Graphs – 1

- Graph is a D.S that Consist of following
1) A finite Set of vertices also Called node
2) A finite set of ordered pair of the form (u,v) Called as edges. Ordered (u,v) is not Same as (v,u) in Case of directed graphs (di-graph)

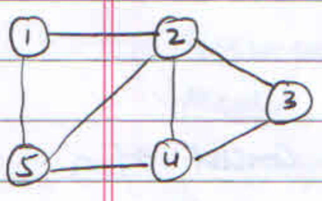
- The pair (u,v) indicate there is an edge from vertex u to v. and may Contain weight.

Representations of graphs

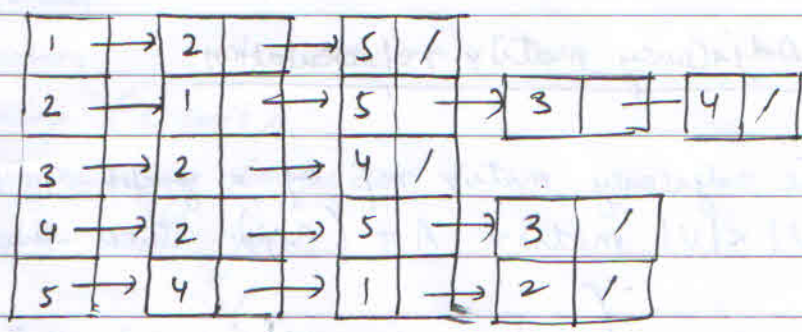
- The 2 most Common Computational representations of graphs

1) Adjancy List

- The adjacency list representation of a graph G = (V,E) Consist of an array Adj of |V| lists, one for each Vertex in V.
- for each u ∈ V the adjacency list Adj[u] Contain all the vertices V Such that there is an edge (u,V) ∈ E
- Adj[u] Consists of all the vertices adjacent to u in G.



(Grap G)



Adjacency list representation

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency matrix representation

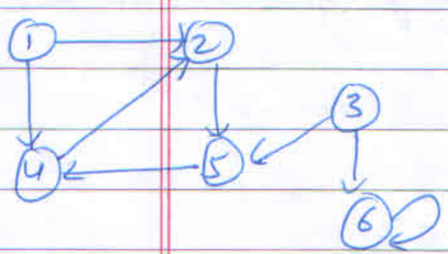
- Provide a compact way to represent sparse graph ($|E|$ is much less than $|V|^2$.

classmate
Date
Page

- Directed graph - Sum of length of all adjacency lists is $\underline{|E|}$

- UnDirected graph - Sum of length of all adjacency list is $\underline{2|E|}$

- for both Directed / undirected graph adjacency list representation requires $\underline{\Theta(V+E)}$ amount of memory.



1 → 2 → 4 /
2 → 5 /
3 → 6 → 5 /
4 → 2 /
5 → 4 /
6 → 6 /

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Disadvantages

- Provide no quicker way to determine whether a given edge(u,v) is present in the graph, only way is to search v in the adjacency list Adj[u].

- Adjacency matrix rep remedies this disadvantage but at the cost of using asymptotically more memory.

② Adjacency matrix representation

- The adjacency matrix rep of a graph $G = (V, E)$ consist of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- It require $\Theta(V^2)$ memory, independent of the no of edges in the graph.

(AJAY RAWAT)

- Adjacency matrix A of an undirected graph is its own transpose : $A = A^T$

Notes

- Adjacency list representation is asymptotically space efficient as compared to adjacency matrix representation.

- Adjacency matrix representation is simpler so we prefer when graphs are reasonably small and for unweighted graph require only one bit per entry.

- Adjacency list rep provides a compact way to represent sparse graph (for which $|E|$ is much less from $|V|^2$

- Adjacency matrix rep is preferred when graph is dense $|E|$ is close to $|V^2|$.

Applications

- Web crawling
- Social Networking
- Network broadcast
- Garbage Collection
- Model checking (Circuit)
- Checking mathematically conjucture
- Solving puzzles and games.

classmate

Date
Page

Breadth first Search

- Given a graph $G = (V, E)$ and a Source vertex s, BFS Systematically explores the edges of G to discover every vertex that is reachable from s.

- It compute the distance (Smallest) from s to each reachable vertex

- It produce a breadth-first-tree with root s that contains all reachable vertices.

- for any vertex v reachable from s, the Simple path in the BFT from s to v Corresponds to Shortest path from s to v in G.

- BFS is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

- BFS works for both directed and undirected graphs.

- BFS is represented using Adjacency lists.

- $u.color$ - Color of each vertex u
  $u.\pi$ - Predecessor of u.
  $u.d$ - Distance from s to u.
  Q - Queue (FIFO)

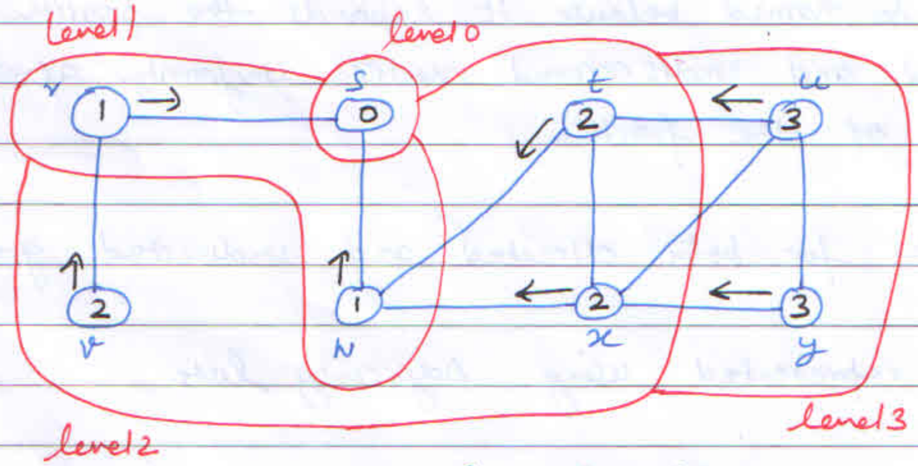
<u>BFS(G, s)</u>

1. for each vertex $u \in G.V - \{s\}$
2. $u.color = white$
3. $u.d = \infty$
4. $u.\pi = NIL$

(AJAY RAWAT)

classmate
Date
Page

```
S.Color = Gray
S.d = 0
S.π = NIL
Q = ϕ
ENQUEUE(Q, s)
While Q ≠ ϕ
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.Color == White
            v.color = Gray
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = Black.
```



frontier ϕ = {s}
$level_1$ = {r, w}
$level_2$ = {v, t, x}
$level_3$ = {u, y}

BFS Example

Analysis

- operations of enqueuing and dequeuing take O(1) time and so the total time devoted to queue operations is O(V)
-
- Because the procedure scan adjacency list of each vertex only when the vertex is dequeued, it scan each adjacency list at most once. (length of Adj list is $\Theta(E)$)

- Total running time of the BFS procedure is $O(V+E)$

Shortest Path

- To find a shortest path, take v, parent[v], parent[parent[v]] until s (or None)

- For every vertex v, fewest edges to get from s to v is

$$\begin{cases} \text{level}[v] & \text{if } v \text{ assigned level} \\ \infty & \text{else (no path)} \end{cases}$$

- Parent pointers form shortest path tree = Union of such a shortest path for each v.

Application of BFS

1. finding all node from one connected component.
2. finding shortest path between two nodes u and v and length of such path.
3. Construct a BSF tree / forest from a graph.
4. Testing graph for bipartiteness.

Depth-first-Search

- DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.

- Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.

- This process continue until we have discovered all the vertices that are reachable from the original source vertex.

- If any undiscovered vertices remain, the DFS select one of them as new source, and it repeats the search from that source.

- Algorithm repeats this process until it has discovered every vertex.

- Each vertex is initially white, is grayed when it is discovered in the search and is blackened when it is finished. (when adjacency list is examined completely.)

- DFS timestamps each vertex. Each vertex has two timestamps
1) $v.d$ records when v is first discovered (grayed)
2) $v.f$ records when the search finishes examining v's adjacency list (blackens v).

- Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$ and BLACK thereafter.

(AJAYRAWAT)

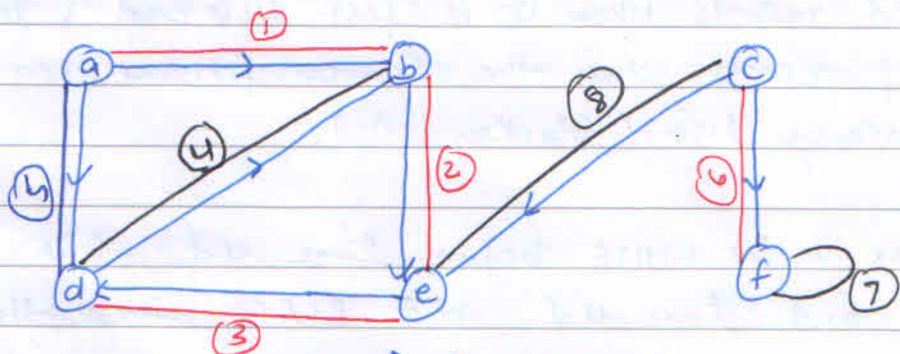
DFS(G)

```
for each vertex u ∈ G.V
        u.color = white
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == white
                DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
time = time + 1          // white vertex u has just been discovered
u.d = time
u.color = Gray
for each v = G.Adj[u]         // explore edge (u, v)
        if v.color == White
                v.π = u.
                DFS-VISIT(G, v)
u.color = Black
time = time + 1
u.f = time
```

Example



DFS

Analysis

- The loops on line 1-3 and lines 5-7 of DFS take $\Theta(V)$ time exclusive of the Calls to DFS-VISIT.

- During the execution of DFS-VISIT(G, v) the loop on line 4-7 executes $|Adj[v]|$ times.

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

So total Cost of executing 4-7 of DFS-VISIT $(\Theta(E)$.
The running time of DFS is

$$\Theta(V+E) \qquad (\text{linear time})$$

Application of DFS

- Detecting a Cycle in a graph.
- Path finding
- DFS traversal produces Minimum Spanning Tree and All pair shortest Path Tree.
- Topological Sorting
- To test if a graph is bipartite
- finding Strongly Connected Component of a graph
- Solving puzzles with one Solution.