# 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 closer to source. | → DFS is better when target is far from source |
| → As BFS consider all neighbour so it is not suitable for decision tree used in puzzle game | → DFS is more suitable for decision tree. As we need to traverse one decision, we need to transverse further to argument the decision. |
| → BF is slower than DFS | → DFS is faster than BFS |
| → TC of BFS = O(V+E) where V is vertices & E is edges | → TC of DFS is also O(V+E) where V is vertices & E is edges |

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 delete cycle in a graph using DFS.
→ Using DFS we can find path between two given vertices u & v.
→ We can perform topological sorting which is used to scheduling jobs from given dependencies among jobs.
→ Using DFS, we can find strongly connected components of a graph.

Application of BFS

→ BFs may also used for detecting cycles in a graph.
→ finding shortest path & minimal spanning trees in unweighted graph.
→ finding a route through GPS navigation system with minimum no. of crossing.
→ In networking finding a route for packet transmission.
→ In garbage collection BFs is used for copying garbage.

Sol 2. BFs (Breadth first search) uses queue data structure for finding the shortest path. A queue (FIFO - first in first out) data structure is used by BFS. You mark any node in the graph as root and start traversing the data from it. BFS traverses all the nodes in the graph and keeps dropping them as completed. BFS visits an adjacent unvisited node, marks it is done & inserts it into queue

DFs (Depth first search) uses stack data structure. DFs algorithm traverse a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

Sol 3. • Sparse graph : A graph in which the number of edges is much less than the possible number of edges.

• Dense graph : A dense graph is 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. Alternatively, if the graph is dense, we should store it as a adjacency matrix.

Sol 4. The existence of a cycle in directed & undirected graph can be determined by whether depth first search (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 a part of cycle

Detect cycle in a directed graph:

→ 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 that is from a node to itself are 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 tree by checking back edge. To detect a back edge, keep track of vertices currently in the recursion stack of function for DFS traversal. If a vertex is reached that is already in the recursion stack, then there is a cycle in the tree. Use recstack[] array to keep track of vertices in recursion stack.

Detect cycle in an 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. A back edge is an edge that is joining a node to itself as one of its ancestors in the tree produced

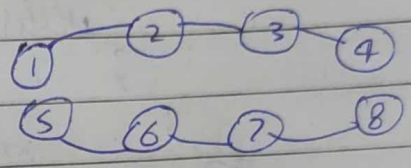
by DFS. To find the back edge to any of its ancestors in the tree produced keep a visited array, if there is a back edge to any visited node then there is a loop & return true.

Q5 Disjoint set data structure:

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

→ The disjoint set can be defined as the subsets where there is no common elements b/w the two sets

Eg $S_1 = \{1, 2, 3, 4\}$
$S_2 = \{5, 6, 7, 8\}$




operation performed:

(i) find: Can be implemented by recursively traverse 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]);
        }
}
```

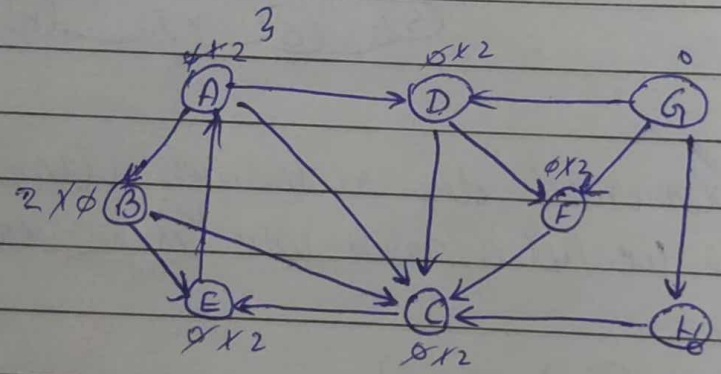
(ii) Union: It take as input, two elements and finds the representation of their sets using the find operation & finally puts either one of the tree under root node of other tree, effectively merge the tree & set

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

(iii) Path compression : It speeds up the data structure by compressing the height of the tree. 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;
}
```

Sol 6.

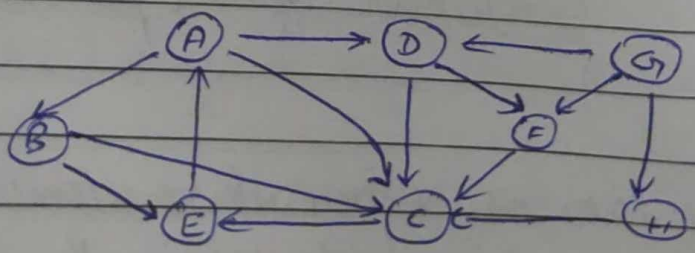



Bfs :- Node B E C A D E

Parent - B B E A D

unvisited node : G & H Path : B → E → A → D → F

DFS :




Node processed B B C E A D F

Stack B CE EE AE DE FE E

Path : B → C → E → A → D → F

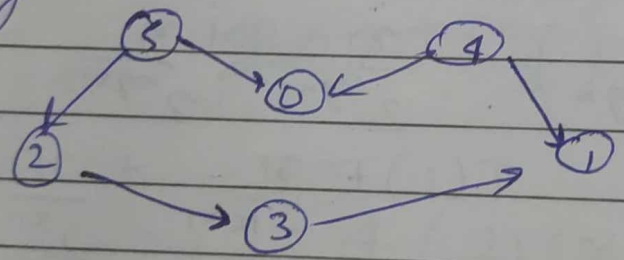
Sol 7. $V = \{a\}\{b\}\{c\}\{d\}\{e\}\{f\}\{g\}\{h\}\{i\}\{j\}$

$E = \{a,b\}, \{a,c\}, \{b,c\}, \{b,d\}, \{e,f\}, \{e,g\}, \{h,i\}$

| | |
|---|---|
| (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,d) | {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

Ans 8. topological sort :-




Adjacent list 0 →

1 →

2 → 3

3 → 1

4 → 0, 1

5 → 2, 0

visited :-

| false | false | false | false | false | false |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

stack (empty)

Step 1 :- topological sort (0), visited [0] = true
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
'5', '4' 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. No more recursion call
stack [0|1|3|2|4|5]

Step 6: Print all elements of stack from top to bottom
5, 4, 2, 3, 1, 0

Sol 9. We can use heap to implement the 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 algorithm where we need to use priority queue are:

(i) Dijkstra's: shortest path algorithm using priority queue: when the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract

minimum efficiently when implementing Dijkstra's algorithm.

(ii) Prism algorithm: It is used to implement Prim's algorithm to store keys of nodes & extract minimum key node at every step.

(iii) Data compression: It is used in Huffman's code which is used to compress data.

Sol 10.

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