Tutorial - 5

Q) What is the difference between BFS and DFS. Please write the applications of both the algorithms.

Breadth First Search (BFS)	Depth first Search (DFS)
(i) BFS uses queue data structure for finding the shortest path	(i) DFS uses stack data structure
(ii) BFS is better when target is closer to source.	(ii) DFS is better when target is far from source.
(iii) As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	(iii) DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision.
(iv) BF is slower than DFS	(iv) DFS is faster than BFS
(v) Time complexity = $O(V+E)$	(v) Time complexity = $O(V+E)$

Applications of DFS

(i) Used to create minimum spanning tree for all pair shortest path tree.

(ii) We can detect cycles in a graph

(iii) Used to find path between two given vertices u & v.

(iv) Topological sorting can be done using DFS

(v) Used to find strongly connected components of a graph.

Applications of BFS:-

(i) In peer to peer network like bit-torrent, BFS is used to find all neighbour nodes.

(ii) Using GPS navigation system BFS is used to find neighbouring places.

(iii) In networking when we want to broadcast some packets, we use BFS

(iv) BFS is used in Ford-Fulkerson algorithm to find maximum flow in a network.

Q Which Data structure are used to implement BFS & DFS and why?

In BFS, we use queue data structure because we don't know the size of the frontier in advance, queue is more memory efficient. Also queue data structures are considered inherently "fair". The FIFO concept that underlines a queue will ensure that those things that were discovered first will be explored first.

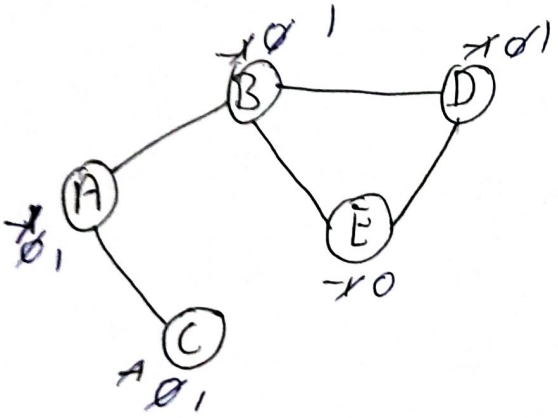
In DFS we uses stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Q3 Sparse graphs :- Sparse graph is a graph in which the no. of edges is close to the minimum no of edges. It can be disconnected graph.

Dense graph:- Dense graph is a graph in which the no. of edges is close to the maximal no. of edges.

- Adjacency lists are preferred for sparse graph
- Adjacency matrix for dense graph

4) Cycle detection in Undirected Graph (BFS)



-1 = unvisited
0 = into the queue
1 = traversed

queue:

A	B	C	D	E

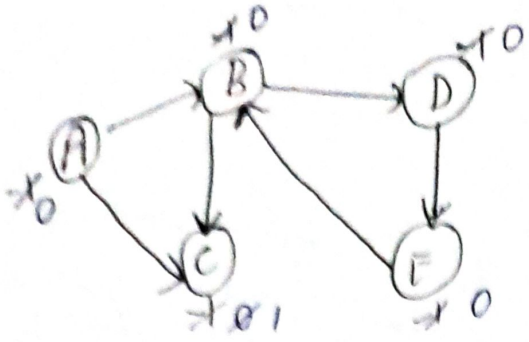
visited set:

A	B	C	D

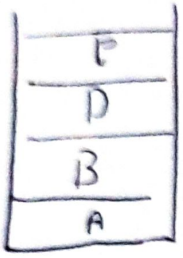
When D checks it adjacent vertices it finds E with 0.

→ If any vertex finds the adjacent vertex with flag 0, then it contains cycle

cycle detection in directed graph (DFS):



-1 = unvisited
0 = visited & in stack
1 = visited & popped out from stack



Visited set: ABCDE

⇒ B → D → E → B

parent map

vertex	Parent
A	-
B	A
C	B
D	B
E	D

there E finds B (adjcent vertex of E) with 0.
⇒ it contains a cycle

5) Disjointset data structure:
The disjoint set data structure is also known as union-find data structure & merge-find set. It is a data structure that contains a collection of disjoint or non-overlapping sets.

The disjoint set means that when the set is partitioned into the disjoint subsets, various operations can be performed on it.

In this case, we can add new sets we can merge the sets, & we can also find the representative number of a sets. It also allows to find out whether the two elements are in the same set or not efficiently

Operation on disjoint set

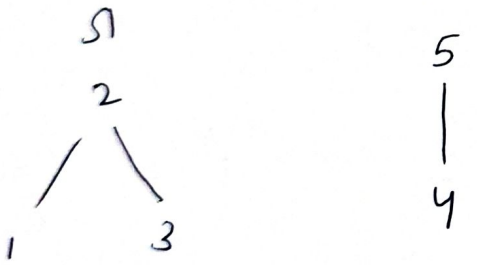
1. Union:

a) If S1 & S2 are two disjoint sets, their union S1 ∪ S2 is a set of all elements x, such that x is in either S1 or S2.

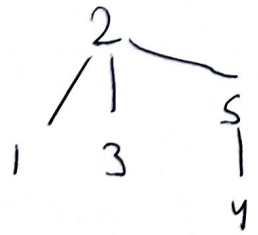
b) As the sets should be disjoint S1 ∪ S2 replaces S1 & S2 which no longer exists.

c) Union is achieved by simply making one of the trees as a subtree of other ie to set parent field of one of the roots of the trees to other root.

Ex



S1 ∪ S2

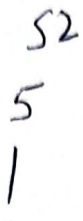


Merge the sets containing x & containing y into one.

2 Find

Given an element x, to find the set containing it

Eg



return in which set x belongs

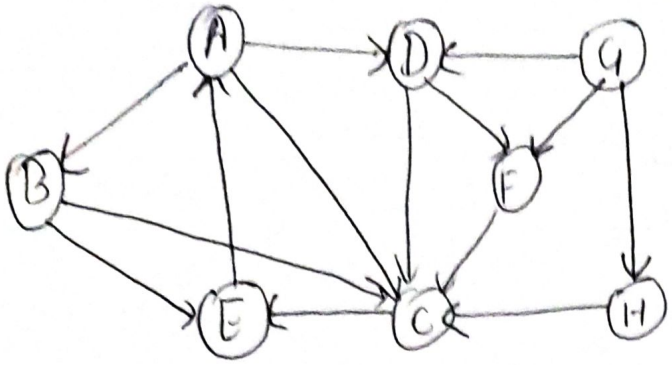
find (3) ⇒ S1

find (5) ⇒ S2

3] Path compression (Modifications to find()):

It speeds up the data structure by compressing the height of the trees. It can be achieved by inserting a small caching mechanism into a find operation

(6)



BFS

Queue	Action	Visited Node
G	Insert Push G	G
~~G~~DFH	Remove G Insert D, F H	GDFH
~~D~~FHC	Remove D Insert C	GDFHC
~~F~~HC	Remove F	GDFHC
HC	Remove H	GDFHC
CE	Remove C Insert E	GDFHCE
~~E~~A	Remove E Insert A	GDFHCEA
~~A~~B	Remove A Insert B	GDFHCEAB
B	Remove B	GDFHCEAB

Path ∴ G → D → F → H → C → E → A → B

DFS

Stack	Action	Node Visited
G	Push G	G
G D	Push D	G D
G D C	push C	G D C
G D C E	push E	G D C E
G D C E A	Push A	G D C E A
G D C E A B	push B	G D C E A B
G D C E A B ~~B~~	POP B	G D C E A B
G D C E A ~~A~~	POP A	G D C E A B
G D C E ~~E~~	Pop E	G D C E A B
G D C ~~C~~	POP C	G D C E A B
G D F	PUSH F	G D C E A B F
G D ~~F~~	POP F	G D C E A B F
G ~~D~~	POP D	G D C E A B F
G H	PUSH H	G D C E A B F H
G ~~H~~	POP H	G D C E A B F H
~~G~~	POP G	G D C E A B F H

Path : G → D → C → E → A → B → F → H

$V = \{a, b, c, d, e, g, h, i, j, l\}$

$E = \{(a,b), (a,c), (b,c), (b,d), (e,i), (e,g), (h,l), (j)\}$

	{a} {b} {c} {d} {e} {g} {h} {i} {j} {l}
(a, b)	{a, b} {c} {d} {e} {g} {h} {i} {j} {l}
(a, c)	{a, b, c} {d} {e} {g} {h} {i} {j} {l}
(b, c)	{a, b, c} {d} {e} {g} {h} {i} {j} {l}
(b, d)	{a, b, c, d} {e} {g} {h} {i} {j} {l}
(e, i)	{a, b, c, d} {e, i} {g} {h} {j} {l}
(e, g)	{a, b, c, d} {e, i, g} {h} {j} {l}
(h, l)	{a, b, c, d} {e, i, g} {h, l} {j}
(j)	{a, b, c, d} {e, i, g} {h, l} {j}

we have

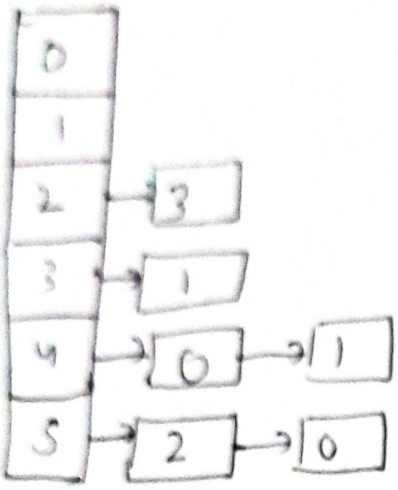
{a, b, c, d}

{e, i, g}

{h, l}

{j}

8)



Algo

1 Go to node 0, it has no outgoing edges so push node 0 into the stack & mark it visited

2. Go to node 1, again it has no outgoing edges, so push node 1 into the stack & mark it visited

3. Go to node 2, process all the adjacent nodes & mark node 2 visited

4 Node 3 is already visited so continue with next node

5) Go to node 4, all its adjacent nodes are already visited so push node 4 into the stack & mark it visited

c) Go to node 5, all it adjacent nodes are already visited so push node 5 into the stack & mark it visited

5 4 2 3 1 0
(output)

q) Heap is generally preferred for priority queue implementation because heaps provide better performance compared to arrays or linked

Algorithms where priority queue is used:

1) Dijkstra's Shortest Path Algorithm: when the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiency when implementing Dijkstra's algorithm.

2) Prism's algorithm: To store keys of nodes & extract minimum key node at every step.

10 Min Heap

(i) ~~For every pair of the parent~~ & d

(ii) In a Min-Heap the key present at the root node must be less that or equal to among the keys present at all of it children

(iii) In a min-heap the minimum key element present at the root.

(iv) A min-heap uses the ascending priority

(IV) In the construction of a min-heap, the smallest element has priority

(V) In a min-heap, the smallest element is the first to be popped from the heap

Max Heap

(i) In a max-heap the key present at the root node must be greater than or equal to among the keys present at all of its children.

(ii) In a max-heap the maximum key element present at the root.

(iii) a max-heap uses the descending priority

(iv) In the construction of a max-heap, the largest element has priority

(V) In a max-heap, the largest element is the first to be popped from the heap.