Phi[0] = -1 ⇒ 0 is the root node

Phi[1] = 0 ⇒ 1 is an outgoing node of 0

Phi[2] = 0 ⇒ 2 " " " " " 0

Phi[3] = 1 ⇒ 3 " " " " " 1

Phi[4] = 2 ⇒ 4 " " " " " 2



BFS Tree.

Code for BFS

```
for (i=0; i<n; i++)
{ Phi[i]=-2; V[i]=false;}

Enqueue(Q,s); Phi[s]=-1; V[s]=true;

while(Q)
{
    u=Dequeue(Q);

    For every edge(u,v)

    if(V[v]==false){
        Enqueue(Q,v);
        Phi[v]=u;
        V[v]=true;}
}
```

O(m+n)

(*)
```
while(Q)
u=Dequeue(Q);
For every edge (u,v)
```

Every node is added once. You'll consider all outgoing edges of the deleted node. So every edge will be considered once.

Enqueue → O(1) Dequeue → O(1)
Every ~~node~~ edge is considered once → takes O(m)

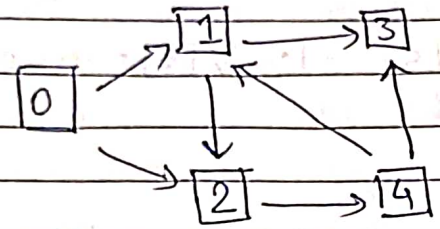
∴ BFS takes O(m+n)
Every node is added once to the Queue → takes O(n) additional space.

Date / /
Page

Depth first Search (DFS) using Stack

Lets look at how DFS works by taking an example.

Ex:-



Let given s be [0].
Let there be stack S

(n+m)O

Let the adjacency list be

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

indices	0	1	2	3	4
Initial Phi	-1	-2	-2	-2	-2
Final Phi	-1	0	1	1	2
Initial V	false ~~true~~	false ~~true~~	false	false	false
Final V	true	true	true	true	true

We first push [0] in S ⇒ S = |0|

then we pop 0 & look at its outgoing nodes
i.e. ~~2 1 2 1~~. 2 & 1.
⇒ V[0] = true
∵ V[2] = false & V[1] = false
⇒ Push 2 & 1
⇒ S = |1, 2|
& Phi[2] = Phi[1] = 0
∵ we pushed 2 & 1 by ~~do~~ popping 0

Now pop 1 & look at its outgoing ~~edges~~ nodes
i.e. 2 & 3
⇒ V[1] = true

∵ V[2] = V[3] = false

⇒ Push 2 & 3

& put Phi[2] = Phi[3] = 1

∵ we pushed 2 & 3 by popping 1

⇒ S = | 3 |
| 2 |
| 2 |

Now, pop 3 & look at its outgoing nodes ⇒ V[3] = true

∵ there are no outgoing nodes

∴ do nothing

⇒ S = | 2 |
| 2 |

Now, pop 2

& look at its outgoing nodes i.e 4

⇒ V[2] = true

∵ V[4] = false ⇒ push 4

∵ we pushed 4 by popping 2 ⇒ Phi[4] = 2

⇒ S = | 4 |
| 2 |

Now, pop 4 & look at its outgoing nodes i.e. 1 & 3

~~but~~ ⇒ V[4] = true

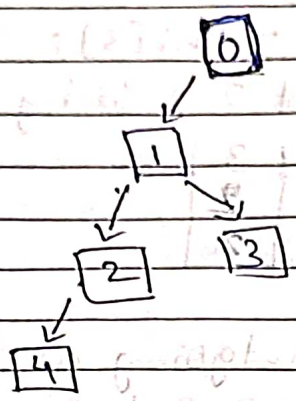
But V[1] = V[3] = true ⇒ do nothing ⇒ S = | 2 |

Now we pop 2 but V[2] is already true

⇒ we do nothing; S becomes empty & we stop.

indices 0 1 2 3 4

Phi = { -1, 0, 1, 1, 2 }



DFS tree.

(*) ~~while (S). v = Pop~~ Code for DFS

```
for(i=0; i<n; i++)
{ Phi[i] = -2;  V[i] = false; }

Push(S, s); Phi(s) = -1;

while(S){
      u = Pop(S);
      if(V[u] == false){
            V[u] = true;
            For every edge(u, v)
              if(V[v] == false){
                    Push(S, v);
                    Phi[v] = u; }
              }
      }
```

O(m+n)

O(m)

(*) while(S)
u = Pop(S)
for every edge (u, v)

You'll consider all outgoing edges of the popped nodes, after you popped the node for the 1st time. So every edge will be considered once

Push → O(1) Pop → O(1)

Total no. of push/pop will be O(m).
Every edge is considered at ~~one~~ once take O(m).

∴ DFS takes O(m)

When you consider an edge, the end node may be pushed into the stack → takes O(m) additional space.

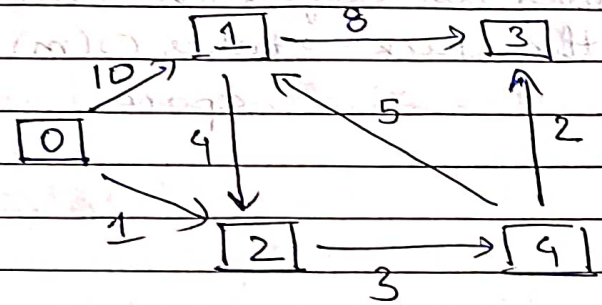
Dijkstra's Algorithm for Shortest Paths

For this algo, the weight of each edge should be positive. The data structure that we'll use, is Priority Queue (Ex is binary heap)
Minimum

So, for this algo we'll have 3 arrays Phi[], V[] (these 2 we already know); and D[]. D will store the shortest distance of each node from the source node. And of course we need maintain a Priority Queue (PQ[])

Let us see how Dijkstra's Algo works by taking an example.

Ex:-



weight of each edge is given
let the source s be 0.

indices	0	1	2	3	4
Initial value Phi	-1	-2	-2	-2	-2
Final value Phi	-1	4	0	4	2

Initial value V	false	false	false	false	false
Final value V	true	true	true	true	true

Min Priority Queue will be maintained on the basis of its corresponding value in the array D. For instance if 0, 1, 2, 3 nodes are to be added in PQ[] & D = {0, 5, 2, 3} ⟶ D[3] = 1
⇒ PQ = {0, 2, 3, 1}

indices	0	1	2	3	4
initial D	0	∞	∞	∞	∞
Final D	0	9	1	6	4

} ⟶ we do this by initialising each index of D to ∞ & then doing DecreaseKey(s, 0). In this case s is 0.

Initially PQ is empty ⇒ PQ = ( )
Then we add 0 to PQ ⇒ PQ = (0,)

We do a DeleteMin(PQ) i.e. delete 0 & look at its outgoing nodes i.e. 1 & 2.
⇒ V[0] = true;

V[1] = V[2] = false
~~⇒ we add 1 &~~

Now, we'll check

This process is called ~~eg~~ edge relaxation ⟵

If (D[1] > D[0] + weight(0,1))
i.e. ∞ > 0 + 10 (which is true)
⇒ D[1] = 10

Similarly for [2] D[2] > D[0] + weight(0,2)
⇒ D[2] = 1

Now, we adds 1 & 2 one by one and update PQ. ⇒ PQ = (2, 1,)
↳ since D[2] < D[1]

∵ we add 1 & 2 by deleting 0
⇒ Phi[1] = Phi[2] = 0

Again we do a deletemin(PQ) i.e. delete 2 & look at its outgoing node i.e. 4 ⇒ V[2] = true

this is exactly the shortest path to reach from 0 to 1 in the graph also.

—x—

(*) while(PQ)
u = DeleteMin();
For every edge (u, v)

Every node is deleted once.
You will consider all outgoing edges of deleted node. So every edge will be considered exactly once.

Every node is deleted once. → DeleteMin()
" " " considered once → DecreaseKey().

⇒ DeleteMin will be called exactly n times
DecreaseKey " " " atmost m times.

We've ~~least~~ learnt 2 Priority Queues upto now :-

1.) Binary heap
- DecreaseKey → O(logn)
- DeleteMin → O(logn)

⇒ Dijkstra using min binary heap takes O((n+m)logn)

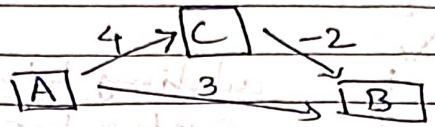
2.) Fibonacci Heap
- DecreaseKey → O(1)
- DeleteMin → O(logn)

⇒ Dijkstra using fibonacci heap takes O(nlogn+m).

Dijkstra's algo

(*) Computes the weight of the shortest path only if the weights are positive.
↳ strictly greater than 0

For ex:-

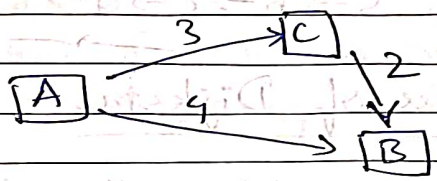


Dijkstra reports that A → B is the shortest path. But the shortest path is A → C → B.

—x—

(*) Dijkstra does not compute the weight of the longest path, by using MaxHeap & simply changing the sign in edge relaxation.

For ex.-



Dijkstra reports that A → B is the longest path. But the longest path is A → C → B.

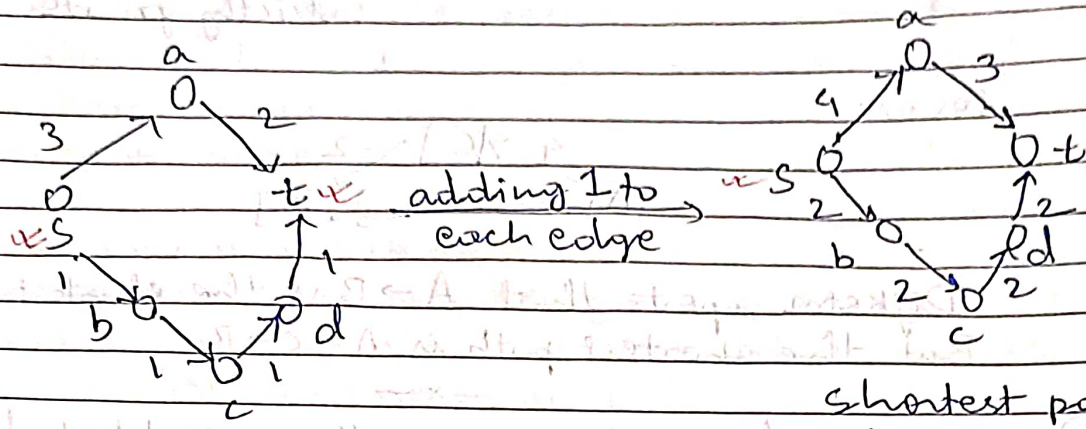
—x—

(*) It must be tempting to assume that shortest paths don't change if a constant is added to the weight of each edge. B/c in that way we can add a constant to each edge of a graph (which has a -ve edge) so that the edges become +ve; compute the shortest path for that new graph & claim that the same path is the shortest path for the original graph also.

However, we see that the shortest paths change

if you add a constant to every edge of the graph.

For ex:-



shortest path from s to t is $s \to b \to c \to d \to t$

shortest path from s to t is $s \to a \to t$

(*) Correctness of Dijkstra

Claim:- $D[v]$ = weight of the shortest path from s to v

Proof:- To prove dijkstra, we make another claim:- if $v_1, v_2, v_3 \dots v_n$ is the order in which the nodes are deleted from the min heap then

$$D[v_1] \leq D[v_2] \leq D[v_3] \leq \dots \leq D[v_n]$$

Proof:- By induction

Base case (i.e. for node s) is trivially true.

Let us assume that the claim is true for $(j-1)$th node, we need to prove that it's

↓e → decrease
↑e → Increase

true for ith node also
For contradiction let us assume that there exists j such that ∃ j > i but $D[v_j] < D[v_i]$;
let j be the smallest such j

When i was deleted $D[v_i] \leq D[v_j]$

Even in the future, if $D[v_j]$ ↓es it can ↓e to
$D[v_j] = D[v_k] + w(k,j)$ for some $k, i \leq k < j$

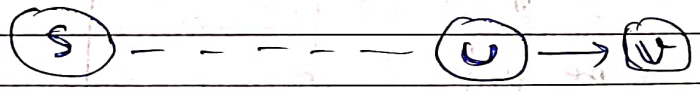
$D[v_j] = D[v_k] + w(k,j) \geq D[v_i] + w(k,j) > D[v_i]$;

lets come to our original claim now.
By induction we see
Base case (for s) is trivially true
Let v be the outgoing node of u

Let us ~~use~~ assume that
$D[u]$ = weight of shortest path from s to u.

As $\forall\, w(u,v) > 0$: $D[u] < D[v]$ so u must be deleted before v

When u was Deleted, edge (u, v) should have been considered & D[v] can't be greater than $D[u] + w(u,v)$



Prim's Algorithm for MST

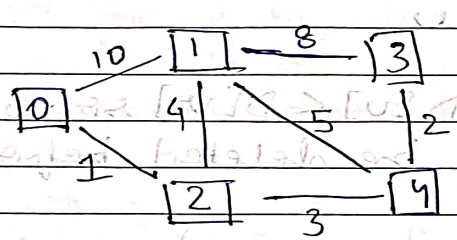
Prim's algo for

As we've already seen, MST is defined only for undirected graph. In this ch we'll see Prim's algo for both Min and Max spanning tree. You'll observe that Prim's algo is very similar to Dijkstra's. However, in contrast to Dijstra's Algo, Prim's algo for minimum spanning tree can be easily modified into max. spanning tree.

The data structure needed for minimum & maximum ~~spanning~~ spanning trees are min & max Priority Queue respectively. We won't look at ~~a~~ a formal proof of Prim's algo. But we'll try to get an intuition as to why it works correctly. Arrays used → Phi, V, D

I Min ST

Let us see how the Prim's algo works



PQ = ( )

↓

empty

indices	0	1	2	3	4
Initial Phi	-1	-2	-2	-2	-2
Final Phi	-1	2	0	4	2
Initial V	false	false	false	false	false
Final V	true	true	true	true	true
Initial D	0	∞	∞	∞	∞
Final D	0	4	1	2	3

Let s be 0

~~Add~~ Add 0 to PQ ⇒ PQ = (0,)

Do a DeleteMin i.e. delete 0 & look at the nodes connected it i.e. 1 & 2
⇒ V[0] = true.

∵ V[1] = V[2] = false
⇒ (check if D[1] > weight(0,1))
b/c ∞ > 10 ⇒ D[1] = weight(0,1)
⇒ D[1] = 10
Similarly D[2] = 1

∵ D[1] & D[2] got updated by deleting 0
⇒ Phi[1] = Phi[2] = 0

For 1 & 2 → Add & update to PQ ⇒ PQ = (2, 1,)
∵ D[2] < D[1]

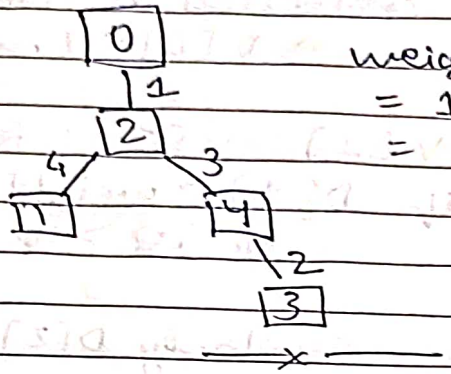
Do a DeleteMin i.e delete 2 & look at the nodes connected to it i.e. 0, 1 & 4
⇒ V[2] = true. ⇒ PQ = (1,)

∵ V[0] is already true ⇒ do nothing for 0.
V[1] & V[4] = false
check if D[1] > w(1,2)
∵ 10 > 4 ⇒ D[1] = 4
Similarly D[4] = 3
⇒ Phi[4] = Phi[1] = 2
Add 4 & update PQ ⇒ PQ = (4, 1,)
& keep doing so until PQ is empty.

indices- 0 1 2 3 4

Finally Phi = (-1, 2, 0, 4, 2)

⇒ Min ST formed will be as follows:-



weight of this Min ST
= 1 + 4 + 3 + 2
= 10

(*) Prim's Algo

→ Computes the weight of the ~~Min ST~~ MST

→ (Phi[v], v) is the set of edges in the MST.

→ Sum of D[v] over all the nodes gives the weight of the MST.

→ Code

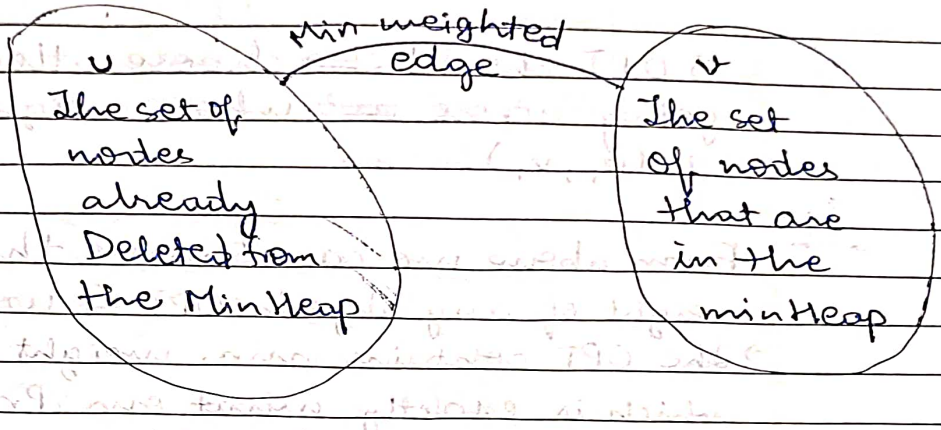
```
→ for(i=0; i<n; i++)
  { Phi(i)=-2; V[i]=false; D[i]=∞; }
  D[s]=0; // DecreaseKey(s,0)
  Phi(s)=-1;
  while(PQ){
      u = DeleteMin(); V[u]=true;
      For every edge(u,v)
      if(V[v]==false && D[v]>w(u,v)){
          D[v]=w(u,v) // DecreaseKey(v, w(u,v));
          Phi[v]=u; }}
```

warning: it will seem vague both in notes and in video.

(*) Intuition for the correctness of Prim's Algo

What we do in Prim's algo is that we're given 2 sets as shown below (of u & v). We choose u_i from set 1 & v_i from set 2 such that (u_i, v_i) forms a minimum weighted edge. These edges form the MinST.

i.e.



Now, let us consider for a moment that MinST does NOT have the min weight and another tree namely optimal ST (OPT) has that min weight.

Now, if we add an edge in the OPT, say (u_1, v_1) from set u & v then it will form a cycle. Thus, OPT will no longer be a tree anymore. To make it a tree again we will need to remove an edge from the cycle. Let that edge be (u_0, v_0)

Suppose $\omega(u_0, v_0) > \omega(u_1, v_1)$

Hence, when we remove (u_0, v_0) then the weight of OPT will become less than the original OPT but

this will make our very assumption (as underlined in the prev pg.) wrong.

If $(w(v_0, v_0) < w(v_1, v_1))$ then the new OPT will have weight greater than the original OPT.

$\Rightarrow w(v_0, v_0) = w(v_1, v_1)$

$\Rightarrow$ OPT should choose atleast one edge going across whose weight is same as $w(v_1, v_1)$.

- From above we can observe that weight of any edge in OPT $\leq w(v_1, v_1)$; $\forall v_i, v_i$
$\Rightarrow$ The OPT contains min. weight edges (v_i, v_i) which is exactly what our Prim's algo does.

$-x-$

(*) Prim's Algo

```
while (PQ)
v = DeleteMin();
For every edge (v,v)
```

Every node is deleted once. You'll consider all outgoing edges of the deleted node. So every edge will be considered exactly once

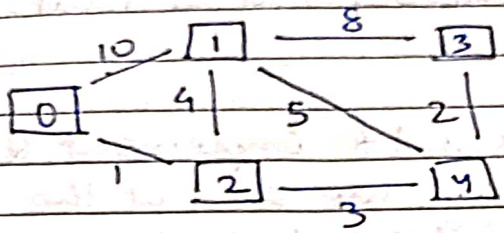
Exactly n deleteMins
atmost m DecreaseKey

Complexity of Prim's using i) Min Heap $\rightarrow O((n+m)\log n)$
ii) Fibonacci Heap $\rightarrow O(n\log n + m)$

II Max ST

How Prim's Algo works

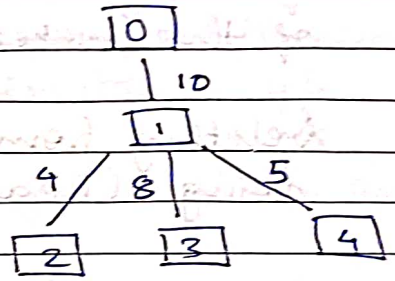
Ex:-



indices	0	1	2	3	4
Initial Phi	-1	-2	-2	-2	-2
Final Phi	-1	0	1	1	1
Initial V	false	false	false	false	false
Final V	true	true	true	true	true
Initial D	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
	0	10	4	8	5

Assuming s to be 0

Maximum spanning tree



DO YOURSELF

—x—

Sum of D[v] over all nodes gives the weight of the MST.

—x—

CODE →

```
for(i=0; i<n; i++)
{Phi[i]=-2;  V[i]=false;  D[i]=-∞;}
D[s]=0; // IncreaseKey (s,0)
Phi[s]=-1;
while(PQ){
    u=DeleteMax(); V[u]=true;
    For every edge (u,v)
    if (V[v]==false && D[v]<w(u,v)){
            D[v]=w(u,v) // IncreaseKey (v, w(u,v));
            Phi[v]=u; }}
```

Lucky Date / /
Page

Implementation of BFS, DFS, Dijkstra & Prims

All we need to understand is that:-

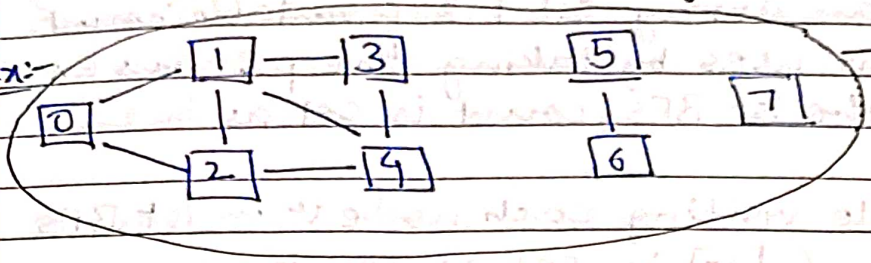
In BFS:-
Linked list is used to implement Queue instead of array. We add at the head of the LL & delete at the tail of the LL.

In DFS:-
Linked list is used to implement stack instead of array. We add & delete at the ~~beginning~~ head of the Linked list.

Deleting from the stack is equivalent to doing (head = head -> next)

Connected Components of an undirected Graph using BFS

(*) Ex:-



→ NOT a connected graph with 3 connected components.

→ A connected component of G is a maximal connected subgraph of G.

→ The set of nodes is the disjoint union of set of nodes in the connected components.

—x—

(*) How we can use BFS to compute the connected components in an undirected graph.

Let us see that by taking the above graph as an ex.

Set of nodes in the above graph
= S = {0, 1, 2, 3, 4, 5, 6, 7}

We choose a node (say 0) & do a BFS. In this way we traverse the nodes 0, 1, 2, 3, 4. We see that nodes 5, 6, 7 are still left.

Hence, we choose a node from the left over nodes (5, 6, 7). Lets say that node is 5 & again do a BFS. Now the nodes traversed are 5, 6. There is still a node left which is 7.

Again we choose 7 & do a BFS. So, we did BFS 3 times to cover all the nodes. Hence, above graph has 3 connected components.

Date ___/___/___
Page ___

(*) Small description of the code that will follow.

We've an array C[] & a variable 'count'. Lets see their uses by taking the previous ex. When we do the 1st BFS, count is set as 1

& while visiting each node v in 1st BFS
C[v] is set as count = 1

indices 0 1 2 3 4
⇒ C = {1, 1, 1, 1, 1,}

i.e. nodes 0, 1, 2, 3 & 4 ~~belong~~ belong to the 1st connected component.

Similarly for 2nd BFS count = 2

indices 0 1 2 3 4 5 6
C = {1, 1, 1, 1, 1, 2, 2,}

nodes 5 & 6 belong to the 2nd connected component.

Finally for 3rd BFS count = 3

indices 0 1 2 3 4 5 6 7
C = {1, 1, 1, 1, 1, 2, 2, 3}

node 7 belongs to the 3rd connected comp.

—×—

→ Coole

```
for (i=0; i<n; i++)
{ c[i]=0; }
count=0;

for (i=0; i<n; i++)
{
   if (c[i]==0){
      count++;
      s=i;
      Enqueue(Q,s);
      c[s]=count;
      while(Q){
         u=Dequeue(Q);
         For every edge (u,v)
         if (c[v]==0){
            Enqueue(Q,v);
            c[v]=count;
            }
         }}
}
```

—x—

→ Complexity

Given a connected graph, the no. of connected components & labeling the connected component to which a node belongs to can be done using ~~BST~~ BFS

in $O(m+n)$ time

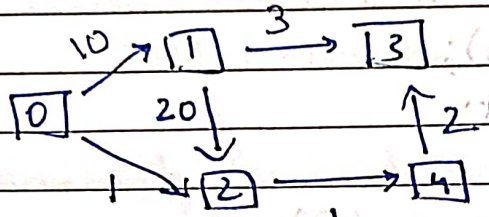
Date / /
Page

Computing Shortest length Path using BFS

Shortest length path

- Given a graph (directed or undirected).
- Given a source node s, find the shortest length path from s to every node in the graph.
- Length of a path is the no. of edges in the path

Ex:-



⇒ Shortest length path from 0 to 3 is $0 \to 1 \to 3$ &

NOT $0 \to 2 \to 4 \to 3$

- We can set the edge weight 1 for each edge & use Dijkstra to compute the shortest length path. But it will take $O((n+m)\log n)$
- We can complete the shortest length path using BFS in $O(n+m)$.

To do this, we'll make a small modification in the code of BFS, which is maintaining an additional array L[ ]

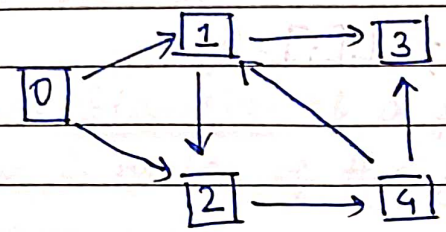
where L[v] = level of vth labeled node in the BFS tree which we'll create.

for for every edge (u, v)

L[v] = L[u] + 1 ; (level of root node = 0)

Every time we update Phi[v], we'll update L[v].
Each index of L will be initialized to ∞ except for the index s. (L[s] = 0)

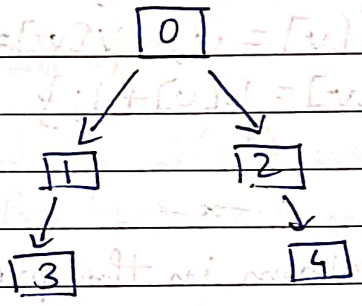
For ex:-



Let s be 0

indices	0	1	2	3	4
Initial Phi	-1	-2	-2	-2	-2
Final Phi	-1	0	0	1	2
Initial L	0	∞	∞	∞	∞
Final L	0	1	1	2	2

BFS tree



—x—

→ The claim is that :-

L[v] = length of the shortest length path from s to v

—x—

(*) L[v] = ∞ (INT_MAX) means node is not reachable from s.

—x—

(*) This shortest length path can be traced by the array Phi[].

CODE →

```
for (i=0; i<n; i++)
{Phi[i]=-2; V[i]=false; L[i]=INT_MAX;}

Enqueue(Q,s);
Phi[s]=-1; V[s]=true; L[i]=0;

while(Q){
    u=Dequeue(Q);
    For every edge (u,v)
    if (V[v]==false){
        Enqueue(Q,v);
        Phi[v]=u; V[v]=true;
        L[v]=L[u]+1;}
    }
```

O(n+m)

—x—

(*) Proof for the claim in the prev. pg.

Proof is by induction

(s) ○ ○ ---- (i) (j) -- (v)

Base case is trivially true

Let us assume that till ith node claim is true

Now for edge (i,j)

Phi[j] will be updated to i

& L[j] will be updated to (L[i] + 1)

{shortest length path from s to i (b/c of our assumption)} {shortest length path from i to j (trivial)}

∴ of course L[j] will be the shortest length path from s to j.

Hence proved

— x —

(*) Distance b/w 2 nodes is weight of the shortest path b/w the nodes.

(*) ~~Dist~~ Diameter of a graph is the maximum distance b/w 2 nodes of the graph

Note:- Diameter of a graph is NOT the same as the longest path in the graph.

— x —

(*) Diameter of a graph

→ Can be found in $O(n^3)$ using Floyd-Warshall's algo. This is the best we can do for a general Graph.

→ However, if all the edges have unit weight then the distance b/w 2 nodes is the length of the shortest length path b/w the nodes.

graph's diameter

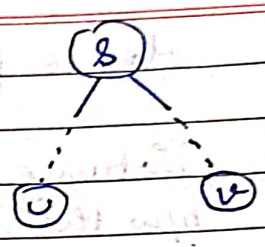
→ This can be found in $O((n+m)n)$ time by calling the BFS from each node of the graph.

→ Let G be an undirected connected graph. (all of its edges are of unit weight).

→ Let L be the maximum level of any node when applied BFS from a node s.

→ Let d be the diameter of the graph and let u & v be the nodes such that d is the length of the shortest length path b/w u & v

d = length of the shortest length path b/w v & u.



$\Rightarrow d \leq 2L[u] + L[v]$

$L[u] \leq L \;\&\; L[v] \leq L$

$\Rightarrow \underline{d \leq 2L} \;\&\; \underline{L \leq d}$ ($\because$ d is diameter)

$\downarrow$ $\downarrow$

$d \leq 2L$	$L \leq d$
true only for undirected graph with unit edges	true for both directed & undirected graphs with unit edges.

$\Rightarrow$ For undirected graph with unit edges.

$$L \leq d \leq 2L$$

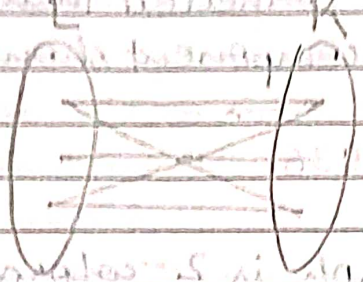
—x—

$\Rightarrow$ There is a linear time approximation algo to compute the diameter of an undirected graph with unit edges.

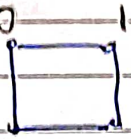
c(v) denotes the colour of the node labelled v.

(17) Bipartite Graphs

(*) An undirected graph is said to be a bipartite graph, if the set of nodes can be divided into 2 disjoint sets say L & R such that every edge has exactly one end point in L & the other end point in R.



-x-

→ Ex:- 

is a bipartite graph (nodes 0 & 2 can be put in L while node 1 & 3 in R)

 is NOT a bipartite graph

-x-

(*) Colorable Graphs

- Let there be a set of k different colours {1, 2, ---, k}. An undirected graph is said to be a k colorable ~~coloured~~ graph, if every node can be assigned one of the colours from {1, 2, 3, --- k} such that for ~~or~~ every edge (u, v) $c(u) \neq c(v)$; $k > 0$

- If $k > 2$, checking if a graph is k colourable or not is a NP-hard problem.

⊛ Bipartite Graphs

- An undirected graph is Bipartite iff it is a 2-colorable. The proof is nothing but simple observation.
- If the graph is not connected then for it to be Bipartite, its each component should be bipartite.

—x—

⊛ König Theorem, 1936

- An undirected graph is 2-colorable iff it does NOT contain an odd cycle.

P – Graph is 2 colorable
Q – It does not contain odd cycle

We need to prove $P \iff Q$

For this first we'll prove $P \Rightarrow Q$ or ~~$\neg P \Rightarrow \neg Q$~~ $\neg Q \Rightarrow \neg P$

$\neg Q \Rightarrow \neg P$ corresponds to:-
If the graph has an odd cycle then it is not 2 colorable.

Let $\neg Q$ be true
Let each node be denoted by either red (R) or blue (B). So, if we start writing the colors of ~~th~~ each node, it'll look like.

in the cycle

colors → R B R B ------ R
nodes → 0 1 2 3 n-1

We see that $c(0) = c(n-1)$
but 0 & n-1 are adjacent nodes
$\Rightarrow$ it is not a 2-colorable graph.

Now, we'll prove $Q \Rightarrow P$

$Q \Rightarrow P$ corresponds to:-
if a graph does not contain odd cycle
then it is a 2 colorable graph

Let Q be ~~be~~ true
So, fix a node v

Now,
$L = \{u$; there is odd length path from v to $u\}$
$R = \{u$; there is even length path from v to $u\}$

A node can either be in L or R, but NOT both.
If ~~it~~ it happens so that a node is in both L & R
then the graph will contain an Odd cycle, which
is not possible if Q is true.

$\Rightarrow$ There is no edge b/w 2 nodes of L
" " " " " 2 " " R.

$\therefore$ it is a 2 colorable graph.

Date ___/___/___
Page ___

(*) Code to check if a graph is 2 colorable or NOT

→ Small description :-

Instead of using a V[], we'll use only the L[] to do BFS. This L[] will do its original job as well as the job of V[]. We'll initialize each index of L[] to -1.
At any time if for a node v
L[v] < 0 ⇒ v has not been visited yet
⇒ we will add v to the queue
At last we'll check for each edge (u, v)
if both L[u] & L[v] are odd or
" " " " " " even
then the given graph is NOT a bipartite graph.

—x—

CODE →

```
for(i=0; i<n; i++)
{  L[i] = -1; }

Enqueue(Q,s);  L[s] = 0;
while(Q)                                   O(m+n)
{  u = Dequeue(Q);
   For every edge (u,v)
      if(L[v]<0){
            Enqueue(Q,v); L[v] = L[u]+1; }
}
```

For every edge (u, v)
if (L[u] %2 == L[v] %2) then the given graph is NOT a bipartite graph.

—x—

(*) We can check if a given graph is a bipartite or NOT in linear time.

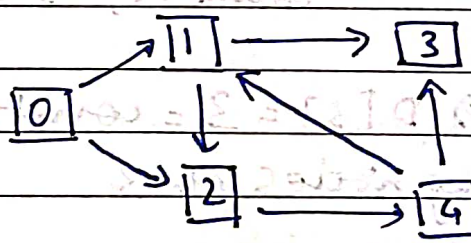
Recursive DFS

Recursive DFS uses the stack implicitly. You don't need to maintain a stack explicitly. This makes the implementation simpler and easier. When we see this recursive algo., we'll also compute 2 important quantities for each of the nodes in the graph which are known as discovery times and finish times. This will be in addition to computing Phi[]. Once we learn to compute discovery time and finish time, in later chapters we'll see some interesting applications of DFS using these.

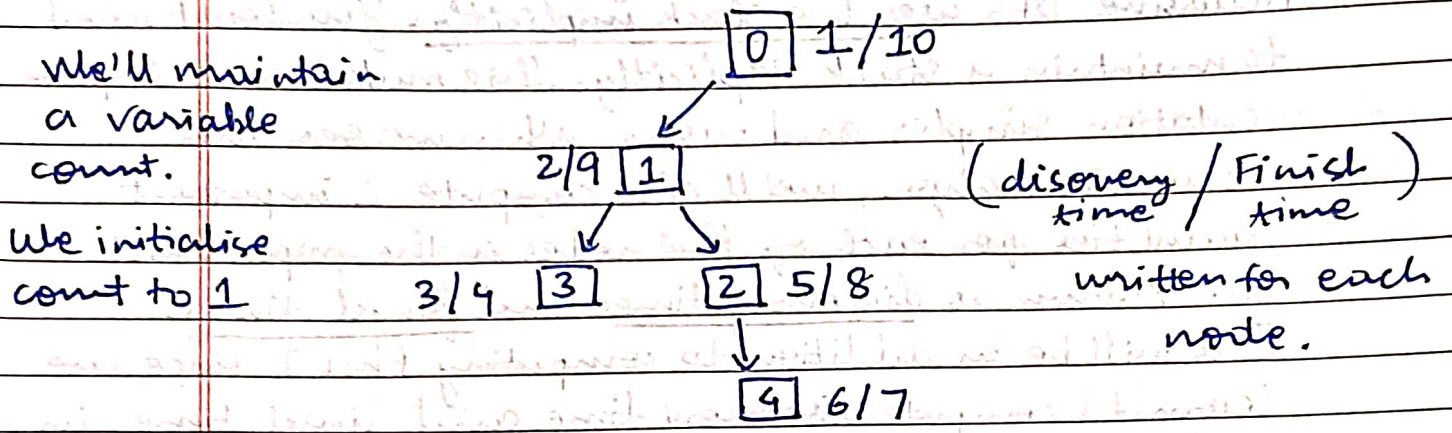
After we've seen the recursive DFS, we'll see how to compute the discovery & finish times using iterative DFS.

To maintain the discovery time of each node, we'll use an array D[]. Similarly for finish time → F[]

Let us see how to calc. discovery & finish time by taking an example.



The DFS tree for the directed graph shown in prev. pg.



We'll traverse each node using DFS in the above DFS tree.

—x—

Note:- The order in which we visit each node in the DFS tree will be the same as the order in which we visit the node in the graph using DFS.

—x—

So 1st we visit [0] ⇒ D[0] = count = 1
Now count++;
⇒ count = 2

then we visit [1] ⇒ D[1] = count = 2
count++; ⇒ count = 3

then we visit [3] ⇒ D[3] = 3 = count; count++
" " " outgoing nodes of 3
but [3] does NOT have any.
⇒ all the outgoing nodes of [3] have been visited
⇒ F[3] = 4 = count; count++

Then we come back to 1 & look at its other outgoing node i.e. [2]

⇒ D[2] = count = 5

count++

Then we look at [4] ⇒ D[4] = count = 6 ; count++

" " " " outgoing nodes of [4]

but [4] does not have any

⇒ F[4] = count = 7 ; count++

Then we come back to [2]

All the nodes of [2] have been visited

⇒ F[2] = count = 8

count++

Then F[1] = 9 & F[0] = 10.

—x—

→ CODE for recursive DFS

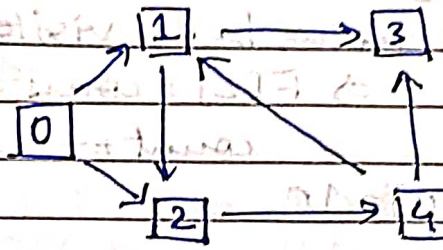
→
```
DFSvisit(G)
{
  for(i=0; i<n; i++)
  { Phi[i]=-2; D[0]=0; V[i]=false; F[i]=0;}

  count=1;
  for(i=0; i<n; i++)
  { if(V[i]==false)
    { Phi[i]=-1; DFS(G,i);}
    }
}
```

O(n+m)

→ DFS(G,u){
V[u] = true;
D[i] = count++;
For every edge (u,v)
if(V[v] == false)
{Phi[v]=u; DFS(G,v);}
F[u] = count++;
}

—x—

(*) How the iterative algo works



indices	0	1	2	3	4
Phi(initial)	-1	-2	-2	-2	-2
Final Phi	-1	0	1	1	2
InitialV	false	false	false	false	false
FinalV	true	true	true	true	true
InitialD	0	0	0	0	0
FinalD	1	2	5	3	6
InitialF	0	0	0	0	0
Final F	10	9	8	4	7

Let s be 0

Stack S is empty at first

count = 1

First, we push s into S i.e 0 ⇒ D[0] = count = 1
~~⇒ V[0] = false~~ count++

Then we pop 0 & look at its outgoing nodes.
i.e 2 & 1 ⇒ V[0] = true
∵ V[2] = V[1] = false
⇒ Phi[2] = Phi[1] = 0
We again push 0 & then
push 2 & then ⇒ S = | 1 / 2 / 0 | ⇒ D[2] = 2 & D[1] = 3
count++ and push 1 & count++

First, we push 8 i.e. |0| ⇒ S = |0|

then we pop 0 & look at its outgoing nodes
i.e. 2 & 1
⇒ V[0] = true ; D[0] = count = 1
update count = 2

Again we push 0.
∵ V[1] = V[2] = false
Phi[1] = Phi[2] = 0;
& then we push 2 &
" " " 1 ⇒ S = |1, 2, 0|

Now, we pop 1 & look at its outgoing nodes
i.e. 2 & 3
⇒ V[1] = true ; D[1] = count = 2
update count = 3

Again we push 1
∵ V[2] = V[3] = false
⇒ Phi[2] = Phi[3] = 1
& then we push 2 &
" " " 3 ⇒ S = |3, 2, 1, 2, 0|

Now, we pop 3 & look at its outgoing nodes
but it doesn't have any
⇒ V[3] = true ; D[3] = count = 3
update count = 4

Again we push 3
∵ no outgoing node
⇒ we do nothing ⇒ S = |3, 2, 1, 2, 0|

Again we pop 3
∵ V[3] is already true
⇒ F[3] = count = 4
update count = 5

⇒ S = | 2 |
| 1 |
| 2 |
| 0 |

Now, we pop 2 & look at its outgoing nodes i.e. 4
⇒ V[2] = true; D[2] = count = 5
update count = 6

Again we push 2
∵ V[4] = false
⇒ Phi[4] = 2
& then we push 4 ⇒ S = | 4 |
| 2 |
| 1 |
| 2 |
| 0 |

Now, we pop 4 & look at its outgoing nodes but it doesn't have any
⇒ V[4] = true; D[4] = count = 6
update count = 7

Again we push 4
∵ No outgoing nodes
⇒ do nothing

⇒ S = | 4 |
| 2 |
| 1 |
| 2 |
| 0 |

Again, we pop 4

but v[4] is already true

$\Rightarrow$ F[4] = 7 = count

update count = 8

$\Rightarrow$ S = | 2 |
| 1 |
| 2 |
| 0 |

Now, we pop 2

but v[2] is already true

$\Rightarrow$ F[2] = count = 8

update count = 9

$\Rightarrow$ S = | 1 |
| 2 |
| 0 |

Now, we pop 1

but v[1] is already true

$\Rightarrow$ F[1] = count = 9

update count = 10

$\Rightarrow$ S = | 2 |
| 0 |

Now, we pop 2

but v[2] is already true

& F[2] is also updated

as F[2] $\neq$ 0

Hence we do NOTHING

$\Rightarrow$ S = | 0 |

Date / /
Page

Atlast we pop 0 but V[0] is already true
⇒ F[0] = count = 10
update count = 11
⇒ S = ⌊⌋

∵ S is empty we STOP;

—x—

(*) Code

```
→ DFSvisit(G)
{
  for(i=0; i<n; i++)
  { Phi[i]=-2; D[i]=0; V[i]=false; F[i]=0; }

  count=1;
  Push(S, s); Phi[s]=-1;

  while(S){
    u = Pop(S);
    if(V[u]==false){
      V[u]=true; Push(S,u);
      D[u]=count++;
      for every edge(u,v)
        if(V[v]==false){
          Push(S,v);
          Phi[v]=u;}
        }
    else if(F[u]==0) F[u]=count++;
  }
}
```

—x—

n = no. of nodes

DFS Properties of finish time

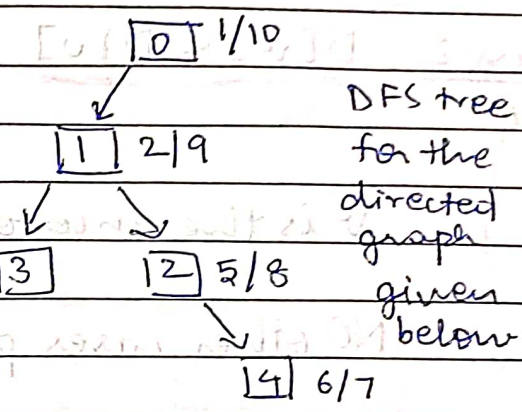
In this chapter we'll look at 2 theorems which exploit the finish time of the vertices/nodes.

The array D[ ] will have n distinct values
" " F[ ] " " " " " "

⇒ Every node is associated with a closed interval
[D[v], F[v]]

Observation

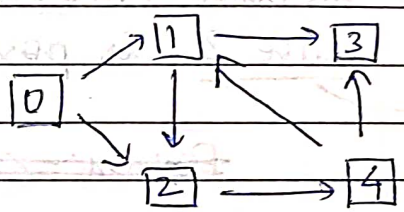
Let u & v be 2 nodes such that D[u] < D[v] then one of the following should be true.



Case 1 F[u] < D[v]

u v

In this case niether u is the ancestor of v nor v is the ancestor of u. (in the DFS tree).
For ex nodes 3 & 4 in the given DFS tree.



Case 2 F[v] < F[u]

v

u

In this case u is the ancestor of v (in the DFS tree). For ex nodes 1 & 3.

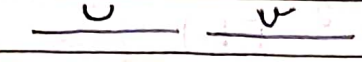
Note :- In the observation given in prev. pg, no other cases are possible

Date __/__/__
Page

(*) The observation in the prev. pg can be rephrased as :-

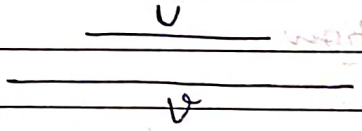
Let u & v be 2 nodes such that F[u] < F[v] then one of the following should be true.

Case 1 F[u] < D[v]



Niether u is ancestor of v nor v is ancestor of u (in the DFS tree).

Case 2 D[v] < D[u]



v is the ancestor of u (in the DFS tree).

NO other cases possible

—x—

(*) In case (u,v) is an edge in the graph then only case 2 is possible

~~For ex :- u = node 1 ; v = node 2~~
~~in the given directed graph.~~

—x—

Theo 1 Let (u,v) is an edge in the graph such that F[u] < F[v] then there must be a path in the graph from v to u.

Th^m 2 If there is a path from u to v in the graph s.t. F[u] < F[v] then there must be a path from v to u.

—x—

(*) Th^m 2 is a generalization of Th^m 1.

→ This is known as topological sorting

Topological Sort of a DAG

(*) DAG (Directed acyclic graph) is a directed graph which does not contain a cycle.

—x—

(*) Apply DFS on G, then list the nodes in the ↓ing order of finish times

—x—

You can do this by initializing a variable 'c' to 1 in the 'DFSvisit' function (the recursive one) & also an array L[].

In the recursive 'DFS' fun^n where we do

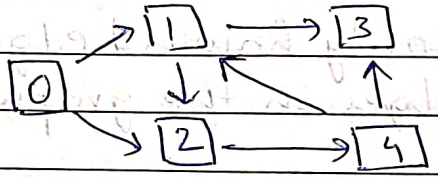
```
F[u] = count;
```

After this line you can add

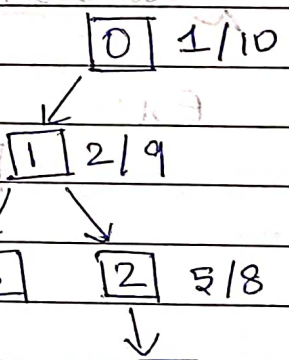
```
L[n-c] = u; c++;
```

—x—

G =



DFS tree



L = {0, 1, 2, 4, 3}

Let there be i & j such that i < j & let u = L[i] & v = L[j] then F[u] < F[v]

⇒ if there exists a path from u to v in the graph then there must be a path from v to u in the graph.

That is there must exist a cycle in the ex in the prev. pg.

$v = [1]$ $u = [2]$

& there exist a path from [2] to [1]
As can be seen there is cycle in the graph.

—x—

(*) Result of above observation

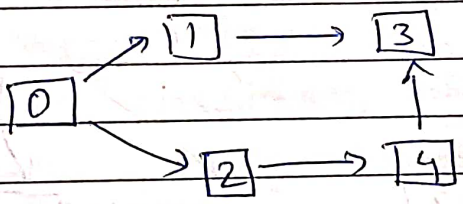
If we list the nodes of a DAG in the ↓ing order of finish times.

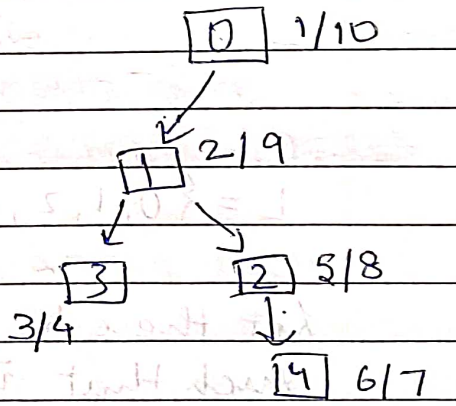
$i < j$; $v = L[i]$; $u = L[j]$

Then ~~can~~ there cannot exist a path from u to v

⟹ There will be only forward edges & there will be no backward edges in the graph

Ex:-





$L = \{0, 1, 2, 4, 3\}$

Strongly Connected Component of a directed Graph.

We talked about connected component in case of undirected graphs. B/c in that case if there exists a path from node u to v then there must exist a path from v to u. However, it's not the same with directed graphs. Hence, comes the concept of strongly connected components (SCC).

—x—

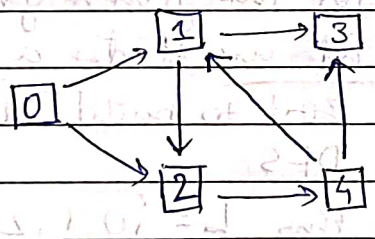
* A directed graph is disjoint union of strongly connected components.

A SSC of a directed graph is a sub graph such that :-

1. Given u and v in same SCC, there must lie a path from u to v & a path from v to u in the graph.
2. Given u in SCC & v not in same SCC, there should not lie a 2 way path b/w u & v. There may be a one way path b/w them.

—x—

Ex :-



SSCs in the given graph are :-

i). {0}

ii) {1, 2, 4} iii) {3}

—x—

* Transpose of a graph

Given G(V, E)

For every edge (u, v) add an edge (v, u) in E^T

Then $G^T(V, E^T)$ is called the transpose of the graph.

Given an adjacency list of G, adjacency list of G^T can be formed in following way:-

Let us say we look at the outgoing nodes of a node u in the adjacency list of G. Let the 1st outgoing node be v.
Now, we go to the v-th index in the adjacency list of G^T & add u. Keep doing this to complete G^T.

As can be observed, transpose of a graph can be found in $O(n+m)$;

—x—

Note:- Adjacency matrix of G^T = Transpose of adjacency matrix of G

—x—

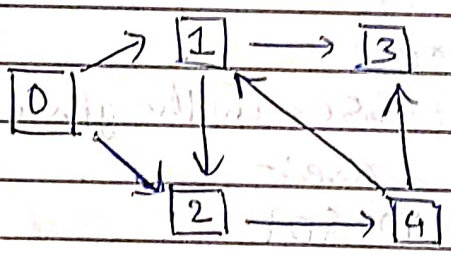
(*) SCCs of G & G^T are same

—x—

(*) How does the algo work to compute the no. of SCCs in a given directed graph work.

Ex:-

G =

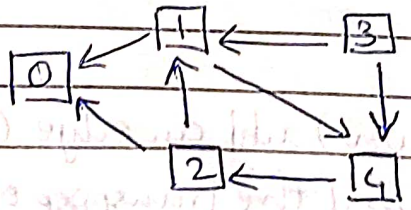


For the given graph 1st we'll do a topological sort to build L[] by doing DFS

For this $L = \{0, 1, 2, 4, 3\}$

Now we create the transpose of a given graph

G^T =



Now by using the same L[] that we built we will do a DFS for each node in G^T.

⇒ indices → 0 1 2 3 4

$L = \{0, 1, 2, 4, 3\}$

We start with $L[0] = 0$ & do a DFS

No other nodes can be visited

⇒ SCC1 = {0}

Then we come to $L[1] = 1$ & do a DFS

⇒ nodes 1, 2 & 4 are visited

⇒ SCC2 = {1, 2, 4}

No wonder, that while doing DFS from node 1, we'll encounter 0, but 0 has already been visited so we do nothing.

Now, we come to $L[2] = 2$ but node 2 is already visited. Similarly for $L[3] = 4$

When we reach $L[4] = 3$, it's not yet visited

Do a DFS ⇒ nodes 0, 1, 2, 4 & 3 will be visited but we've already encountered 0, 1, 2 & 4 earlier ⇒ we do nothing

⇒ SCC3 = {3}

∴ The given graph has 3 SCCs;

i) {0} ii) {1, 2, 4} iii) {3}

Date ___/___/___
Page ___

→ Code to build L[] for Graph

→
```
DFSvisit(G)
{
  for(i=0; i<n; i++)
  {Phi[i]=-2;  D[i]=0;  V[i]=false;  F[i]=0;}

  count=1; c=1;
  for(i=0; i<n; i++)
  { if(V[i]==false)
      { Phi[i]=-1; DFS(G,i); }
  }
}
```

→
```
DFS(G,v){
  V[v]=true;
  D[v]=count++;
  For every edge(v,v)
    if(V[v]==false)
      {Phi[v]=v; DFS(G,v);}

  F[v]=count++; L[n-c]=v; c++;
}
```

→ Code to do a DFS for G^T & count the no. of SCCs.

→ DFSvisit (G^T)
{
for(i=0; i<n; ++i)
{ V[i]=false;}

scc=0
for(i=0; i<n; ++i)
if(V[i]==false)
{scc++; DFS(G^T, L[i]);}

→ DFS(G^T, u){
V[u]=true;
SCC[u]=scc;
For every edge (u, v) in G^T
if(V[v]==false)
DFS(G^T, v);
}

→

(*) Correctness of algo.

To prove that our algo works correctly, we'll need to prove that what we calculated are actually SCCs. And to prove this we need to show the following 2 conditions:—

i) Given u and v in same SCC, there must be a path from u to v & a path from v to u in the graph.

ii) Given u in SCC & v not in same SCC, there should not be a

path from u to v and a path from v to u in the graph.

—x—

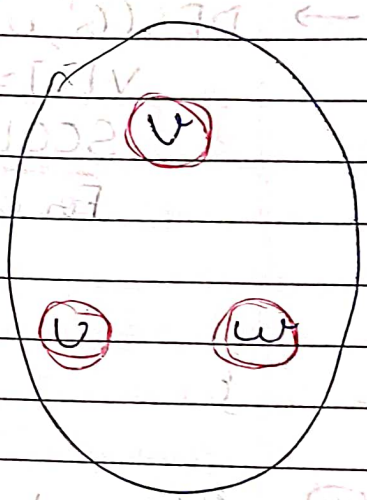
Proof for condition 2

In our algo, we move on to the new SCC only when we can't find any path connecting the nodes which are still not covered. And we don't count the nodes which we've encountered in old SCCs.

i.e. if $i < j$ then there can't be any edge from SCC[i] to SCC[j] in the G^T.

—x—

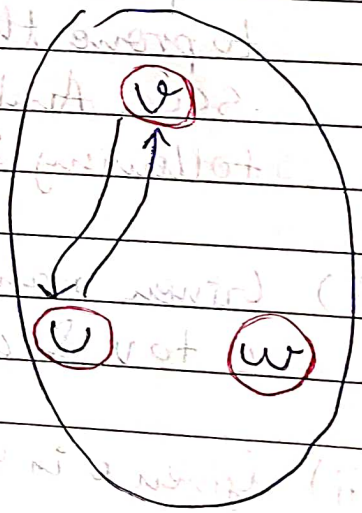
Proof for condition 1



Let u and w be 2 nodes in the same SCC.
Let this SCC obtained by applying DFS starting at the node v

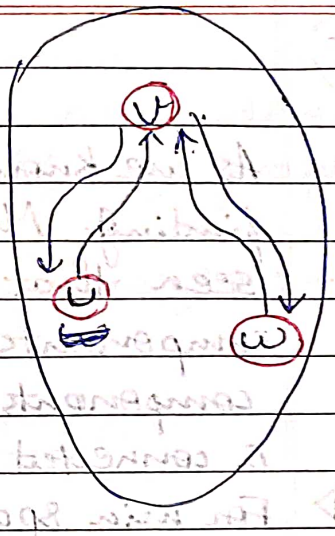
$\Rightarrow F[v] > F[u]$ & $F[v] > F[w]$

In G^T there exists a path from v to u



So in G there must exist a path from u to v. But we know that $F[v] > F[u]$
$\Rightarrow$ there exist a path from v to u in graph G as well

Similarly,
In G^T there exists a path from v to w.
So in G there exist a path from w to v.
But F[v] > F[w]
⇒ there exists a path from v to w in G.



Remember { Connected component → CC → undirected graph
SCC → directed graph

Date / /
Page

Kruskal's Algorithm

As we know already that this algo. is used for finding Min & Max spanning tree. We've also seen that adding an edge across the connected components reduces the no. of connected components by 1, while adding an edge within a connected component introduces a cycle.

→ For min spanning tree
So, what we do here now is, initially we'll have all the vertices of the graph (but no edges), i.e. intially these vertices are not connected.

Now, we'll arrange the edges of the graph in ↑ing order (A/T their weights). We'll ~~start~~ consider each edge and ask a question whether these end points will go across the CC or within the CC. If it goes across the CC, then we'll add this into the Min spanning tree. However, if they go within the CC, then we won't add it b/c adding it will introduce a cycle which will in turn dissatisfy the very defⁿ of a tree.

—x—

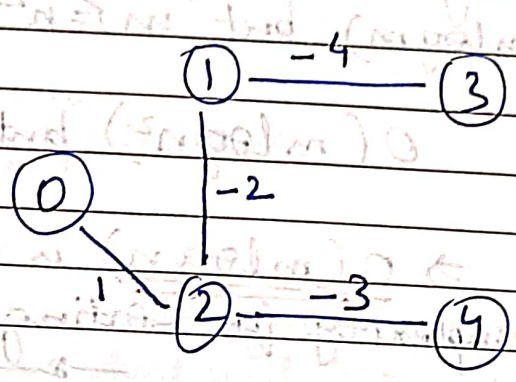
If instead of considering the edges in ↑ing order, we consider the edges in ↓ing order & perform the same operations as above after that, we'll get a max spanning tree.

(*) For example

The following are the edges of ~~the~~ an undirected graph given in ↑ing order A/T their weights :-

→ i j weight

✓ 1. (1, 3, -4)
✓ 2. (2, 4, -3)
✓ 3. (1, 2, -2)
✗ 4. (1, 4, -1)
✗ 5. (3, 4, 1)
✓ 6. (0, 2, 1)
✗ 7. (0, 4, 3)
✗ 8. (0, 1, 4)
✗ 9. (2, 3, 5)
✗ 10. (0, 3, 6)

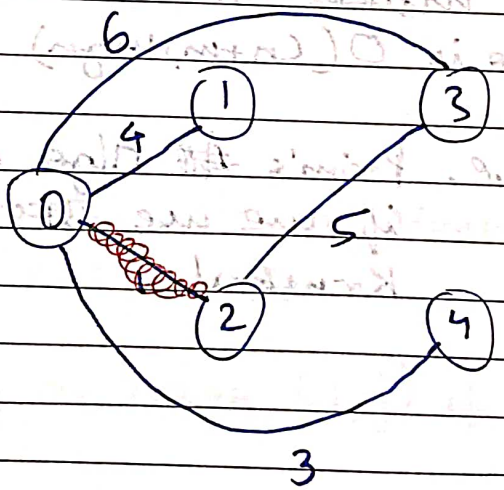


Weight of the min spanning tree thus formed = -8

—x—

Now, taking the edges of the same undirected graph in ↑ing order.

✓ 1. (0, 3, 6)
✓ 2. (2, 3, 5)
✓ 3. (0, 1, 4)
✓ 4. (0, 4, 3)
✗ 5. (0, 2, 1)
✗ 6. (3, 4, 1)
✗ 7. (1, 4, -1)
✗ 8. (1, 2, -2)
✗ 9. (2, 4, -3)
✗ 10. (1, 3, -4)



Weight of Max spanning tree thus formed = 18

(0, 2) → NOT an edge

→ Kruskal's algo comes under the category of greedy algo. We'll see the correctness of Kruskal's algo when we'll revisit it under greedy algo.

—x—

(*) Sorting the edges based on their weights takes $O(m \log m)$ but $m \leq n^2$

$$O(m \log n^2) \text{ but } O(\log n^2) = O(2 \log n)$$
$$= O(\log n)$$

$\Rightarrow O(m \log n)$ is the complexity for sorting the edges

—x—

Q.) How do you check if the given edge (u, v) goes across the CCs or goes within the CC?

Ans We can apply BFS/DFS from u on the MST & check if v is connected to v

This makes Kruskal's Algo $O(mn)$ but Prim's Algo is $O((n+m)\log n)$

$O(mn)$ ↓ using BFS/DFS

i.e. Prim's ~~Algo~~ Algo is much more efficient. if we use BFS/DFS to implement Kruskal

Disjoint Union 1

In this chapter we'll study a data structure called disjoint union data structure.

—*—

Let us assume we're given a universal set $U = \{0, 1, 2 \dots n-1\}$; cardinality of $U = n$

(*) We'll perform 3 kinds operations:-

1.) MakeSet

{It will make n singleton disjoint sets.

Analogous to n CCs that we start with to form MST where each nodes is a CC, initially.

2.) Disjoint (i, j)

{It will let us know whether the elements i & j belong to the same set.

Analogous to answering whether i & j nodes belong to 2 different CCs or same.

3.) Union (i, j)

It will replace the sets containing i & j with the union of these 2 sets.

Analogous to adding an edge (i, j) if i & j belong to 2 different CCs.

(#) analogous to a given graph with n nodes

For ex:- $A = \{1, 3, 5\}$ $B = \{2, 7\}$ $C = \{4, 6\}$

Disjoint (3, 4) returns true
while Disjoint (3, 5) returns false.

Union (3, 4) does $A = A \cup C$

$\Rightarrow A = \{1, 3, 4, 5, 6\}$

Now, let us assume that each set is represented by a colour

say C_i represents colours of S_i

Then given $U = \{0, 1, 2, 3, 4\}$
cardinality of $U = 5$
if we do a Makeset, we get

C_1 [1] C_3 [3]

C_0 [0]

C_2 [2] C_4 [4]

5 ~~different~~ disjoint singleton sets.

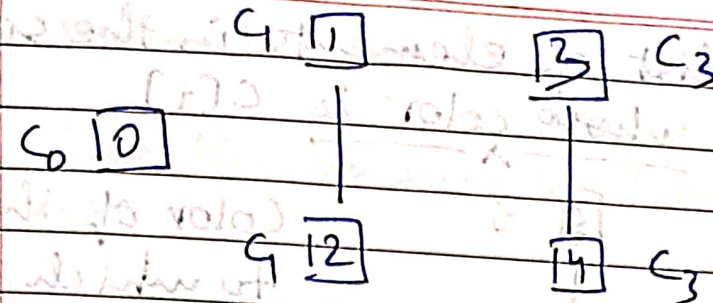
we do Union (1, 2)

C_1 [1] C_3 [3]

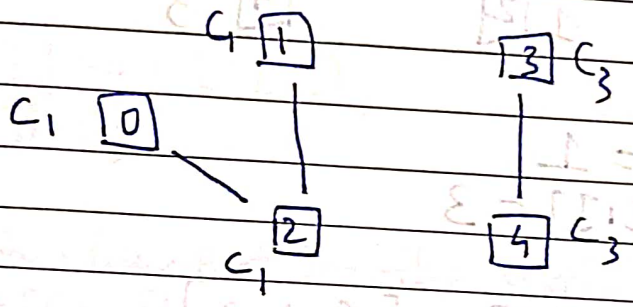
[0]
C_0

C_1 [2] C_4 [4]

Then we do Union (3, 4)



Union (0, 2) gives



We want to make minimum no. of colour changes. To achieve this :-

if we do union(a,b) such that $(a \in S_1 \ \& \ b \in S_2)$ & $S_1 > S_2$

⇒ we'll merge S_2 to S_1 & NOT S_1 to S_2 i.e. we'll change the colour of the elements in S_2 to colour C_1 instead of doing it vice versa.

→

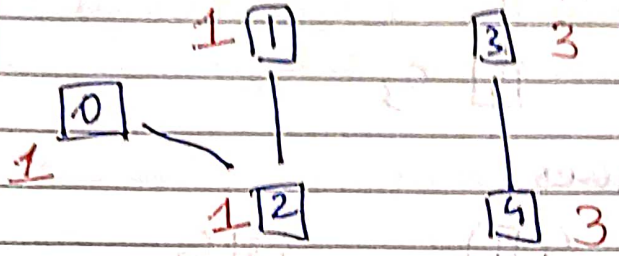
(*) To perform these operations, we'll design a data structure which is called 'Disjoint Union' & to implement this we'll use :-

1.) C[i] - the colour of the element set to which the element of i belong to.

2.) S[C[i]] - no. of elements in the set whose color is C[i].

3.) List [C[i]] - list of elements in the set whose color is C[i]

for ex:-



Color of the set to which that node belongs to is written beside each node

C[1] = 1
S[C[1]] = 3

List [C[1]] = [0, 1, 2]

—x—

→ Code for MakeSet

→ For (i=0; i<n; i++)
{ C[i] = i; List[i] = newnode (List[i], i);
S[i] = 1;
}

—x—

(*) Disjoint (i, j) → check if C[i] == C[j]

Element i & j belong to same set iff C[i] = C[j]

It takes O(1).

—x—

(*) Union(i, j)

Take the smaller set & merge it with the bigger set

```
if (S[C[i]] < S[C[j]])
{
    S[C[j]] += S[C[i]];
    S[C[i]] = 0;
}
```

Delete each element in the list[C[i]] & add it to the list[C[j]] & change the color of each element in C[i] to C[j]

—x—

(*) "Union(i,j)" in worst case can become O(n)

if S[C[i]] = S[C[j]] = n/2;

But this doesn't happen quite often.

(*) We shall now show that n 'union' takes O(nlogn) time.

i.e. Each element can change its color atmost O(logn) times.

To prove this, is equivalent to prove the Thm given below

—*—

Thm :- If an element has changed its color k times, then the size of the set to which it belongs to should be $\geq 2^k$.

i.e. to prove $S[C[i]] \geq 2^k$.

—x—

Proof is by induction.

$k=0 \quad S[C[i]] \geq 2^0 = 1$

Assuming Th^m to be true for k-1 colour changes

$$S[C[i]] \geq 2^{k-1}$$

It will be merged with a bigger set

So after kth color change

$\Rightarrow S[C[i]] \geq 2^{k-1} + 2^{k-1} = 2^k$

$\Rightarrow$ no. of elements $\geq 2^{k-1}$

no. of elements already there

$\Rightarrow S[C[i]] \geq 2^k$

Hence proved

—x—

So, Size of the Universal Set is n

So, any element can change its color atmost $\log n$ times (using above Th^m)

(*) $\Rightarrow$ n 'unions' can take atmost $O(n \log n)$ time

SUMMARY

1. MakeSet – $O(n)$ time.
2. Disjoint (i, j) – $O(1)$ time
3. n Unions $\rightarrow$ $O(n \log n)$ time

(*) Kruskal's Algo makes m calls to Disjoint & $n-1$ calls to Union.

$\Rightarrow$ Complexity of Kruskal's Algo becomes :-

$$O(m \log n + m + n \log n)$$

$$= O((n+m) \log n)$$

Sorting the edges based on their weights

Date ___/___/___
Page ___

(11) Disjoint Union 2

(*) We've already seen in prev. chapter :-

1. MakeSet - O(n) time
2. Disjoint(i, j) - O(1)
3. Union(i, j) - O(log n) on average.

—x—

(*) In this chapter we're gonna see :-

1. MakeSet(O(n))
2. * Disjoint (i, j) →
   a.) O(log n) & then modify it to
   b.) O(1) → AMORTIZED
3. * Union(i, j) - O(1)

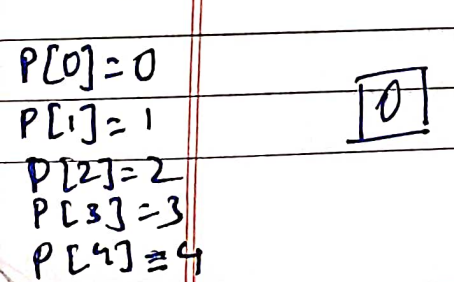
→—

→ To achieve this, we'll introduce a concept of parent & root. (same as the parent & root we learnt in trees).

∴ Each node will be associated with a parent. while " set will be " " " root.
& given a root means given a tree.
i.e. each set is represented by a tree.

—x—

(*) Let us see by taking an example



initially each node is a seperate set (i.e. seperate tree)
⇒ each node is a parent of itself ⇒ P[i] = i

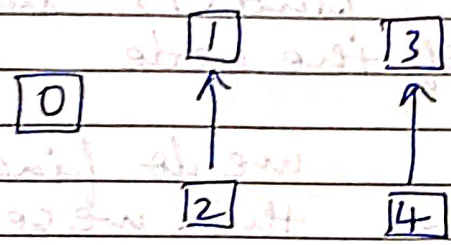
NOTE:- if P[i] = i ⇒ node i is a root node at that instant

* Union(i,j) → means we make P[root(j)] = root(i)
  if (CC of i > CC of j) in terms of no. of elements

We do Union(1,2)

⇒ we set parent of 2 to be 1
⇒ P[2] = 1

similarly union(3,4) ⇒ P[4] = 3

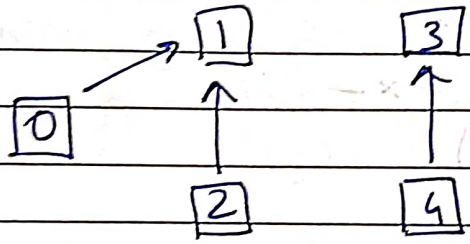


⇒ Here we've 3 CCs, each CC being represented by a root node instead of color (as we saw in prev chapter)

The root nodes in above graph are 0, 1 & 3

—x—

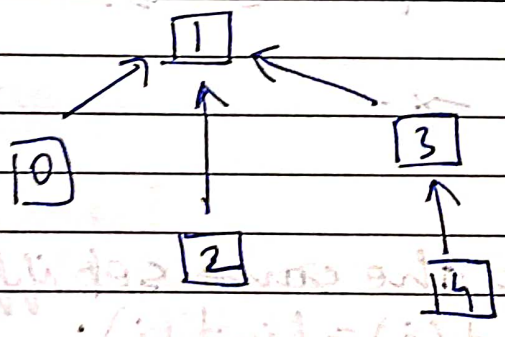
Now, we do union(0,2)



2 root nodes (1 & 3)
⇒ 2 CCs

⇒ P[0] = 1

Again we do union(4,2)



⇒ P[3] = 1

Only one root node (1)
⇒ 1 CC

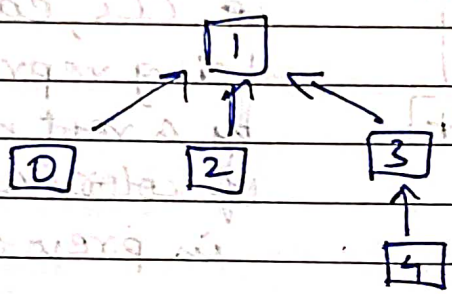
Code for MakeSet

```
for (i=0; i<n; i++)
    P[i]=i;
```

—x—

(*) We've a function 'find(i)' which returns the root of the node

For ex:-



we do find(4) then we see
P[4]=3
then
P[3]=1 then
P[1]=1
⇒ 1 is the root node
Hence we stop & return 1

⇒ find(4) = 1

—x—

Code for find(i)

```
find(i){
    if (P[i]==i) return i;
    else return find(P[i]);
    }
```

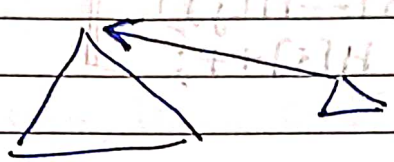
—x—

(*) Disjoint(i,j)

i & j belong to the same set iff find(i) = find(j);

(*) Union(i,j) can be done in 2 ways:-

1.) union by rank:- subtree with fewer no. of elements become the child of a subtree having more no. of elements.



→ Code

→ MakeSet{
for each node
{P[i]=i; S[i]=1;}
}

S[i] stores the no. of elements i.e. size of the subtree rooted at node i

union by rank;
if (S[i] > S[j])
{ P[j]=i; S[i]+=S[j]; }

2.) union by height :- subtree with smaller height becomes the child of the subtree with bigger height

→ Code

→ MakeSet{
for each node
{ H[i]=0; P[i]=i; }
}

H[i] = height of i

VC

Date ___/___/___
Page ___

union by height

```
if (H[i]>H[j])  P[j]=i;

else
{
    P[i]=j;
    if(H[i]==H[j])
        H[i]++;
}
```

→ important to observe

(*) Here the assumption is that we call union only after we call disjoint, & if they are in 2 different sets.

```
a = find(i);
b = find(j);

if (a!=b)
    union(a,b)
```

→ you always merge roots of the subtree.

(*) Each union is O(1)
Each disjoint is 2 find( )

Each 'find' is O(h), h is the height of the tree.

{ We can think of color of a node as the root of the subtree.

→ Write this statement, if you are asked to prove that if an element changes its root k times then size of set ≥ 2^k; & then

exactly copy the proof of the Th^m in the prev. chapter.

Do the above if the union is union by rank.

→ For union by height

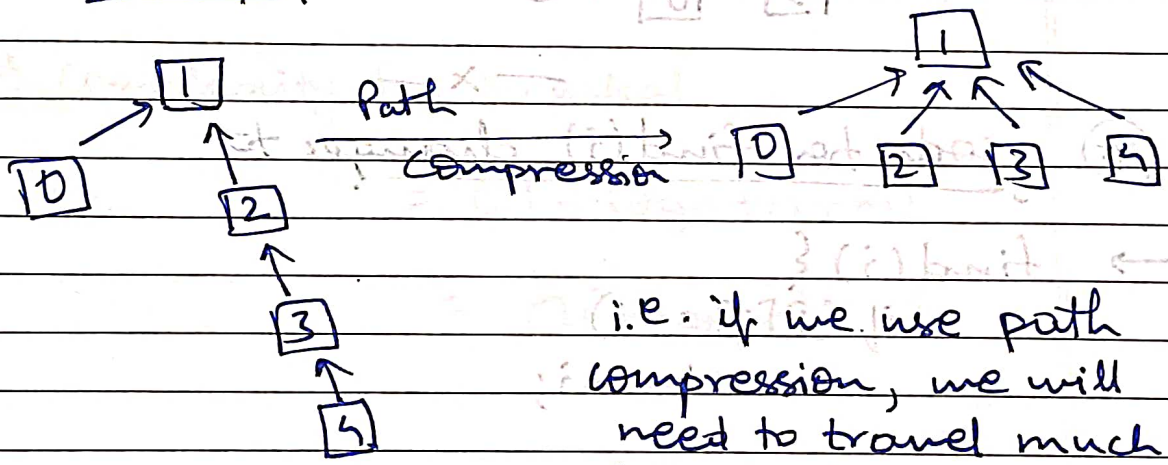
Th^m A tree of height h will have $\geq 2^h$ elements

Proof:- By induction, if $h=0$ then the tree has 1 node.

Height of a tree ↑es only when the height of both the trees to be merged has the same height.

$$\text{no. of elements} \geq 2^{h-1} + 2^{h-1} = 2^h$$

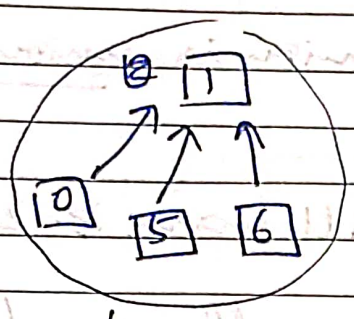
(*) **Path Compression**



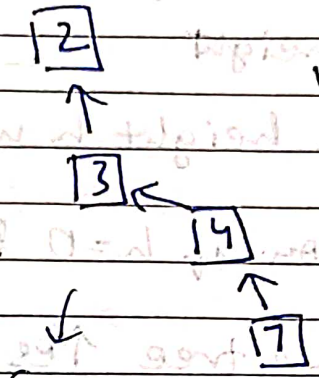
i.e. if we use path compression, we will need to travel much less to infer the root node of the subtree

To use path compression, we'll do union like the following :-

Ex :-



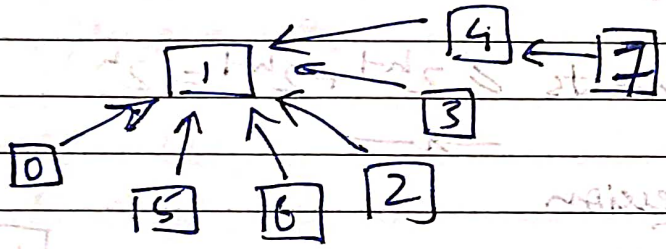
Subtree 1 (ST 1)



ST 2

if we do union (5, 4)

we will connect all the nodes in ST 2 with heights ≥ height of 4 directly to the root node of ST 1 (1)



— x —

(*) code for find (i) changes to

→
```
find(i){
    if (P[i] == i)
        return i;
    else
        return P[i] = find (P[i]);
    }
```

→ using Path compression

(*) Now,

Makeset – O(n) time

Disjoint(i,j) – O(α(n)) time → Amortized *

Union(i,j) – O(1)

where α(n) is the inverse Ackermann fun^n

& α(n) < 4 for n < 10^600.

i.e. α(n) < 4 ∀ practical purposes

i.e. Disjoint(i,j) → O(1) → Amortized

Kruskal makes m calls to Disjoint & n-1 calls to union would take O(n+m) time

⇒ Complexity for Kruskal

= O(mlogn + n + m)

= O(mlogn)

Updating MST

(*) In this chapter we'll see how efficiently we can update the min or max spanning tree if the graph is a dynamic graph.

(*) Let $G(V, E, w)$ & $G'(V, E, w')$ be 2 graphs

i) if $w'(u,v) = w(u,v) + c$; $c \in \mathbb{R}$ $\forall (u,v)$

$\Rightarrow$ MST of G is also MST of G'

This is NOT true for shortest path as we've already seen earlier.

(#) It is b/c if ST is any spanning tree, the no. of edges remain unchanged (i.e. $n-1$)

$$\Rightarrow \sum w'(u,v) = \sum (w(u,v) + c)$$

$$= c(n-1) + \sum w(u,v)$$

As we can see, minimizing or maximizing LHS is the same as minimizing or maximizing RHS

(#) It is ~~so~~ b/c no. of edges in the shortest may ~~or may not~~ change. It is NOT a constant.

ii) $w'(u,v) = w(u,v) \times c$, $\forall (u,v)$

Case a ($c > 0$)

- MST for G is same for G'
- Shortest path for G is same for G' **

Case b ($c < 0$)

~~Min spt.~~ Min ST for G becomes the Max ST for G'

& vice-versa.

—x—

It's so b/c

if ST is any Spann. tree

$$\sum w'(u,v) = \sum c \times w(u,v)$$

$$= c \times \left[\sum w(u,v)\right]$$

i.e. c comes out as a constant from the summation.

(*) Deleting an edge → for min MST → However, can easily be modified for max ST

Let (u,v) be the edge ~~det~~ deleted from the graph

If (u,v) is not an edge of MST, do nothing.

If (u,v) is an edge of the MST, delete the edge from the MST. It will create 2 CCs. Among all the edges going across, pick an edge of min weight & add it to the MST.

—x—

→ Code (using BFS) → If MST is Min ST

BFS →

```
for (i=0; i<n; ++i)
    V[i] = false;

Enqueue (Q, s);
V[s] = true;

while (Q) {
    u = Dequeue (Q);
    For every edge (u,v) in the MST
    if (V[v] == false) {
        Enqueue (Q, v);
        V[v] = true; } }
```

~~BFS(v)~~

Deleting edge → BFS(v);
Min = INT_MAX;

For every edge (u, v) in the graph — O(E)
if (V[v] != V[u] && w(u,v) < min)
{ min = w(u,v); a = u; b = v; }

Add the edge(a, b) to the MST

—x—

(*) To be more precise, it's O(V+E)

However, being a connected graph
E > V
⇒ it becomes O(E).

—x—

(*) Adding an edge → for min MST

Let (u, v) is added to the graph.

Adding (u, v) edge to the MST, it will create a cycle. Delete the max weighted edge in the cycle.

Date / /
Page

→ Code for ~~deleting~~ adding an edge (using BFS)

BFS →

```
for (i=0; i<n; ++i){
    V[i] = false;   M[i] = INT_MIN; }

Enqueue(Q,s);
V[s] = true;

// M[i] = max weighted edge reachable from s;

while(Q){
    u = Dequeue(Q);
    for every edge (u,v) in the MST
    if (V[v] == false){
        Enqueue(Q,v);
        V[v] = true;}

    if (M[u] <= w(u,v))
    { M[v] = w(u,v);   A[v] = v;   B[v] = u; }
    else
    {M[v] = M[u];   A[v] = A[u];   B[v] = B[u]; }
}
```

Add edge → Add the edge (u,v) to MST
BFS(u);

Delete the edge (A[v], B[v]) from the MST | O(V)

Note that w(A[v], B[b]) = M[v];

→

takes less time ⇒ i.e. it is easy to update MST.
However, we don't have such easy ways of update shortest path or BFS or DFS.

(*) Updating edge

update (u, v, w) → set w(u,v) = w

If Increase key i.e. w > w(u,v)
⇒ it is similar to Deleting an edge ⇒ O(E)

If Decrease key i.e. w < w(u,v)
⇒ it is similar to Adding an edge ⇒ O(V)

(*) Adding a node

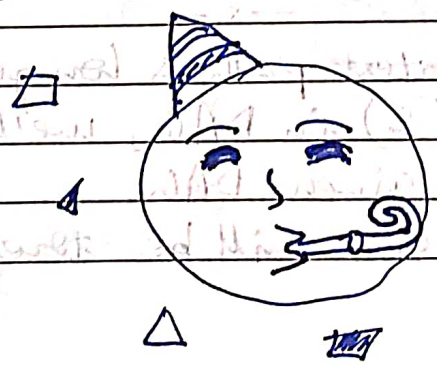
G(V, E, w) be the graph & MST be its Min ST.

A new node & k weighted edges incident on this node are added to the graph.

We've 2 ways of doing this.

i) Add k edges using the previous algo
⇒ O(kn) time

ii) Add k edges to the MST to compute a Graph G' & apply Kruskal's Algo to find MST. This takes O(n log n) time.



COURSE ENDS

Shortest & Longest paths in a DAG:

(*) We saw already, that for a general directed as well as undirected graph, shortest path problem is solvable in polynomial time, but longest path problem is NP-hard.

→ However, for DAG both shortest and longest path problems are solvable in ~~polynomial~~ linear time. When we say linear time it's linear ~~in~~ in terms of the no. of vertices & no. of edges in the graph using DFS.

In this ch we'll look at how to find shortest & longest from a source node to every node in a graph in linear time.

Just to recap,
Apply DFS on G, then list the nodes in the ling order of finish times. This is called the topological sort.

We also know that in this list, there will be only forward edges & there will be no backward edges in the graph if the given G is a DAG.

—x—

To find the shortest path & longest path from a source node (s) in DAG, we'll need a topological sort of the given DAG.
The topological sort will be stored in array T[].

The shortest path for each node will be stored in S[]
" longest " " " " " " " " " L[]
For shortest path tree → Phi1[]
" longest " " " → Phi2[]

→ After computing T[], we'll walk through each node of T[] and then do edge relaxation for all its corresponding edges

for shortest path

```
if (S[v] > S[u] + w(u,v))
{S[v] = S[u] + w(u,v);}
```

for longest path

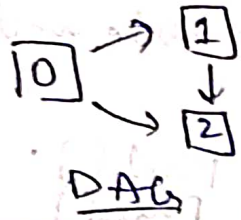
```
if (L[v] < L[u] + w(u,v))
{L[v] = L[u] + w(u,v);}
```

CODE after computing T[]

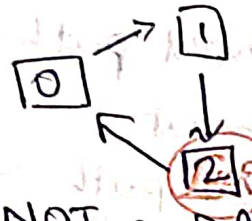
```
Path(s){
  for(i=0; i<n; ++i)
  { L[i] = Min; S[i] = Max; Phi1[i] = Phi2[i] = -2;}
      ↳ -∞          ↳ +∞
  L[s] = S[s] = 0;
  Phi1[s] = Phi2[s] = -1;

  for(i=0; i<n; ++i) {
      u = T[i];
      for every edge (u,v)
          if (S[v] > S[u] + w(u,v))
              { S[v] = S[u] + w(u,v); Phi1[v] = u;}
          if (L[v] < L[u] + w(u,v))
          {L[v] = L[u] + w(u,v); Phi2[v] = u;}
  }}
```

O(m)

Mind it



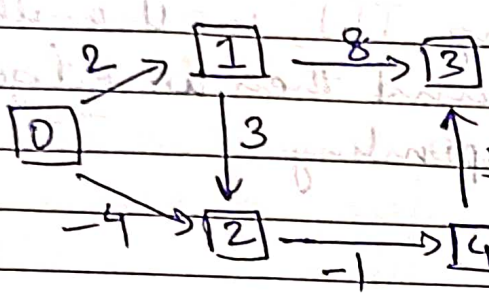
DAG



NOT a DAG

Date / /
Page

For ex:-



Do a dry run if you wish.

→ Gives DAG with weight written for each edge

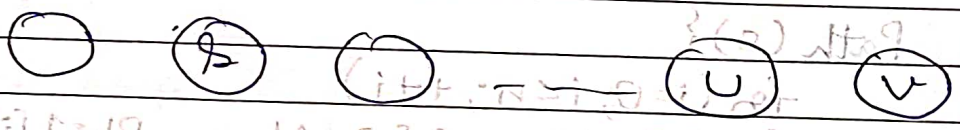
$T = \{0, 1, 2, 4, 3\}$

$S = \{0, 2, -4, 0, -5\}$; $L = \{0, 2, 5, 10, 4\}$

(*) Correctness of Algo for shortest path

Proof is by induction

Following are the nodes in order in T[]



∵ there are no backward edges.

⇒ shortest path of nodes prior to s is ∞ as they ~~are~~ are NOT reachable from s.

S[s] = 0 trivial

Assuming S[v] = shortest path of v from s ~~if at all~~ ∀ v b/w s & u.

⇒ before looking at the outgoing edges of u, we'd have considered all the incoming edges to u.

& if at all, at any time we'd find an incoming edge such that

$$S[v] > S[u] + w(u,v)$$

we'd update $S[v] = S[u] + w(u,v)$

Hence $S[v]$ finally contains the shortest path from s to v

Similarly, we can do a proof for longest path.

—x—

(*) Complexity

Complexity for finding shortest & longest path in DAG = Topological Sort + Path(s)

$= O(m+n) + O(m)$

$= O(m+n)$

NOTE:- Algorithm works even for the -ve edges