Tutorial-5

Sol 1:

BFS	DFS
→ BFS stands for Breadth first search.	→ DFS stands for Depth first search.
→ BFS uses queue to find the shortest path.	→ DFS uses stack to find the shortest path.
→ BFS is better when target is close to source.	→ DFS is better when source is far from source.
→ As BFS considers all the neighbours so it is not suitable for decision tree used in puzzle games.	→ DFS is more suitable for decision tree, As with one decision we need to transverse further to argument the decision, if we reach the conclusion we mean.

Application of BFS-

→ BFS is used in detecting cycles in graph.
→ Finding shortest path and minimal spanning tree in unweighted graph.
→ Finding a route through GPS navigation system with minimum number of crossings.
→ In networking finding a route for packet transmission.
→ In building the index by search engine crawlers.
→ In peer-to-peer networking, BFS is used to find neighbouring node.
→ In garbage collection, BFS is used for copying garbage.

Application of DFS-

→ If we perform DFS on unweighted graph, then it will create minimum spanning tree for all pair shortest path tree.
→ we can detect ~~grap~~ cycles in graph using DFS. If we get one back-edge during BFS, then there must be one cycle.

using DFS we can find path between two given vertices u & v.
we can perform topological sorting is used to scheduling jobs from given dependencies among jobs
we can find strongly connected components of a graph. If there is a path from each vertex to every other vertex, then it is strongly connected.

Solution 2: BFS (Breadth First Search) uses queue data structure for finding shortest path.
DFS (Depth First Search) uses stack data structure.
A queue (FIFO - First In First Out) is used by BFS. you mark any node in the graph as root and start transversing the data from it. BFS transverses all the nodes from the graph and keeps dropping them as completed. BFS visits an adjacent unvisited node, marks it as done and insert it into a queue.
DFS transverses a graph in depth word motion nd uses a stack to remember to get the next vertext to start a search, when a dead end occurs in any iteration.

Solution 3: Sparse graph :- A graph in which the number of edges are much less than the possible number of edges.
Dense graph: A graph in which the number of edges is close to the maximal number of edges.
If the graph is sparse we should store it as a list of edges. If the graph is dense, we should store it as adjacency matrix.

Solution 4: The existence of a cycle in directed and undirected graph can be determined by whether DFS finds an edge that points to an ancestors of the current vertex (it contains a back edge). All the back edges which DFS skips over are part of cycles.

* Detect cycle in a directed graph.

→ DFS can be used to detect cycle in a graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge that is from a node to itself or one of its ancestors in the tree produced by DFS.

→ For a disconnected graph, get the DFS forest as output. To detect cycle check for a cycle in individual trees by checking back edges.
To detect a back edge, keep track of vertices currently in the recursion stack of function for DFS transversal. If a vertex is reached that is already in the recursion stack, then there is a cycle in the tree. The edge that connect the current vertex to the vertex in the recursion stack is a back edge. use recstack[] array to keep track of vertices in the recursion stack.

Detect cycle in undirected graph-

Run a DFS from every unvisited node. DFS can be used to detect a cycle in a graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back-edge present in the graph.

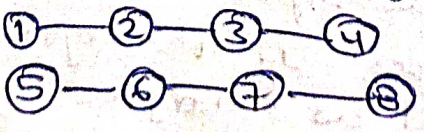
Solution 5: Disjoint set data structure-

→ It allows to find out whether the two elements are in the same set or not efficiently.

→ It can be defined as the subsets where there is no common element between two sets.

$S_1 = \{1, 2, 3, 4\}$

$S_2 = \{5, 6, 7, 8\}$



Operations performed-

(1) Find- can be implemented by recursively traversing the parent array until we hit a node who is parent to itself.

```
int find (int i)
{ if (parent [i] == i)
   { return i;
   }
  else
   {
     return find (parent [i]);
   }
  }
```

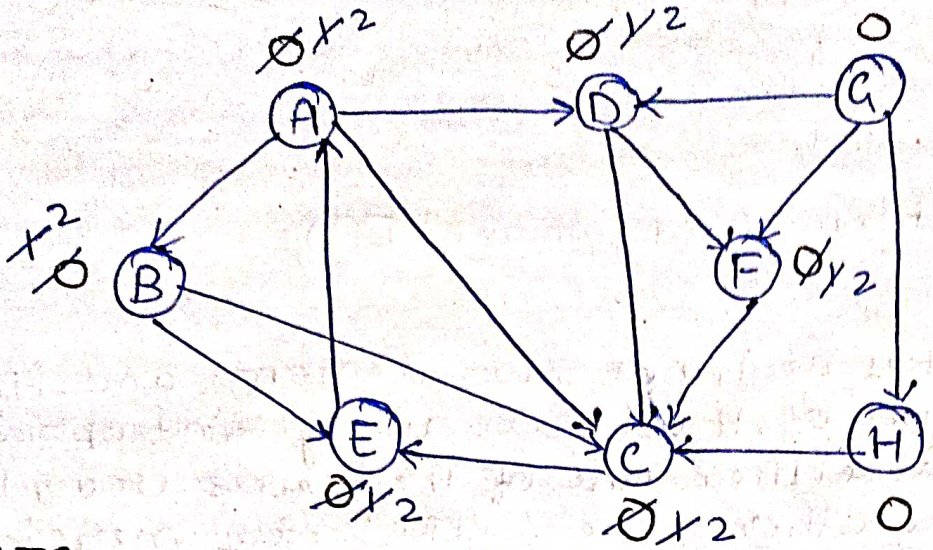
i) union: It takes as input two elements and find the representatives of their sets using find operation and finally puts either one of the trees under the root node of the other tree, efficiently merging the tree and the sets.

```
void union (int i, int j)
{ int irep = this.find [i];
  int jrep = this.find [j];
  this.parent [irep] = jrep;
}
```

ii) path compression (modification to find()): It speed up the data structure by compressing the height of the trees. It can be achieved by inserting a small caching mechanism into find operation.

```
int find (int i)
{ if (parent [i] == i)
   { return i;
   }
  else
   {
     int result = find (parent [i]);
     parent [i] = result;
     return result;
   }
}
```

Solution 6:

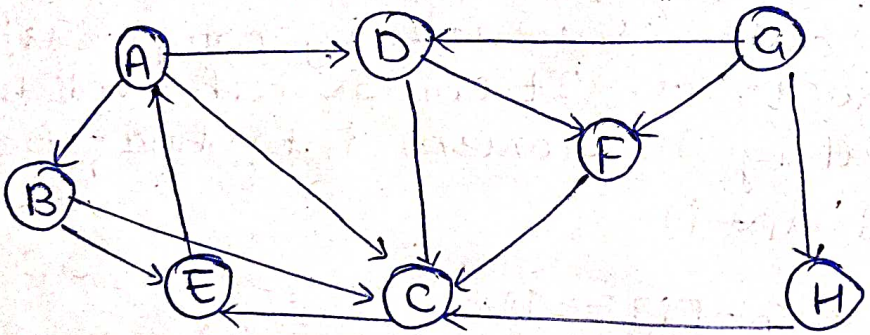


BFS Node (B) (E) (C) (A) (D) (F)

Parent – B B E A D

Unvisited nodes - G and H

Path = B → E → A → D → F



Node processed	Stack
~~B~~	B
B	C E
C	E E
E	A E
A	D E
D	F E
F	E

Path : B → C → E → A → D → F

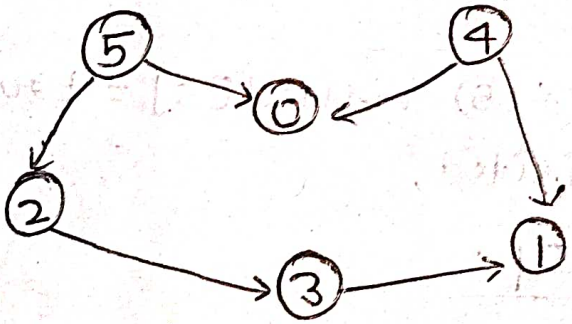
Solution 7: V = {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}, {j}

{a,b}, {a,c}, {b,c}, {b,d}, {e,f}, {e,g}, {h,i}, {j}

a,b	{a,b}, {c}, {d} {e} {f} {g} {h} {i} {j}
a,c	{a,b,c} {d} {e} {f} {g} {h} {i} {j}
b,c	{a,b,c} {d} {e} {f} {g} {h} {i} {j}
b,d	{a,b,c,d} {e} {f} {g} {h} {i} {j}
e,f	{a,b,c,d} {e,f} {g} {h} {i} {j}
e,g	{a,b,c,d} {e,f,g} {h} {i} {j}
h,i	{a,b,c,d} {e,f,g} {h,i} {j}

Number of connected components = 3

Solution 8: topological sort



Adjacent list

0 →
1 →
2 → 3
3 → 1
4 → 0,1
5 → 2,0

Visited

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

stack empty

Topological sorting

List is empty no more recursion call.

stack: | 0 |

step 2: Topological sort(1) visited[1] = true

list is empty no more recursion call

stack | 0 | 1 |

step 3: Topological sort(2) visited[2] = true

↓

Topological sort(3) visited[3] = true

1 is already visited, no more recursion call

stack, | 0 | 1 | 3 | 2 |

step 4: Topological sort(4) visited[4] = true

0,1, are already visited, no more recursion call

stack | 0 | 1 | 3 | 2 | 4 |

step 5: Topological sort(5) visited[5] = true

2,0 are already visited,

stack | 0 | 1 | 3 | 2 | 4 | 5 |

step 6: Print all elements from top to bottom;

5, 4, 2, 3, 1, 0

solution 9: We can use heaps to implement priority queue. It will take O(log N) time to insert and delete each element in the priority queue. Based on heap structure, priority queue has also two types max priority and min priority queue.

Some algorithms where we need to use priority queue-

) Dijkastra's shortest path algorithm using priority queue. When the graph is sorted in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkastra's algorithm.

) Prim's algorithm: It is used to implement Prim's algorithm to store key of nodes and extract minimum key node at every step.

3) Data compression: It is used in Huffmann code which is used to compress data.

Solution 10:

Min heap	Max Heap
In min heap key present at the root must be less than or equal to among the keys present at all of its children.	→ In max heap key present at the root node must be greater than or equal to among the keys present at all of its children.
the minimum key element present at the root.	→ the maximum key element present at the root.
uses the ascending priority.	→ uses descending priority.
in a construction of min-heap, the smallest element has priority.	→ in the construction the largest element has priority.
the smallest element is the first to be popped from heap.	→ the largest element is the first to be popped from the heap.