SAGAR NEGI SEC- H ROLL No:- 10

Tutorial 5

1. What is difference between DFS and BFS. Please write the application of both the algorithms.

Sol:-

DFS	BFS
→ DFS Stands for depth first search.	→ BFS stands for breadth first search.
→ It uses the data structure Stack.	→ It uses the data structure Queue.
→ It uses the concept of backtracking.	→ No concept of backtracking.
→ less memory required.	→ Requires more memory.
→ It is used to find a path from source node to destination node.	→ It is used to find single source shortest path in unweighted graph.

Applications :

(i) BFS

→ It is used for detecting cycles in a graph.

→ It is used for finding route from GPS.

→ It is used for finding shortest and minimum path in unweighted graph.

(ii) DFS

→ It is used for detecting cycles in a graph.

→ It is used for finding path between two vertices.

→ It is used for job scheduling process.

2. Which data structure are used to implement BFS and DFS and why?

Sol:- In DFS, we need to traverse a whole branch of a tree. So, to keep track on the current node, it requires last In first out approach which can be implemented by using stack. After it reaches depth of the node, then all the nodes will be popped out of stack.

→ In BFS, we have to go through the nodes with minimum number of nodes in between. So we don't have to look for all nodes. Therefore it uses queue data structure. If it uses stack then it will go through all the adjacent nodes which consumes more time and thus do not find minimum path.

3. What do you mean by sparse and dense graphs? Which representation of graph is better for sparse and dense graph?

Sol:- Dense Graph is a graph in which number of edges is close to the maximal number of edges.

Sparse Graph is a graph in which number of edges is close to the minimal number of edges.

→ for sparse graph, adjacency list is used.

→ for Dense graph, adjacency matrix is used.

4. How can you detect cycle in a graph using DFS and BFS?

Sol:- Using DFS:

(i) Create a graph using given no. of edges and vertices.

(ii) Create a recursive function that have current index visited array and parent node.

(iii) Mark current node as visited.

(iv) Find all vertices which are not visited and are adjacent to current node. Recursively call the function for these vertices. If the recursive function returns true, return true.

(v) If the adjacent node is not a parent and ~~is~~ already visited, then return true.

(vi) Call the recursive function for all the vertices and if any function return true, return true.

(vii) Else for all the vertices, the function returns false, return false.

Using BFS:

(i) Pick all vertices with status 0 and add them with a queue.

(ii) ~~Compute~~ Remove a vertex from queue and then

→ increment count by 1 for all its neighbouring nodes.

→ decrease status by 1 for all its neighbouring nodes.

→ If status of neighbouring nodes is reduced to 0, then add it to queue.

(iii) Repeat step (ii) until queue is empty.

(iv) If count is not equal to the no. of nodes in a graph then cycle otherwise not.

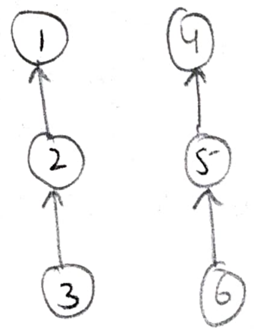
5. What do you mean by disjoint set data structure? Explain operation alongwith examples.

Sol:- Disjoint set are similar to sets in mathematics but they are modified for the usage in algorithms.

Operations:

(i) Find → It tells the set to which an element belongs.

eg→



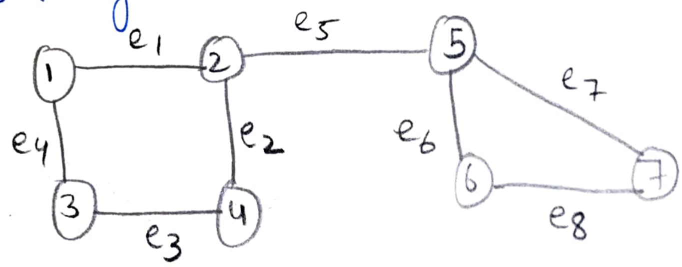
$S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$

find(1) = S_1 , find(5) = S_2

→ It is used basically to find their parent.

(ii) Union → It is used to merge two sets when an edge is added. If both the component belong to same set, then it forms a cycle.

$U = \{1, 2, 3, 4, 5, 6, 7\}$



① Add $e_1 = \{1, 2\}$
1, 2 belongs to U
$\therefore S_1 = \{1, 2\}$

② Add $e_2 = \{2, 3\}$
2 is in 'S_1' and 3 is in 'U'
$\therefore S_1 = \{1, 2, 3\}$

③ Add $e_3 = \{3, 4\}$
$\therefore S_1 = \{1, 2, 3, 4\}$

④ Add $e_4 = \{4, 1\}$
Both 4, 1 belong to S_1
$\therefore$ cycle detected.

⑤ Add $e_5 = \{2, 5\}$
$\therefore S_1 = \{1, 2, 3, 4, 5\}$

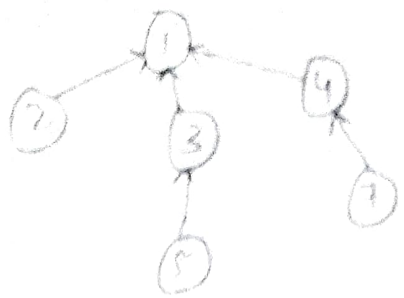
⑥ Add $e_6 = \{5, 6\}$
$\therefore S_1 = \{1, 2, 3, 4, 5, 6\}$

⑦ Add $e_7 = \{6, 7\}$
$\therefore S_1 = \{1, 2, 3, 4, 5, 6, 7\}$

⑧ Add $e_8 = \{5, 7\}$
Both 5 and 7 belong to S_1
$\therefore$ cycle detected.

③ Path compression → It speeds up data structure by compressing the height of tree.

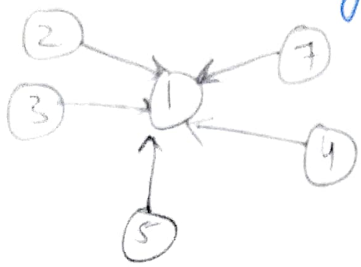
eg →



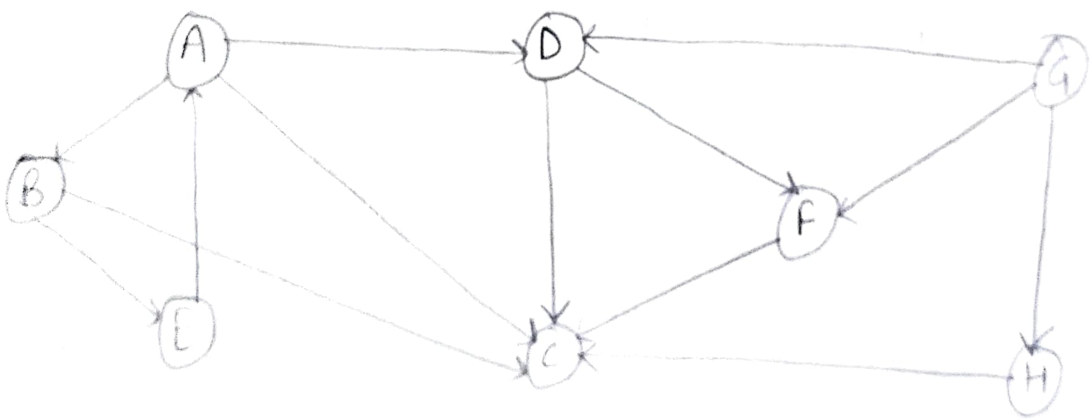
Now parent of 5 = find(5) = 3 → 1

and parent of 7 = find(7) = 4 → 1

To speed up data we can directly make path of 7 and 5 to 1.



6. Run DFS and BFS on following graph:



BFS:

Node	A	D	B	C	f	E
parent	–	A	A	A	D	B

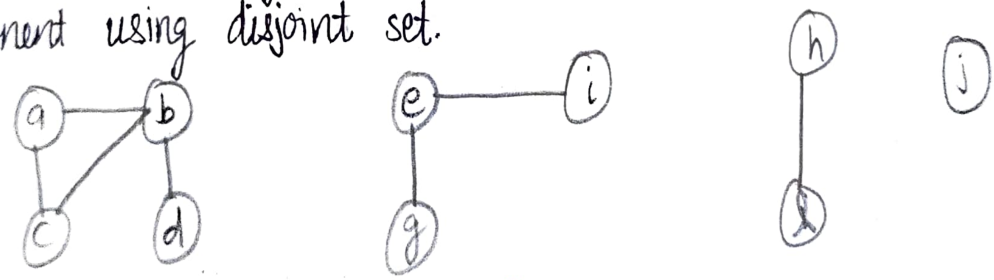
Path: A → B → E

DFS:

Node	Stack (←)
	A
A	BDC
C	BDE
E	BD
D	BF
F	B
B	G
G	H
H	

Path :- $A \to C \to E \to D \to F \to B \to G \to H$

7. Find out the number of connected components and vertices in each component using disjoint set.



$S_1 = \{a, b, c, d\}$
$S_2 = \{e, i, g\}$
$S_3 = \{h, l\}$
$S_4 = \{j\}$

Connection from a :-

Find (a) = S_1

- (a, b) = find (b) = S_1
  connected
- (a, c) = find (c) = S_1
  connected

- (a, g) = find (g) = S_2
  not connected
- (a, h) = find (h) = S_3
  not connected

- (a,d) = find (d) = S_1
  connected
- (a,e) = find (e) = S_2
  not connected
- (a,i) = find (i) = S_2
  not connected
- (a,l) = find (l) = S_3
  not connected
- (a,j) = find (j) = S_4
  not connected

* connected from a:
  b, c, d

* connection from b: find (b) = S_1
- (b,a) = find (a) = S_1
  connected
- (b,c) = find (c) = S_1
  connected
- (b,d) = find (d) = S_1
  connected
- (b,e) = find (e) = S_2
  not connected
- (b,i) = find (i) = S_2
  not connected
- (b,g) = find (g) = S_2
  not connected
- (b,h) = find (h) = S_3
  not connected
- (b,l) = find (l) = S_3
  not connected
- (b,j) = find (j) = S_4
  not connected

* connected from b
  a, c, d

* Connection from c: find (c) = S_1
- (c,a) = find (a) = S_1
  connected
- (c,b) = find (b) = S_1
  connected
- (c,d) = find (d) = S_1
  connected
- (c,e) = find (e) = S_2
  not connected
- (c,i) = find (i) = S_2
  not connected
- (c,g) = find (g) = S_2
  not connected
- (c,h) = find (h) = S_3
  not connected
- (c,l) = find (l) = S_3
  not connected
- (c,j) = find (j) = S_4
  not connected

* connected from c:
  a, b, d

Connection from d: find $(d) = S_1$

- $(d,a) =$ find $(a) = S_1$
  connected
- $(d,b) =$ find $(b) = S_1$
  connected
- $(d,c) =$ find $(c) = S_1$
  connected
- $(d,e) =$ find $(e) = S_2$
  not connected
- $(d,i) =$ find $(i) = S_2$
  not connected.
- $(d,g) =$ find $(g) = S_2$
  not connected
- $(d,h) =$ find $(h) = S_3$
  not connected
- $(d,j) =$ find $(j) = S_4$
  not connected

* connected from d:
a, b, c

Connection from e: find $(e) = S_2$

- $(e,a) =$ find $(a) = S_1$
  not connected
- $(e,b) =$ find $(b) = S_1$
  not connected
- $(e,c) =$ find $(c) = S_1$
  not connected
- $(e,d) =$ find $(d) = S_1$
  not connected
- $(e,i) =$ find $(i) = S_2$
  connected
- $(e,g) =$ find $(g) = S_2$ connected
- $(e,h) =$ find $(h) = S_3$
  not connected
- $(e,j) =$ find $(j) = S_4$
  not connected

connected from e: i, g

Similarly i is connected from : R, g

and g is connected from : e, i

connection from h: find (h) = S_3

- (h, a) = find (a) = S_1
  not connected
- (h, b) = find (b) = S_1
  not connected
- (h, c) = find (c) = S_1
  not connected
- (h, d) = find (d) = S_1
  not connected
- (h, e) = find (e) = S_2
  not connected
- (h, i) = find (i) = S_2
  not connected
- (h, g) = find (g) = S_2
  not connected
- (h, l) = find (l) = S_3
  connected
- (h, j) = find (j) = S_4
  not connected

→ connected from h: l

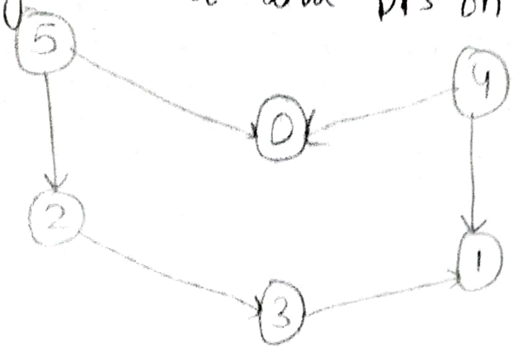
similarly connected from l: h

Connection from j: find (j) = S_4

- (j, a) = find (a) = S_1
  not connected
- (j, b) = find (b) = S_1
  not connected
- (j, c) = find (c) = S_1
  not connected
- (j, d) = find (d) = S_1
  not connected
- (j, e) = find (e) = S_2
  not connected
- (j, i) = find (i) = S_2
  not connected
- (j, g) = find (g) = S_2
  not connected
- (j, h) = find (h) = S_3
  not connected
- (j, l) = find (l) = S_3
  not connected

→ Connected from j = ϕ

8. Apply topological sort and DFS on following graph:



Sol:- DFS

Node	Stack
–	5
5	0 2
2	0 3
3	0 1
1	0
0	4
4	

Path: $5 \to 2 \to 3 \to 1 \to 0 \to 4$

Topological sort

Stack	Node
–	5
0	50
0	52
0	523
0	5231
01	523
013	52
0132	5
01325	4
013254	

Path:- $4 \to 5 \to 2 \to 3 \to 1 \to 0$

9. Heap data structure can be used to implement priority queue? Name few graph algorithm where you need to use priority queue and why.

Sol:- Heap can be use to implement priority queue because in heap, the highest (or lowest) priority element is always stored at the root.

→ However heap is not sorted, it can be regarded as ~~the~~ partially ordered.

→ It is useful when to remove highest or lowest priority.

<u>Use of priority Queue</u>:

(i) <u>Dijkstra's shortest path</u>: Priority queue is used to extract minimum efficiently node when implementing Dijkstra Algorithm.

(ii) <u>Prim's algorithm</u>: Priority queue is used to store keys of node and extract minimum key node at every step.

(iii) <u>Huffmann algorithm</u>: Priority queue uses data to compress data.

(iv) <u>A* search algorithm</u>: Priority queue is used to keep track of unexplored routes.

10. What is the difference between min and max heap?

Sol:-

Min Heap	Max Heap
→ The key present at root must be less than or equal to other nodes.	→ The key present at root must be greater than or equal to other nodes.
→ Minimum key element is present at root.	→ Maximum key element is present at root.
→ uses ascending priority.	→ uses descending priority.
→ smallest element has priority.	→ largest element has priority.