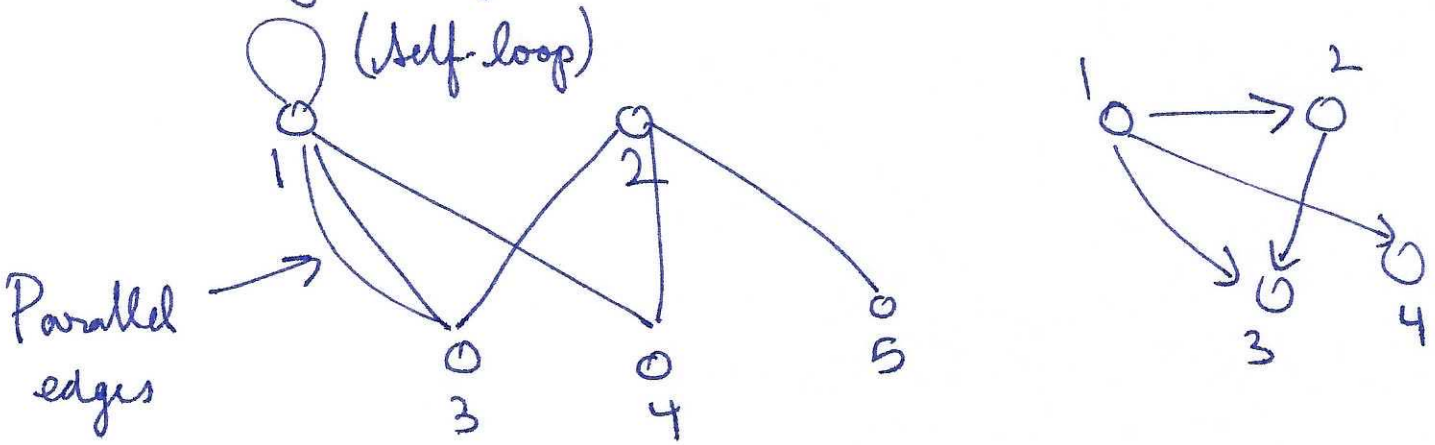
Graphs

Graph is a (data structure)/combinatorial object representing "entities" (vertices/nodes) and "relationships" (edges) between them.



$G = (V, E)$ V is a set of vertices (in our case $[n]$)

E is a set of pairs of vertices

E is unordered : undirected graph

E is ordered pairs : directed graph

Undirected : $(u,v) \in E$ iff $(v,u) \in E$

Weight : Edges may have weights associated with them

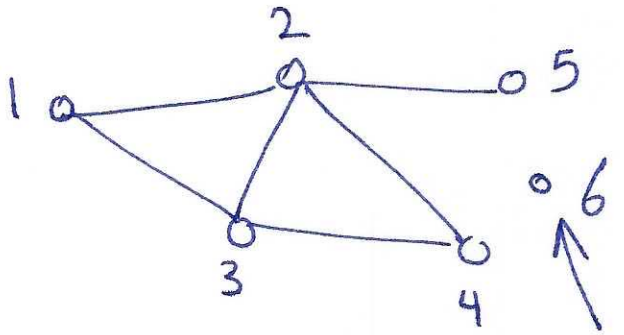
Simple : unweighted, undirected, no self-loops

Simple graphs

v is a vertex

$N(v)$: neighborhood of v $= \{u \mid (u,v) \in E\}$

$\Gamma(v)$



$N(2) = \{1,3,4,5\}$

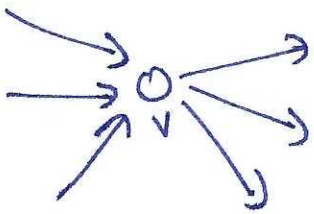
$N(1) = \{2,3\}$

Isolated vertex

In directed graph,

out-neighborhoods $N^+(v)$

In-neighborhoods $N^-(v)$

$d^-(v)$ $N^-(v)$  $N^+(v)$ $d^+(v)$

$d(v)$: degree of vertex v $= |N(v)|$ = # edges incident to v

$=$ # neighbors

How to represent a graph ?

Graphs change over time, but we also wish to process fixed graphs.

(Raw) List of edges

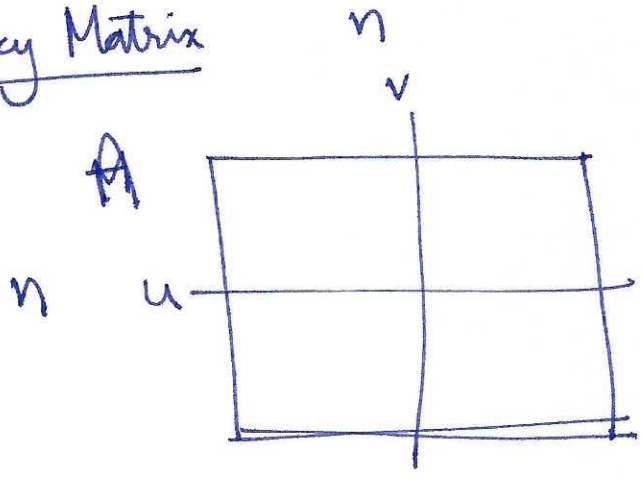
List of pairs

(u_1, v_1)	(u_2, v_2)	(u_3, v_3)	— —	(u_K, v_K)

n = #vertices m = #edges

Storage size for list of edges = $O(m)$

Adjacency Matrix



$A(u,v) = \begin{cases} 1 & \text{if } (u,v) \text{ is edge} \\ 0 & \text{else} \end{cases}$

= weight of edge (u,v)

A is $n \times n$ matrix

If $A^T = A$, then graph is undirected.

Q. How much storage does adjacency matrix use?

(R) $\Theta(n)$ (G) $\Theta(m)$ ✓ (B) $\Theta(n^2)$

n = 100 million = 10^8 Adj matrix = $(10^8)^2$ bits

$m = 100 \times 10^8 = 10^{10}$

Terabyte = 10^{16} bits

Peta ↑ = 1000 TB = 1 Petabyte $\approx 10^{15}$ bytes

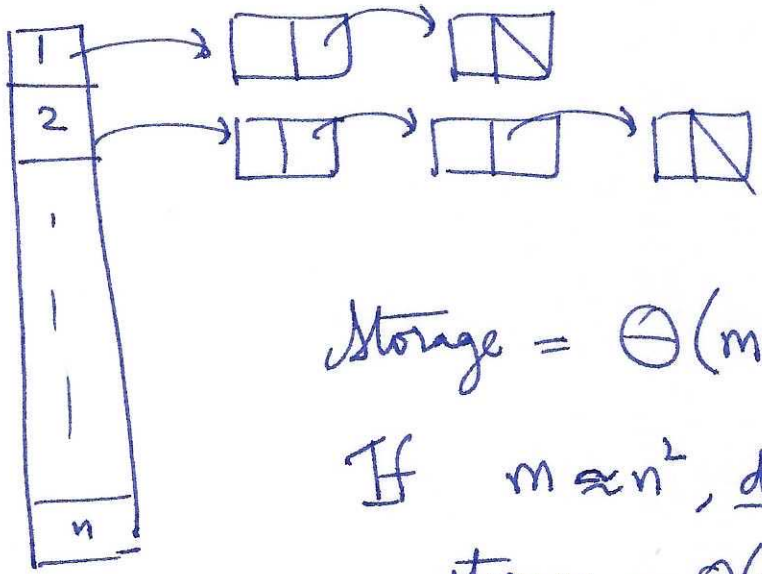
10^{10} edges = 20×10^{10} bytes

= 2×10^{11} bytes = 200 GB

Adjacency list Array of linked lists

Each $N(v)$ is stored as a linked list

$V = [n]$



Storage = $\Theta(m+n)$

If $m \approx n^2$, dense graph

Storage = $\Theta(n^2)$ pretty much same as adj. matrix

$m \ll n^2$

When $m = O(n)$, graph is sparse.

	Is (u,v) an edges?	Get all of $N(v)$	Insert edges	Delete edge (with ptr)
Raw list of edges	$\Theta(m)$	$\Theta(m)$	$\Theta(1)$	$\Theta(1)$
Adj matrix	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Adj list	$\Theta(d(v))$	$\Theta(d(v))$	$\Theta(1)$	$\Theta(1)$

$d(v) = O(n)$

degree

Traversal/explore operations

Change operations

BFS (Breadth First Search)

Technically, BFS is "just" a method to traverse a graph.

visit all vertices and edges

It's really a technique to solve MANY problems.

Adjacency list representation

Def: Vertex u ~~to v~~ is connected to vertex v if there exists a path of vertices from v to u

BFS(s) has a single argument, source vertex

↖ source vertex

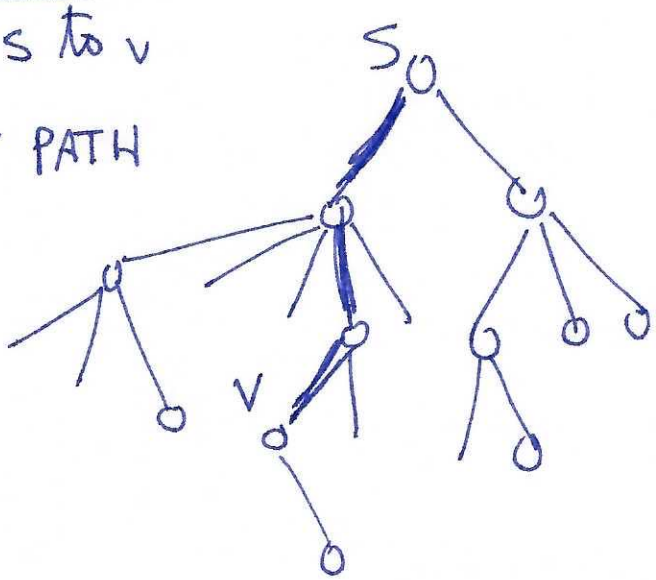
BFS(s) outputs a tree of vertices with s at the root.

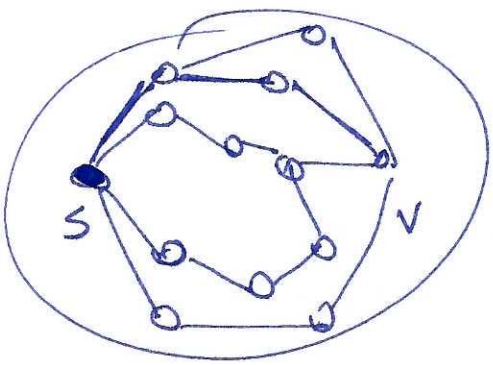
This tree has the following properties

(1) ALL vertices connected to s appear in the tree

(2) **The tree is a shortest-path tree.**

For any v, the path from s to v in the tree is a SHORTEST PATH from s to v in G.





Def: A path from s to v is a sequence of edges $(s, u_1)(u_1, u_2)(u_2, u_3) - - (u_k, v)$. The length of the path is the number of edges in this path.

Def: The shortest path distance (distance) between s and v, denoted $d(s,v)$ is the length of a shortest (min. length) path from s to v

BFS(s) : $V = \{1, 2, \ldots, n\}$

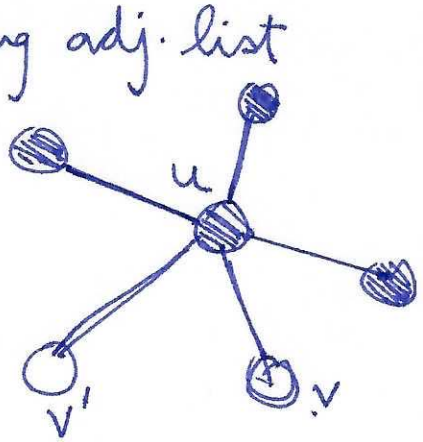
pred: array indexed by V

pred[v] ← parent of v in BFS/shortest path tree

visited: boolean array indexed by V

$$\text{visited}[v] = \begin{cases} \text{false} & \text{if BFS(s) does NOT visit v} \\ \text{true} & \text{if BFS(s) visits v} \end{cases}$$

1. Q is a FIFO queue initialized with s.
2. pred is initialized to all null and visited is all False. (but visited[s]=True)
3. while (Q is not empty):
   (i) u = dequeue(Q)
   ~~(ii) visited[u] = True~~
   (iii) ~~if (visi~~ For every $v \in N(u)$:
      (a) if (visited[v] is False)
         (1) pred[v] = u
         (2) enqueue(Q, v) // v is added to queue
         (3) visited[v] = True



When BFS(s) terminates, the following hold.

(1) visited[v] = True iff v is connected to s

(2) pred[v] = null iff v = s or v is not connected to s

(3) For v connected to s,

the vertices v, pred[v], pred[pred[v]]. . . . , s

form a shortest path.