Tutorial Sheet--5

Date: / /

1-

BFS	DFS
• BFS stands for Breadth First Search.	• DFS stands for Depth First-Search.
• It uses queue to find the shortest path.	• It uses stack to find the shortest path.
• It is better when target is closer to source.	• It is better when target is far from source.
• BFS is slower than DFS.	• DFS is faster than BFS.
• TC of BFS = O(V+E) where V is vertices and E stands for Edges.	• TC of DFS = O(V+E) where V is vertices and E is Edges.
• As BFS considers all neighbours so it is not-suitable for decision tree used in puzzle games.	• DFS is more suitable for decision tree. As with 1 decision we need to traverse further to argument-the decision. If we reach the conclusion, we won.

Applications of DFS:

1- If we perform DFS on unweighted graph, then it will create minimum spanning tree for all pair shortest path tree.

2- We can detect cycles in a graph using DFS. If we get one back-edge during BFS, then there must be one cycle.

3- Using DFS we can find path between two given vertices u & v.

4- We can perform topological sorting used for scheduling jobs from given dependencies among jobs.

5- Using DFS, we can find strongly connected components of a graph. If there is a path from each vertex to every other vertex, i.e., strongly connected.

Applications of BFS:

1- Like DFS, BFS may also be used for detecting cycles in a graph.

2- Finding shortest-path and minimal spanning tree in unweighted graph.

3- Finding a route through GPS navigation system with minimum no. of crossings.

4- In networking finding a route for packet transmission.

5- In building the index by search engine crawlers.

6- In peer-to peer networking, BFS is to find neighbouring nodes.

7- In garbage collection BFS is used for copying garbage.

2-

- BFS (Breadth first search) uses queue data structure for finding the shortest path
- DFS (Depth first search) uses stack data structure.
- 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 as done, & inserts it into a queue.
- DFS algorithm traverses 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.

3- Sparse Graph : A graph in which the number of edges are much less than the possible no. of edges. In this we should store it as a list of edges.

Dense Graph : A graph in which the no. of edges is close to the maximal number of edges. In this we should store it as adjacency matrix.

4- The existence of a cycle in directed and undirected graphs 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 are part of cycles.

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 (self-loop) 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.

Date: / /

To detect a back edge, keep track of vertices currently in the recursion stack of function for DFS traversal. If a vertex is reached i.e., already in the recursion stack, then there is a cycle in the tree. The edge that connects 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 an undirected graph

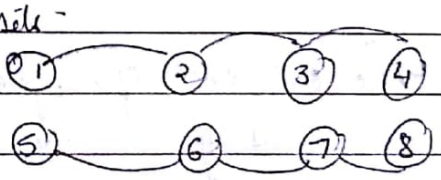
- Run a DFS from every unvisited node. DFS can be used to detect a cycle in a graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back-edge is an edge that is joining a node to itself (self-loop) or one of its ancestor in the tree produced by DFS.
- To find the back-edge to any of its ancestor keep a visit array. And if there is a back edge to any visited node then there is a loop and return true.

5- 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 element b/w the two sets.

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

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



Operations performed:

i) 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]);
}
```

Date: / /

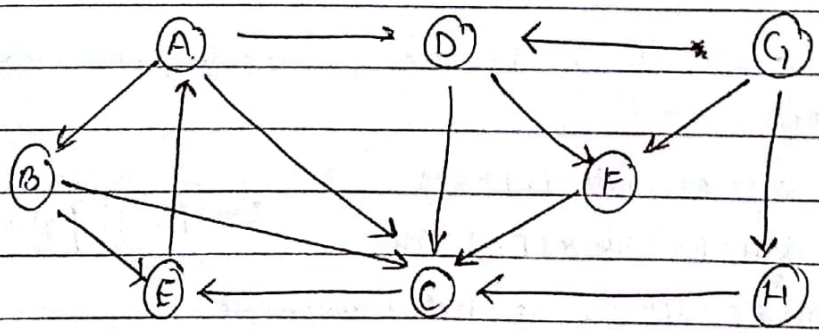
ii) Union: It takes, as input, two elements and finds the representatives of their sets using the find operation. And finally puts either one of the tree (representing the set) under the root node of the other tree, effectively merging the trees & the sets.

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

iii) 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 find operation.

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

6-



Bfs:

Node: B E C A D F

Parent: - B B E A D

Unvisited nodes: 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

Date: / /

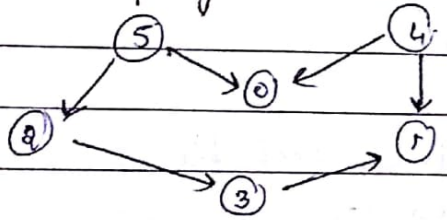
7- V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

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

No. of connected components = 3

8- Topological sort -



Adjacent list -

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

visited

0	1	2	3	4	5
false	false	false	false	false	false

Stack (empty)

S1: Topological sort-(0), visited[0] = true ; list is empty, no more recursion call.

Stack | 0 |

S2: Topological sort-(1), visited[1] = true ; list empty, no more recursion call.

Stack | 0 | 1 |

S3: Topological sort-(2), visited[2] = true

Topological sort-(3), visited[3] = true

Stack | 0 | 1 | 3 | 2 |

'1' is already visited, No more recursion call

S4: Topological sort-(4), visited[4] = true

Stack | 0 | 1 | 3 | 2 | 4 |

'0', '1' already visited, No more recursion call

S5: Topological sort-(5), visited[5] = true

Stack | 0 | 1 | 3 | 2 | 4 | 5 |

'2' & '0' already visited No more recursion call.

S6: Print all elements of stack from top to bottom.

5, 4, 2, 3, 1, 0.

Date: / /

9-

We can use heaps 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 algo's where we need to use priority queue are:

i) Dijkstra's shortest path algorithm: When the graph is stored in the form of adjacency list or matrix, priority queue can be used extract-minimum efficiently when implementing Dijkstra's algorithm.

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

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

10-

Min-heap	Max-heap
• In this the key present at the root must be less than or equal to among the keys present at all of its children.	• In this 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.	• The maximum key element present at the root.
• It uses the ascending priority.	• It uses the descending priority.
• In construction of this, the smallest element has priority.	• In construction of this, 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.