Graphs

1- Representation of graphs:

- Two ways to represent graphs: adjency lists, adjency matrix → yenfa3o (directed w undirected graphs
- adjency list compact for <u>sparse graphs</u> ($|E| <<< |V|^2$)
  ↳ method of choice

- adjency matrix prefered when <u>dense graphs</u>
  ($|E|$ is close to $|V|^2$)
  → or we need to tell if there is an edge connecting two vertices quickly

$\boxed{1}$ The ~~adjency~~ adjacency list algorithm

→ Each vertex stores a list of vertices adjacent to it resulting in an array of linked lists

→ neighbors can appear in any order.

→ each edge appears twice

→ n ⇒ number of vertices
  e ⇒ number of edges

- undirected graph → space usage for adj. list ⇒ $n + e * 2 * 2 = n + 4e$

  2 nodes per edge

  2 units per node (number + next)

  while adjacency matrix → n^2

→ If the graph is weighted, the weight must be stored

2 The adjacency matrix:

→ undirected graph: adjacency matrix A

$A = A^T$ → symmetric.

→ Simpler, prefered when graphs are small

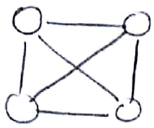
→ further advantage on unweighted → one bit entry.

Adjjency List	Adjacency Matrix
Space: $\Theta(V+E)$	$\Theta(V^2)$
• can put weights in the lists	• $\lvert V\rvert \times \lvert V\rvert$ matrix $A = a_{ij}$ • can store weights instead of bits for weighted graph $a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$
Time: • To list all vertices adjacent to u. $\Theta(\text{degree}(u))$	• $\Theta(V)$
• To determine if $(u,v) \in E$: $O(\text{degree}(u))$	• $\Theta(1)$

Scanned by CamScanner

→ Total # of edges in Undirected graph

$= \frac{n(n-1)}{2}$ → max number of vertices



→ directed graph

$= n(n-1)$ edges

0 edges ⟷ $\frac{n(n-1)}{2}$ → undirected

n vertices $n(n-1)$ → directed

← Sparse graph dense graphs → space utilization in adjacency matrix ↑↑

- Comparison for space usage:

Units of space to store graphs with n vertices, e edges (not counting space to store vertex names)

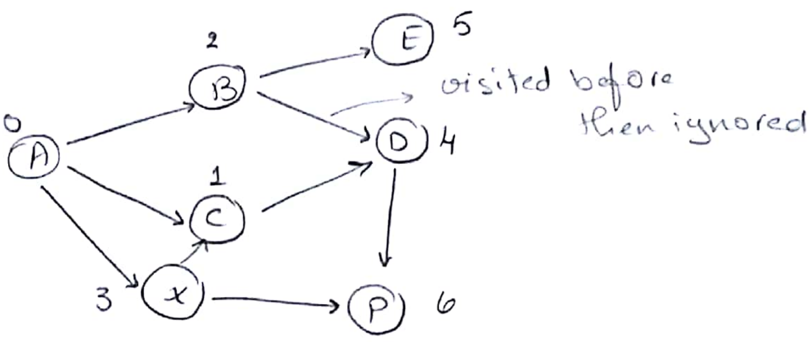
Graph	adjacency linked lists → $\Theta(V+E)$	adjacency matrix → $\Theta(V^2)$
① Undirected no weights	$n+4e$	n^2
② Undirected, weighted	$n+6e$	n^2
③ Directed, no weights	$n+2e$	n^2
④ Directed, weighted	$n+3e$	n^2
	each linked list → 2 units unweighted ↳ 3 units weighted	each array cell is one unit space

→ Mesh sahl fel adjacency - list eny at2aked eno edge mawgood. El asras eny adwar 3ala vertex Bas fel adjacency matrix ashal bas memory az3yad

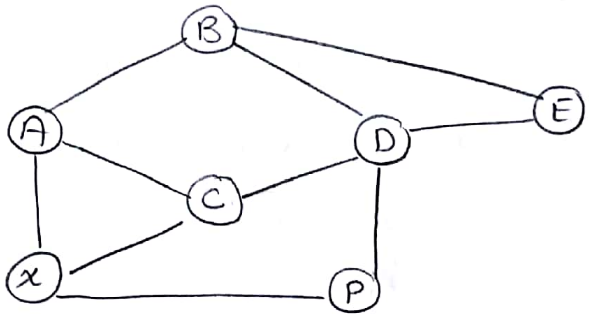
---

Breadth-first search:

→ It doesn't matter which vertex to start with.



A: B, C, x
B: A, D, E
C: A, D, x
D: B, C, E, P
E: B, D
P: D, x
x: A, C, P



start at C

Queue

~~C~~ ~~A~~ ~~D~~ ~~x~~ ~~B~~ ~~E~~ ~~P~~ till Queue is empty

C x

C, A, D, x, B, E, P, skipped because already visited.

→ If graph composed from 2 islands → restart BFS

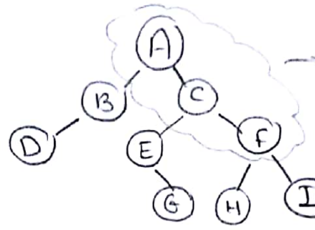
→ Time: n+e → directed   n+2e → undirected

$\therefore O(n+e)$

→ If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black

→ Gray vertices may have some adjacent white vertices, they represent the frontier between discovered and undiscovered vertices.

N.B A node that is connected to all lower-level nodes is called "ancestor". The connected low-level nodes are "des-cendants" of ancestor node



every node is both ancestor and des-cendant of itself

→ Notes on code:

- Parent (predecessor) of u is stored in $u.\pi$
- If $u = s$ or not visited yet $u.\pi =$ NIL
- $u.d$ → holds distance from source s to vertex u
- At line 10, Q contains gray vertices.
- Complexity:

Time Queue → enqueue / Dequeue $O(1)$ ∴ total time $O(V)$

Scan of adjacency list for each vertex (only on dequeued) → $O(E)$

∴ $O(V+E)$

Space same size of adjacency list

→ P598

Applications checking connectivity, checking acyclicity, shortest path

→ Print shortest path from s to v after BFS has already computed breadth-first tree:

```
PrintPath (G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "no path from" s "to" v "exists"
else PrintPath (G, s, v.π)
    print v
```

Depth-First Search:

→ As BFS, if we start from a vertex and all vertices are not all visited, then restart DFS from another vertex

→ directed graph → $n+e$.
undirected graph → $n+2e$
$\therefore O(n+e)$

→ Unlike BFS, whose predecessor subgraph forms a tree, the predecessor subgraph produced by DFS may be composed of several trees because the search may repeat from several sources.

→ $v.d$ → records when v is first discovered (and grayed)
$v.f$ → records when the search finishes examining v's adjacency list (and blacken v)

$u.d < u.f$ $\quad\quad$ $1 \le u.d, u.f \le 2|V|$
one discovery, one finish

Scanned by CamScanner

→ white before u.d , gray between u.d and u.f , black thereafter.

→ Timestamps are important to give info about the structure of the graph and the behavior of DFS.

→ u = v.π if and only if DFS-Visit(G, v) was called during a search of u's adjacency list.

→ v is a descendant of vertex u in DF-forest if and only if v is discovered during the time in which u is gray.

→ Property: discovery and finishing times have parenthesis structure

→ Directed graph is acyclic if and only if a depth-first search yields no "black" edges.

→ Clasification of edges: (edges from (u, v)

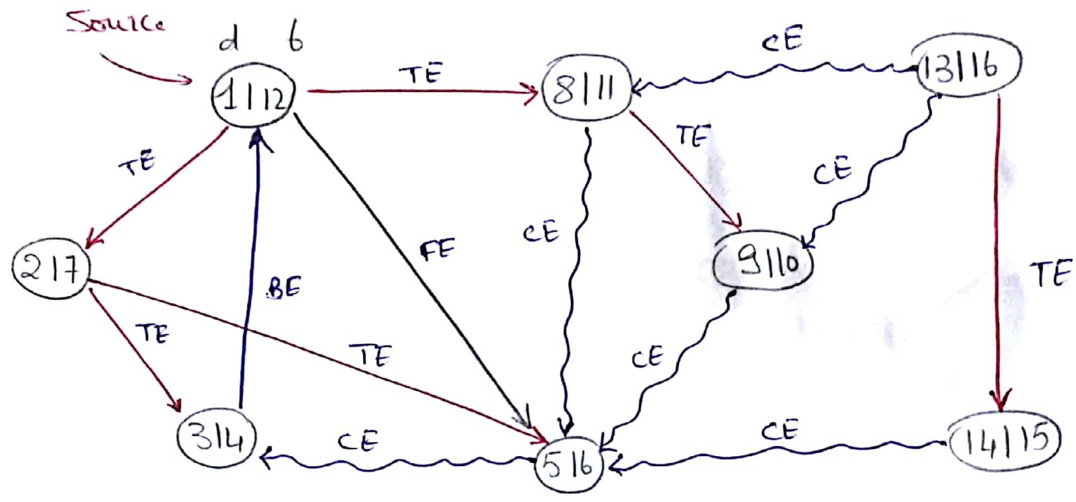
1- Tree edges: encounter new (white vertex) ↳ v

2- Back edge: From descendant to ancestor in DFT (self loops in directed graph) (gray vertex) ↳ v

3- Forward edges: from ancestor to descendent in DFT (Black) ↳ v

4- Cross Edges: between nodes in a tree or Subtrees.
- From grey to black
- nodes not ancestors of each other

(Black) ↳ v



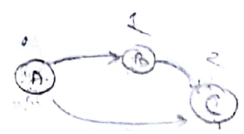
Applications: 1- Find connected components
2- checking ayclicity

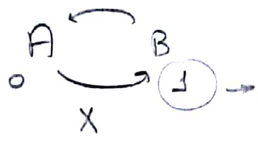
N.B . Forward and cross edges never occur in DFS of undirected graph

---

Topological Sort:

→ Valid for directed graphs without cyles or acyclic graphs (DAGs)





→ topological number

ABC

→ Assign the TN when you are about to backup from a vertex (Applying dFS
a5er vertex ta5od akbar rakam

→ Ordering of its vertices along a horizontal line so that all directed edges go from left to right

→ Time $\Theta(V+E)$

Strongly Connected Components:

→ Kosaraju's Algorithm:

1- Create an empty stack s

2- Do DFS, push the finished vertex to stack

3- G^T → reverse directions of all edges

4- One by one pop a vertex from stack. Take v as a source and do DFS on G^T in G^T. The DFS starting from v prints strong connected components.

5- Output the vertices of each tree as a separate SCC

→ when there is a path between all pairs of vertices

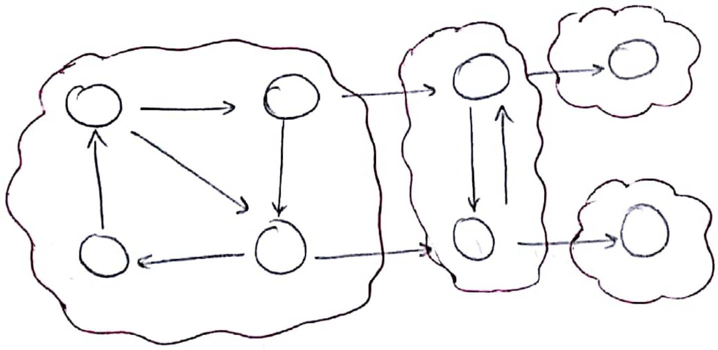
→ SCC of a directed graph is a maximal strongly connected subgraph $(u \to v, v \to u)$

→ Call DFS → $O(V+E)$

G^T → $O(V+E)$

Call DFS on G^T → $O(V+E)$

$\therefore O(V+E)$



Slide 13

stack ⇒ b e a c d g h f

From GT

1- b-a-e

2- c-d

3- g ~~f~~ f

4- h