AOA

RAYYAN SHABBIR

BITF20M535

SEM #05

lecture # 07

AoA

GRAPHS

* Pictorial representation
* Collection of vertices / nodes & edges.

G (V, E)

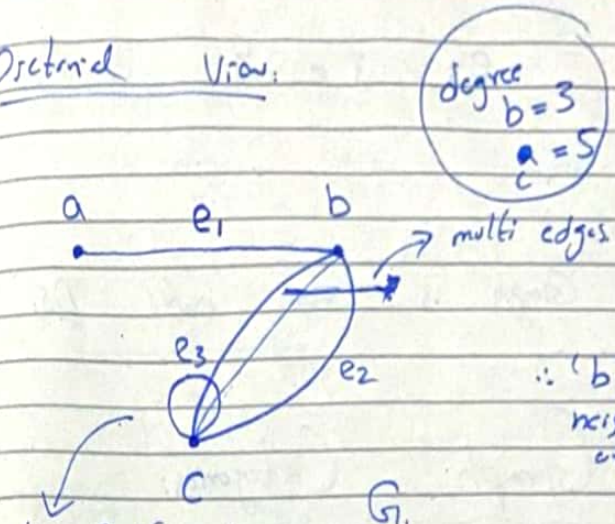
- Nonempty set of vertices
- set of edges

* Graph by Definition: (machine see it like this)

V = { a, b, c }

E = { {a,b}, {b,c}, {c,c} }

* Pictorial View:



∴ 'b' is neighbour of 'a'.

∴ vertix c is accessible from 'a' & 'b'.

path: a - b - c

* graph can never be empty.
* May graph cannot have any edges.

To represent edge: → by line

↳ connection b/w two vertices.

→ bcz it has two vertices So, we write it as:

$e_1 = \{a, b\}$

↳ end-points of edge.

$e_2 = \{c, b\}$

$e_3 = \{c, c\}$

★ Graph is very useful DS.

Graph Categories:

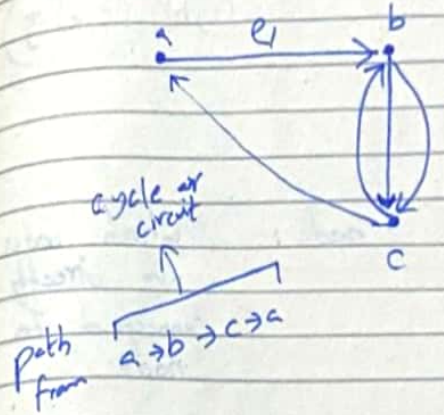
(i) Undirected Graph:

↳ edges have no direction.

(ii) Directed Graph:

↳ edges have direction.

∴ 'a' is not neighbour of 'b'.



∴ 'b' is neighbour of 'a'

G_2

★ As, it has edges which have direction so, we write it in order pair form.

'a' → start / initial vertex

'b' → final / terminal vertex.

order pair.

$E = \{(a,b), (b,c), (c,b), (c,a)\}$

$V = \{a, b, c\}$

Previous

Graph by definition

$G_2 (\{a, b, c\}, \{(a,b), (b,c), (c,b), (c,c)\})$

neighbour of a node: which nodes are directly connected to node.

or

adjacent vertices.

Path: → if one length it is edge.

→ if multiple edges it will be path.

cycle: or circuit — is a path where start and end vertices are same.

loop

* Graph can have many cycles.

* Simple path

$a \to b \to c$

* Not Simple Path:

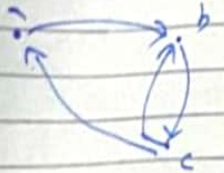
$a \to b \to c \to b \to c$ ← ∴ same edges are repeated here

Categories of Directed or Undirected - graph.

1) ⇒ **Simple - Graph:**

↳ Graph which have no loop & no multi-edge

↳ cycle can exists.



← simple graph

2) ⇒ **Multi - Graph:**

↳ no loops

↳ but no restriction of multi-edges.

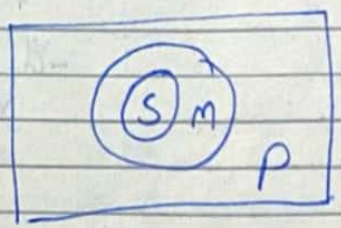
* Eg simple graph is also a multi-graph.

* But eg multi-graph may or may not simple graph.

3) ⇒ **Pseudo Graph**

↳ everything allows

* multi-graph can be a pseudo-graph



So,	loop	multi-edge
1- Simple	X	X
2- Multi	X	✱
3- Pseudo	✱	✱

Maybe

multi-edges : more than one connection b/w two vertices.

* In directed graph there can also two outgoing edges.

Undirected

degree of vertix: how may edges associated with a vertix.

* In cycle we consider it as '2' in degree.

Directed

In-degree:

↳ how my incoming edges

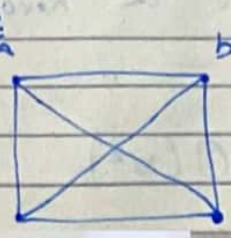
Out-degree:

↳ how my outgoing edges

Types of

→ Complete Graph in Simple Graph

each vertix of graph is connected with all other vertices of graph is called complete graph.



K_4

↳ vertices or nodes = 4

Total no. of edges:

if vertices = n

for vertice → 1 = n-1

= n-2

= n-3

⋮

= 3

= 2

= 1

= 0

Total edges = (n-1) + (n-2) + ⋯ + 2 + 1

$= \frac{n(n-1)}{2}$

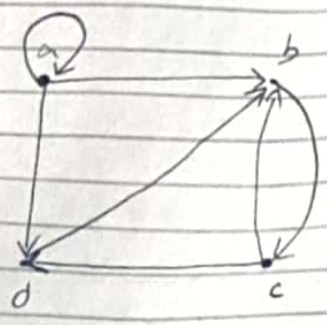
$= O(n^2)$

Max no. of edges that simple graph can have

So, $|E| = O(V^2)$

Sub-graph: if we remove some edges from a graph it will be sub-graph of orignal graph.

* Graph is also its sub-graph.



G_2.

Matrix:

G_2:

	a	b	c	d
a	1	1	0	1
b	0	0	1	0
c	0	1	0	1
d	0	1	0	0

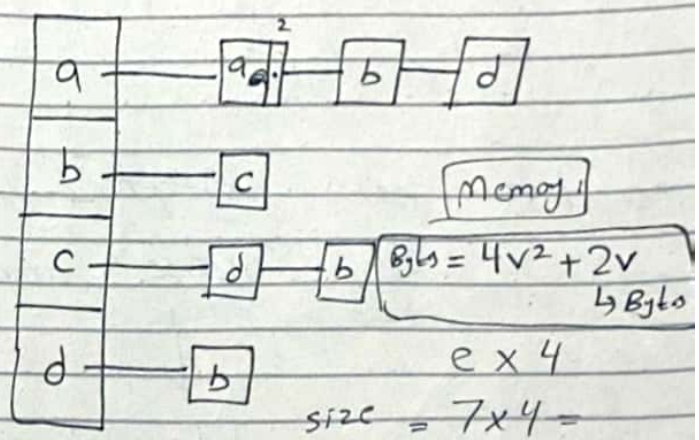
* In directed graph

1's = edges #

① Adjacency List Representation

* link list of adjacent vertices of a vertex.

G_1:



G_2:

Memory:

Bytes = $4v^2 + 2v$

4 Bytes

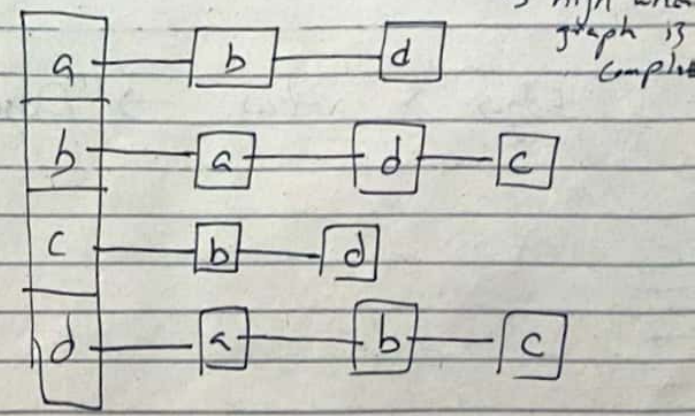
$e \times 4$

size = 7×4 =

* here in directed graph

* So, In Adjacency list memory is high when graph is complete

G_1:



* Space/size of matrix does not effected by directed or undirected.

↳ only no. of 1's are effected by no. of edges.

* less nodes graphs is called "sparse graph".

↓

Edges < vertices

Edges > vertices → "Dose graph".

* Thus, when graph is dense

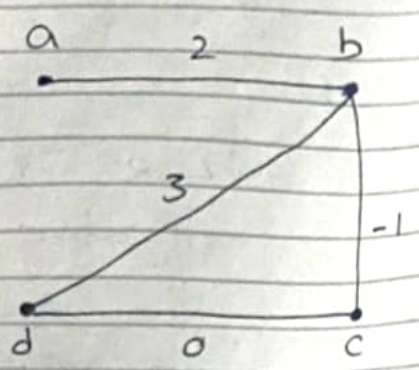
↳ A.L.R is expensive

&

A.M.R is not expensive

In ~~sparse~~ sparse

↳ A.L.R is because less memory but A.M.R does not get any effected.



Weighted undirected simple graph

↳ Every ~~good~~ edge has same weight.

↳ we represent it by
A · $\overset{m}{B}$ · R (not boolean)

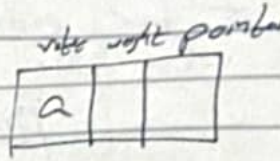
↳ place '∞' if not adjacent (not edge) b/w two vertices.

↳ place edge in matrix.

↳ In A·L·R

↳ each vertice

↳ has size



How to traverse graph

Traversal : go every vertice of graph.

if Matrix : run '2' loops → 2D array

$O(V^2)$

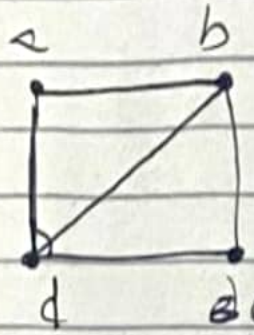
if list : 2 loops → 1: vertice
2: list of vertices.

algo next page

Adjacency

Matrix Representation

if we have graph.



G_1

Matrix Base Representation (For Simple Graph)

'1' means edge b/w two vertices

$|V| \times |V| = \{u, v\}$

0 means not connected (not edge)

G_1:	a	b	c	d
a	0	1	0	1
b	1	0	1	1
c	0	1	0	1
d	1	1	1	0

Memory

Bits = $|V| \times |V| \times 1$ =

$$\text{Bits} = \frac{V^2}{\text{bytes} \in 8}$$

In undirected graph:

* In this graph edges $\Rightarrow 2$

↳ $2 * 1$'s $\in$ edges.

↳ each edges repeat '2' time in matrix.

In List (Traversing)

graph

1) print_Edges (G)

vertex

1- for each vertex $u \in G.V$ (set of vertices which does not have any order)

2- list ← for each vertex $v \in Adj[u]$

3- print "(u,v)"

Run Time :

edges of vertex

$O(V \times (V-1))$

$O(V^2)$

In Matrix (Traversing)

matrix

print_Edges (G)

1- for i=1 to |G.V|

2- for j=1 to |G.V|

3- if G(i,j)=1 then

print "(i,j)"

Run Time = $O(V^2)$

Code of BFS

```
                                    start vertex
                                        ↓
BFS(G, s)
1-   for each vertex u ∈ G.V                    } O(V)
2-       u.color = white
3-       u.π = Nil
4-       u.d = ∞
5-   s.color = gray
6-   s.d = 0                          before
                                        ↓
7-   Q = {s}  or  Enqueue(Q, s)   ▷ Q is a queue
8-   while Q ≠ empty                  } O(V+E)
9-       u = Dequeue(Q)
10-      for each vertex v ∈ Adj[u]
11-          if v.color = white then  } constant
12-              v.color = gray
13-              v.π = u
14-              v.d = u.d + 1
15-              Enqueue(Q, v)
16-      u.color = black
```

KnapSack (v, W, C)

1- dp[0 ... C] = 0
2- n = len(v)

3- for i=1 to n
4- for w=i to C
5- if $W_i \leq w$
6- dp[w] = max{dp[w], v_i + dp[w-W_i]}

[illegible]

a → b → c

Lect # 10

AOA

Graph

DFS:

↳ Traverse 1 adjacat completly until we find a adjcat whoose adjacents are fully traversed.

* keep info of parat (it gives most of info).
* We take any vatex to start.
  ↳ and tha check its adjacab vertices.
* no part of graph remains which is undiscovered.

Algo

Running time

DFS (G) → O(V+E)

```
1-  for each vertex u ∈ G.V
2-      u.color = white
3-      u.π = Nil

4-  time = 0   ← global variable.

5-  for each vertex u ∈ G.V
6-      if u.color = white then
7-          DFS_Visit(u)
```

O(V)

O(V+E)

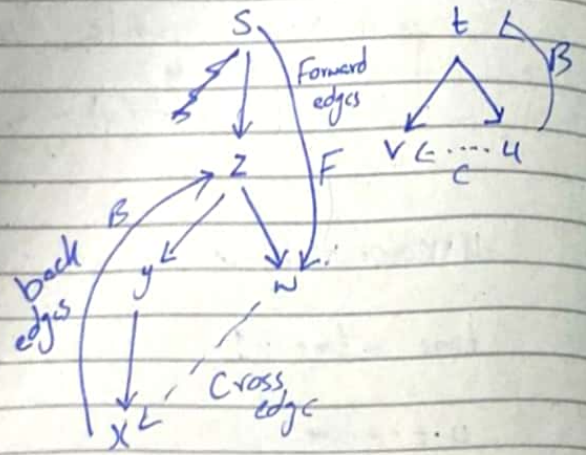
DFS_Visit (u) → node to visit

```
1-  time = time + 1   → discover time
    u.d = time

2-  u.color = gray

3-  for each vertex v ∈ Adj[u]
4-      if v.color = white then
5-          v.π = u
6-          DFS_Visit (v)

7-  u.color = black

8-  time = time + 1

9-  u.f = time
```

O(E)

⇒ discovery edges & finish by parent.

DFS Forest:



Tree edge: Direct edge

Forward edge: edge from ancestor to descendant.

back edge: edge from descendant to ancestor.

↳ it makes cycle

ancestor & descendant relationship only maybe siblings.

⇒ if we want to see cycle in the after running after DFS then DFS.

we check the DFS → colors scheme or { discovery time / finishing time }

Time

(1) If the time slots of '2' vertices are overlapped then those two nodes have Back edges b/w them.

if we have nodes: (u, v)

u.d < v.d < v.f < u.f

For
* Cycle_Detection we use

Disjoint set Data Structure.

↳ These are set DS.

means all set are disjoint of each other (No overlapping b/w these)

A = {a, b}
B = {c, d, e}
F = {d}
D = {f}

Algo in These D.S

Make_Set (u) → constant time
↳ Make set of a vertex.

Find_Set (u) → (log V)
↳ return name of set which contain this vertex

→ Union (u, v) → (log(V))
↳ make union of u & v elements.

Union (a, f)

E = {a, b, d, f}

* we first do make_Set (u)
↳ of all vertices.

↳ Place two connected vertices in same set.

if not same set

Then we union it.

{a} {b} {c} {d} {e} {f} {g} {h} {i}

{a b} {c, i, d, h, g, f}

* we check cycle . by find set. Here.

$O(E(E+V))$

↳ in place of this we call find & union

So, it runs in

$O(E \lg V)$

edge $(u,v) \in G.E$

u v

nlogn

Critical edge: which we cannot remove bez otherwise our graph will be isolated.

Algo steps Strategy #1

O(ElgE) or O(ElgV)

* Sorted edges in decreasing order

* if components ↑ at a vertex (which has large weight) is '1' then
  - we remove it.
  - if component is greater than 1 i.e 2 it (we cannot remove it) & go for 2nd largest weight edge.

O(E(E+V))

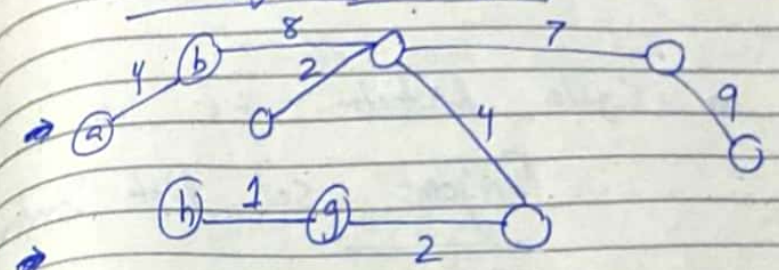
& remove accordingly.

DFS property (must read)

① Connected component

②

Strategy #2 Algo



* Sort the edges in increasing order.

→ Connected the edge which has minimum weight.

Then choose the next minimum.

* Then we run cycle detection algorithm.
  - if cycle exist then remove that edge.
  - We use DFS → back edge

it must remain.

it also run in O(E(E+V))

cycle detection ext. cost.

Spanning Tree: is a subgraph which have all the vertices of a graph & all these vertices are connected also.

* Spanning Tree which has least weight is called minimum spanning Tree.

H_1 is the "minimum spanning Tree."

* There can be more than '1' MST.

.

.

⇒ How To MST for a graph?

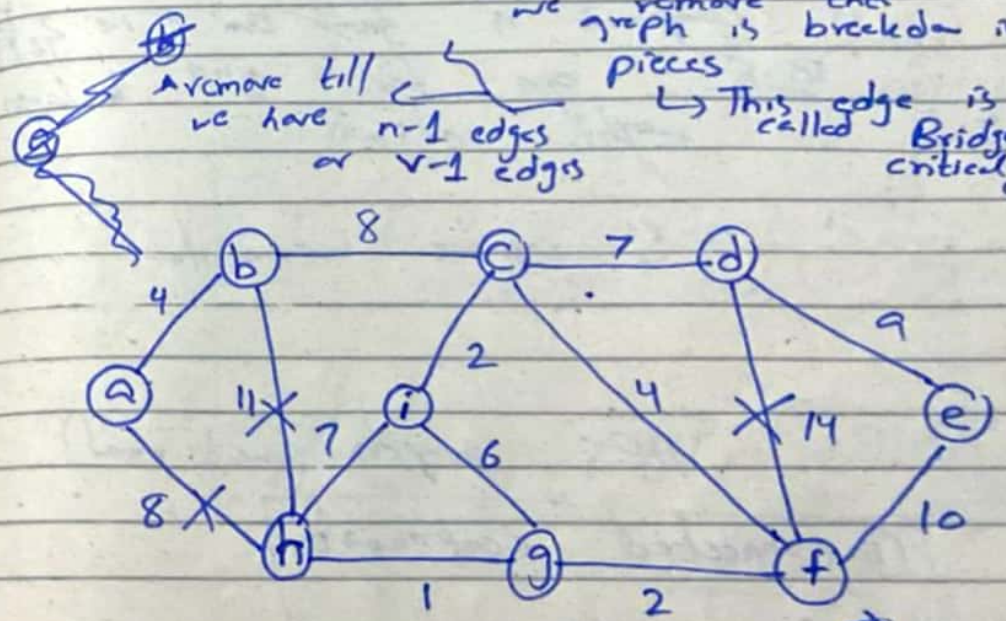
Minimum Spanning Tree [MST]

* if a Tree has n nodes then it has n-1 edges.

* if 'q' vertices / nodes the we have 8 edges then it would be a Tree.

Which edge to remove?
→ remove largest weight edges
↳ but these largest should not be that one which if we remove the graph is broken into pieces
↳ This edge is called Bridge or critical edge.

remove till we have n-1 edges or v-1 edges



1 2 5 6 — 1 2 3 4 — 1 2 1 2

u.s < u.f && v.s < v.f
&& u.f < v.s

(2) if the discovery time of 1st node is less than discovery time of other (2nd) node && if the finishing time of 1st node is less than finishing time of other (2nd) node then those nodes have a forward edge.

if two nodes : (u, v)

v.d < u.d < v.f < u.f

means 1st node starts earlier than 2nd node and also finished earlier than 2nd node.

(3) if the time slots of '2' nodes does not overlap (means 1 time slot does not fall in other) then it means those nodes have a cross edge.

Lecture # 11

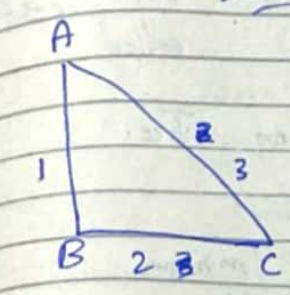
4: 7, 8
v 10, 12

u.s < v.s
u.f < v.f
v.s > u.f

AOA

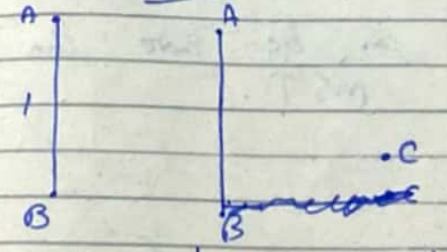
Weighted UNDIRECTED GRAPH

Minimum Spanning Tree

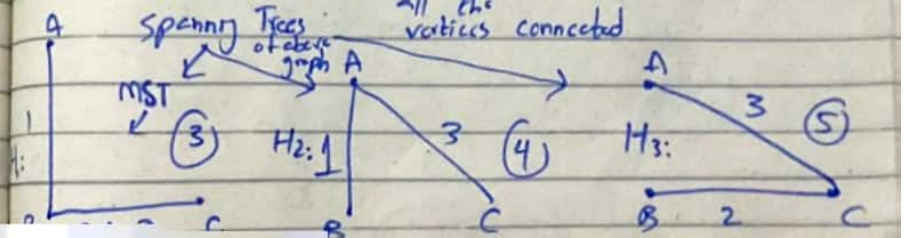


Graph → also a subgraph

sub graphs



Tree — not a Tree bez it don't have all the vertices connected — Tree



Perot:

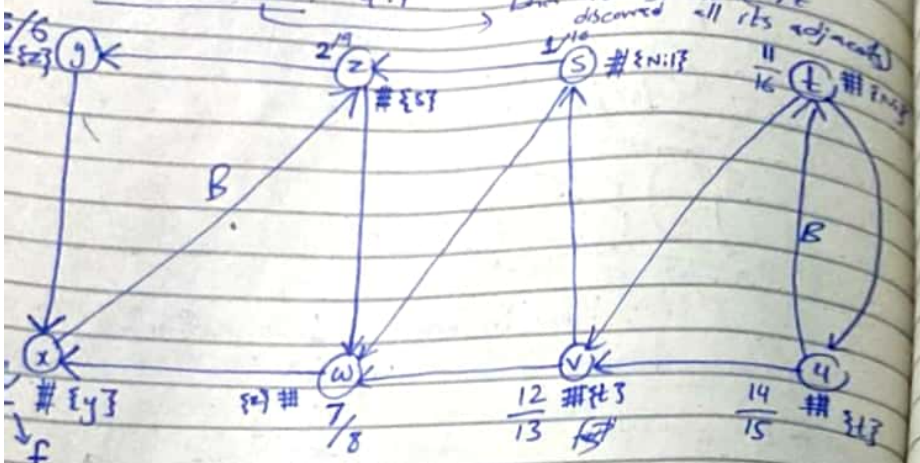
* $4 \cdot \pi$

color scheme

4 colors

full_traversed → black

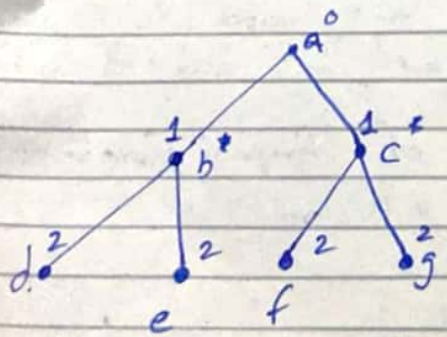
* discovery time: 4.d
* finish time: 4.f → when it black (it discovered all its adjacents)



use a must capture first

distance of vertex from root node

Ex:- 13.2.7



distance.

Q:- 13.2.7

Good people, Bad people

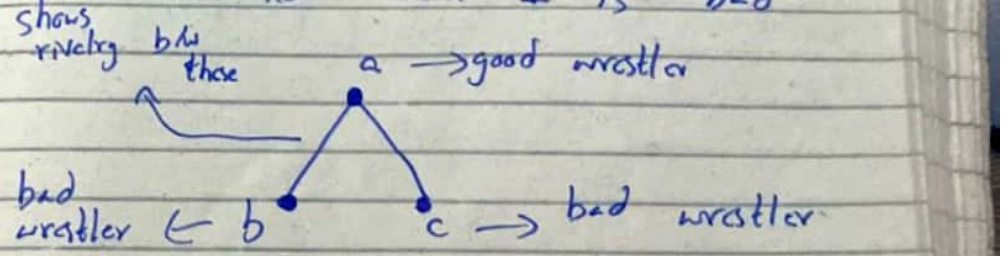
n wrestlers.

edge shows rivalries

wrestlers → nodes / vertices.

⇒ Algo which diff good & bad wrestler if not possible why?

1 connected is ± good and other 1 is bad

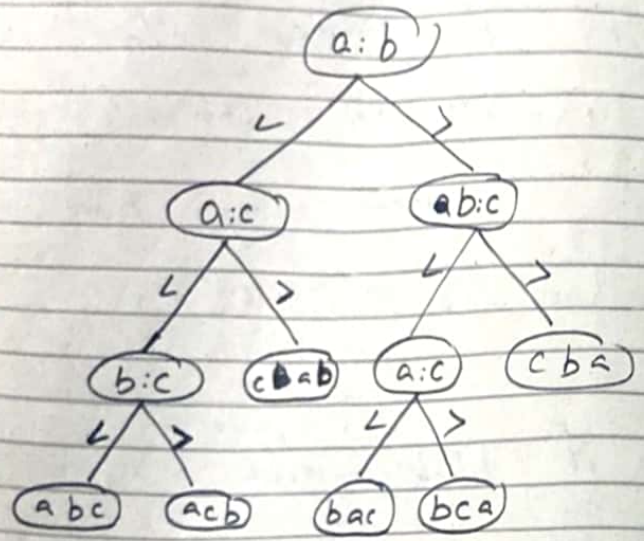


a < b

a > c

b ~ a c

Rough



b < c

c < b

a < b

c < a

a < c

b < c

b < a

b < c

a < c

Quiz

Knapsack 0-1

duplicatin

Algo

KnapSack (v, W, c)

1- n = len(v)

2- dp[0] = 0

3- for w=1 to W

4- dp[w] = dp[w-1]

5- for i = 1 to n

6- if $w_i \leq w$

7- dp[w] = max{ dp[w], v_i + dp[w-w_i]

8- return dp[W]

2nd method next page

Time Analysis of previous code =

$O(V+E)$

Linear ← BFS runs in this

* To enter in a Queue elements we make it color grey.

* Q has all elements whose color is grey.

↳ elements which go at of queue are white or black.

if Adjacents:

grey to grey edge means they are siblings → there parent or grand parent are same.

* if two vertices have grey color then it means they have a cycle.

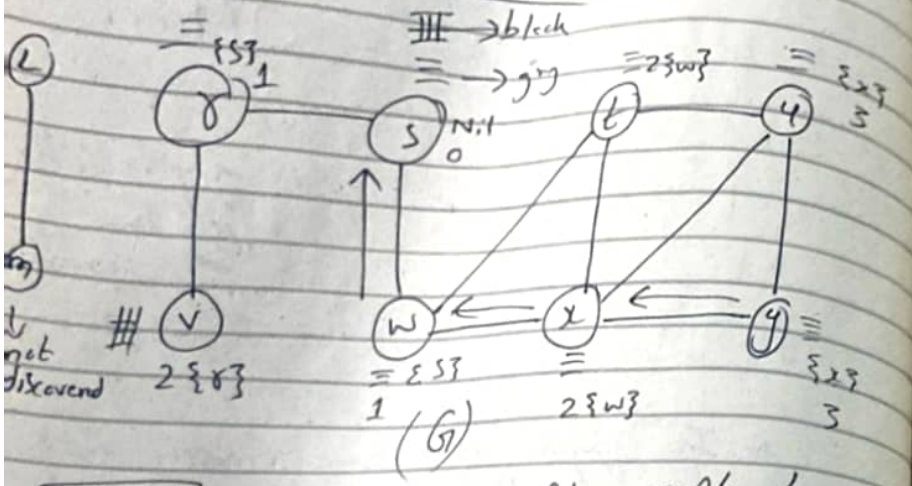
* if black adjacent ↳ then no cycle means grey to black.

↳ then what?

Lect #09

TRAVERSING of graph ACA

→ Two techniques of traversing
(1) BFS
(2) DFS

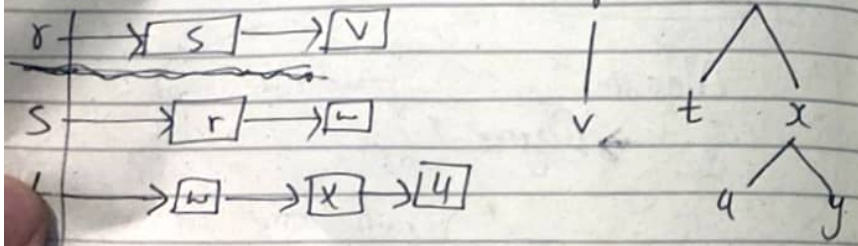


* Above graph has 2 components.

Q : s, r, u, v, w, x, t, y

u = s

BFS predecessor or BFS Tree sub-graph



① Breadth First Search

↳ neighbour of any vertex will be visit (Repeat)

↳ First traverse → length 1
then length → 2 then so on

length of u traversed then u+1

⇒ start from 's' → then choose shortest path. (s → x)

* Graph have 3 types of nodes.

↳ Discovered — u.color — grey

↳ not Discovered — white

↳ ~~Not~~ traversed / Discovered and also — black
complete, its adjacent discovered.

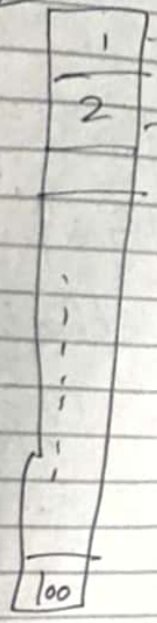
saving parent : u.π → contains parent of vertice (who discovered it)

saving distance : u.d → contains the shortest distance from src vertice

	Space	Speed
List	Bad	Good
Matrix	Good	Bad

if we have graph,

Algo#1



vertices = 100

edges = 0

So, Algo#1

O(V)

← if complete graph

100 x 99

V x (V-1)

↳ So, we say it runs for 'edges' time.

As, we not know how much vertices & edges ←

So we write

↳ O(V+E) ← ↳ its running time.

if assume E > V → Dense graph

O(E)

if E < V sparse graph

O(V)

So, Algo#1 runs in

linear → O(V+E)

↳ in graph the is how is linear.

☆ In graph we assume mostly there is list representation

↳ if given in question then only it will be matrix.

otherwise it will always be list.

Hamc Walk

Algo#

undirected graph

→ Degree (G)

↳ prints degree of each vertices