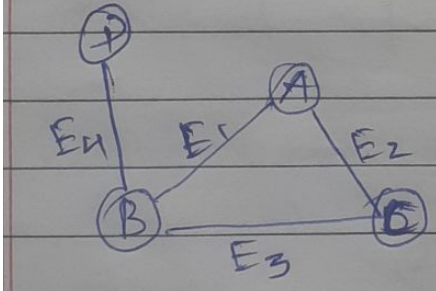
Graph Representation

Incidence Matrix

-> An incidence matrix is a matrix where each column represents an edge - connected to two vertices.

In this representation, the graph is represented using a matrix of V*E.

Possible values for
Undirected graph: 0, 1
Directed graph: 0, 1, -1



	E_1	E_2	E_3	E_4
A	1	1	0	0
B	1	0	1	1
C	0	1	1	0
D	0	0	0	1

*i,j if is a weighted graph '1' will be replaced as value.

In directed graph:

Out going edge : 1
incoming Edge : -1
No edge : 0

Disadvantages:-

→ uses $O(VE)$ space as opposed to $O(V^2)$ for the adjacency matrix.

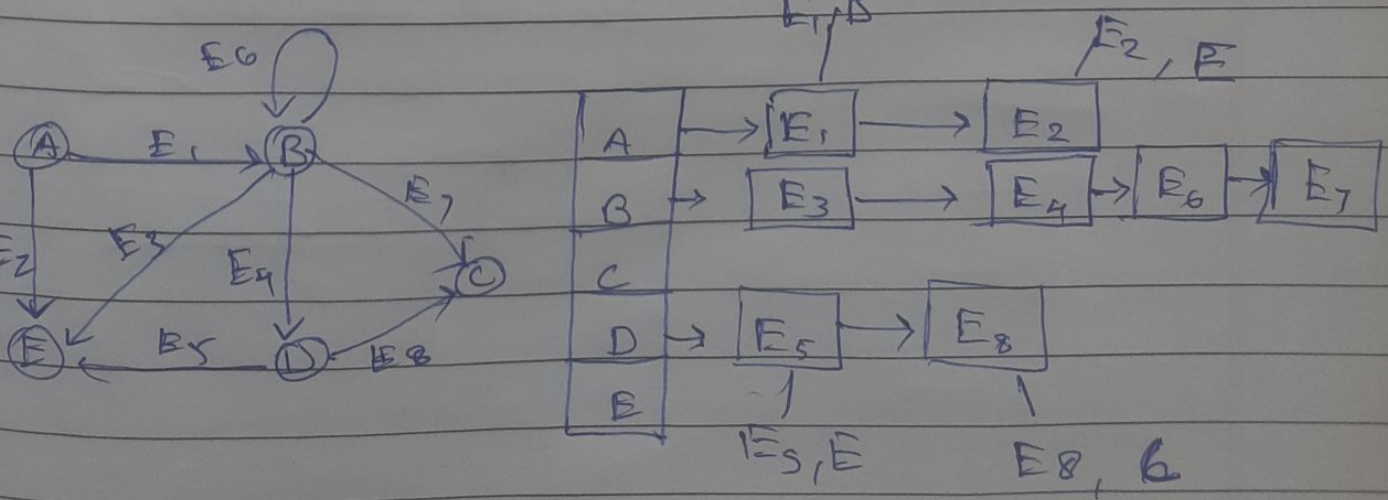
→ Checking if a node is related to some other node is $O(E)$, so it's worse than an adjacency matrix for this.

→ Traversing a node's adjacencies is $O(E)$.

Incidence List:-

A list which stores for each vertex a list of objects representing the edges incident to that vertex.

To complete the structure, each edge must point back to the two vertices forming its endpoints.



Graph Traversal (Search)

→ Technique used for visiting each vertex of the graph.

For:-

→ Finding all reachable nodes (for garbage collection)

→ Finding the best reachable node (single-player game search) or the minmax best reachable node (two-player game search).

→ Finding a best path through a graph (for routing and map directions).

Classification:-

Depends on the order in which vertices are visited.

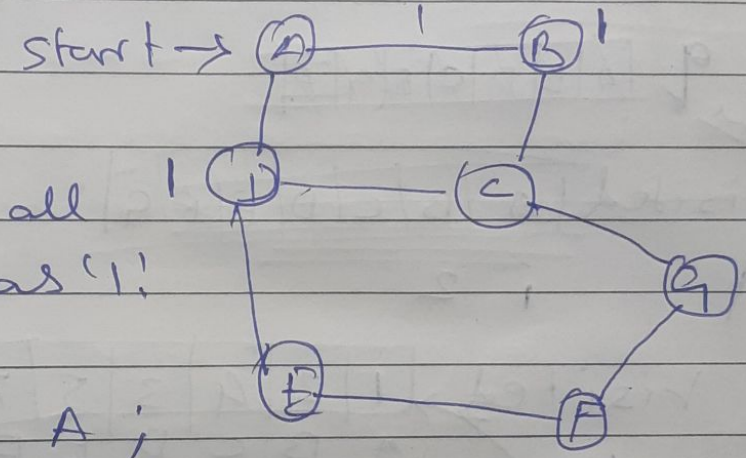
→ BFS - Breadth First Search.

→ DFS - Depth First Search.

Breadth First Search (BFS)

"Like an army of soldiers spreading out to cover the area".

→ Start traversal from a node and explore graph layerwise.



Let's have all the weights as '1'.

Then, From A;

B & D are in distance → 1 (1st layer)

C, D → 2 (2nd layer)

G, F → 3 (3rd layer)

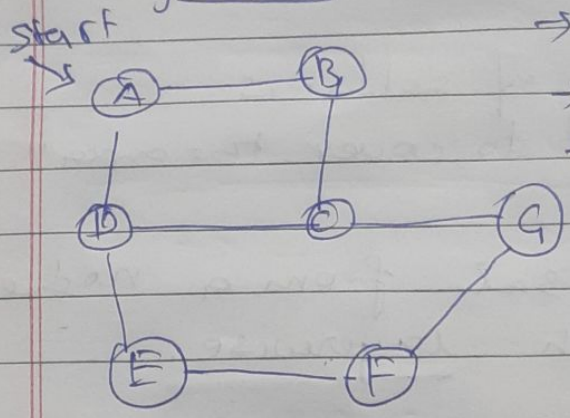
Data structure:

FIFO data structure (Queue)

To avoid cycles:-

→ Keep track of visited nodes.

Algorithm:-



→ add start vertex to queue q
→ mark start as visited
→
while q != empty:
v = front of queue.
for each adjacent vertex aw of v:
if aw is not visited:
add aw to queue q
mark aw as visited

q | A | B | D | C | E | G | F |

front of the queue will be dequeued

visited | A | B | C | D | E | F | G |
1 2

visited → | 1 | 2 | 4 | 3 | 5 | 7 | 6 |
A B C D E F

~~start~~ ith encounter

O/P:-

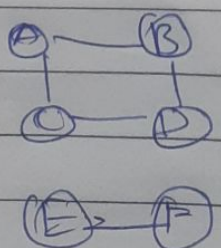
A B D C E G F

there can be multiple results here, for example:-

A D B C E G F

A D B E C F G

In another case:



here, A B C D E F

first connected vertices.

then,

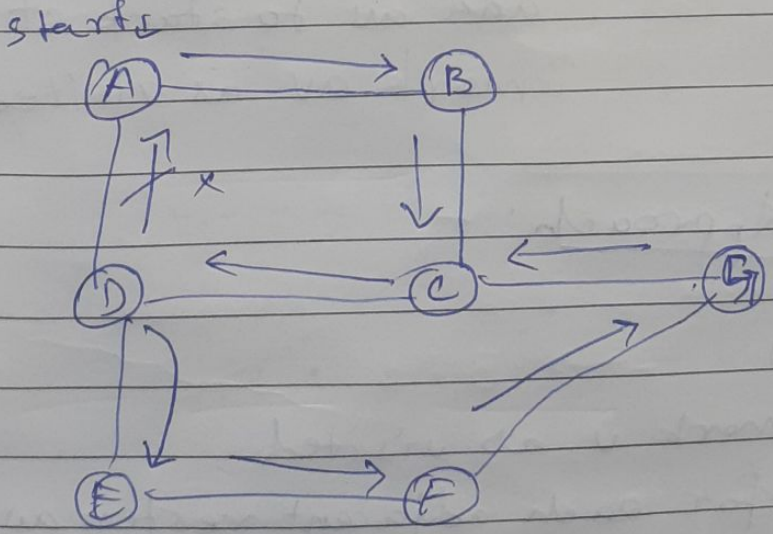
Start From E.

Applications :-

→ Peer to peer network.
→ Social Networking Websites.
→ GPS Navigation System
→ Path Finding
→ Broadcasting in Network.
→ Garbage collection.

Depth First Search (DFS).

Like a single searcher probing unknown area as deeply as possible, retreating only when meeting dead ~~ends~~ end.



Start traversal from a node and goes as far as it can a given path, then backtracks, until it finds an unexpected path, and then explores it.

DS to use:

LIFO data structure (Stack)

To avoid cycles:

→ keep track of visited nodes.

Algorithm:-

add start vertex to stack st
mark start as visited

while st != empty :
    v = top of stack , pop stack.
    for each adjacent vertex av of v :
        if av is not visited :
            add av to stack st
            mark av as visited

Recursive Approach:-

```
dfs(v)
    mark v as visited
    for each adjacent vertex av of v :
        if av is not visited :
            dfs(av)
```

Applications:-

→ Topological sorting. (example in OS)
→ Scheduling problem.
→ Cycle detection graph.
→ Solving puzzles with only one solution. eg maze.

BFS vs DFS

BFS	DFS
Start traversal from root node and visit nodes in level by level manner (i.e visit the ones closest to the root first)	Starts the traversal from the root node and visit nodes as far as possible from root node. (ie depth wise)
→ Uses FIFO (e.g Queue	→ uses LIFO (eg Stack)
Time Complexity $O(V+E)$	$O(V+E)$
→ Space complexity: $O(V)$, $O(b^d)$	→ Space Complexity $O(V)$, $O(bd)$
b - branching factor, argautdgree d - distance from start node	
" Best case of one is worst case of another one (vice versa)	
→ Optimal	→ Not Optimal.

[A search algorithm is optimal if when it finds a solution, it is the best one e.g the shortest]

To choose :-

-> Depends on the structure of the graph and the number and location of searched for items.

-> If you know a solution is NOT FAR from the root of the tree
--> BFS.

-> If solutions are frequent but located deep in the tree --> DFS.

-> If the tree is very deep and solutions are rare, DFS would be too slow.

-> if the tree is very wide, a BFS might need too much memory, so it might be completely impractical.

~~Examples~~ Examples.

* Facebook/LinkedIn Friend suggestion:
--> BFS

* Solve maze or sudoku having only one solution ?
--> DFS

* Detect cycle in a graph.
DFS (preferred), BFS.