Name: Nishitha Jain
Section: H
Roll No: 39

Design And Analysis Of Algorithms (Tutorial-5)

Q1 What is difference between DFS and BFS. Please write the application of both the algorithms.

⇒

BFS

* BFS stands for Breadth First Search.
* BFS uses queue to find the shortest path.
* BFS is better when target is close to source.
  As BFS considers all neighbours so it is not suitable for decision tree used in puzzle games.
* BFS is slower than DFS.

DFS

* DFS stands for Depth First Search.
* DFS uses stack to find the shortest path.
* DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision & we reach the conclusion.
* DFS is faster than BFS

Applications Of DFS (Depth First Search)

* If we perform DFS on unweighted graph, then it will create minimum spanning tree for all pair shortest path tree.
* We can detect cycles in a 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~~ two given vertices.

Applications Of BFS (Breadth First Search)

* Like DFS, BFS may also used for detecting cycles in a 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 of search engine transist.

Q2 Which data structures are used to implement DFS and BFS and why?

⇒ BFS (Breadth First Search) uses queue data structure and 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 traverse all the nodes in the graph and keep dropping them as completed. BFS visits an adjacent unvisited node, mark it as done, and insert it into a queue.

DFS algorithm traverse a graph in depthward motion and uses a stack (LIFO → Last In First Out) to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Q3 What do you mean by sparse and dense graphs? Which representation of graph is better for sparse and dense graphs?

⇒ 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.

Q4 How can you detect a cycle in a graph using BFS and DFS?

⇒ The existence of a cycle is directed and undirected graph can be determined by whether DFS finds an edge that points to an ancestors of the current vertex. All the back edge which DFS skips over are part of cycles.

To Detect Cycle In A Directed Graph

DFS can be used to detect a cycle in a graph. DFS for a connected graph produce 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. Then for a disconnected graph, get the DFS forest as output to detect cycle Check a cycle in individuals trees 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. The edge that connects the current vertex to the vertex in the recursion stack is a back edge. Use recursion stack [] array to keep track of vertices in the recursion stack.

To 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 or one of its ancestor in the tree produced by DFS.

To find the back edge to any of its ancestor keep a visited array and if there is a back edge to any visited node then there is a loop and return true.

Qs What do you mean by disjoint set data structure? Explain three operations along with examples, which can be performed on disjoint sets.

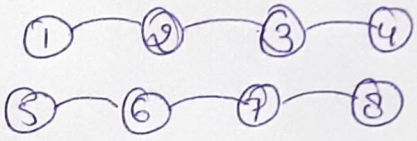
⇒ Disjoint set Data Structure

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 subset where there is no common elements b/w the two sets.

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

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



Operations Performed

① Find: can be implement by recursively traverse the parent array until we hit a node who is present to itself.

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

② Union: Takes as input, two elements and finds the representation of their sets using the find operation, and finally puts either one of the trees under the root node of the

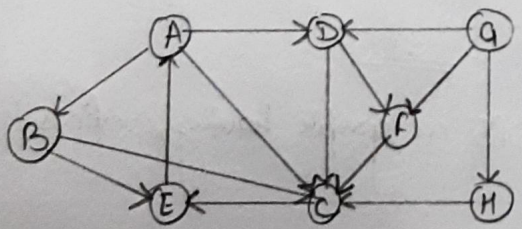
other tree, effectively merging the tree and the sets

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

(A11) Path Compression (Modification 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 k)
{
    if (Parent [i] == ?)
    {
        return ?;
    }
    else
    {
        int result = find (parent[i]);
        parent [i] = result;
        return result;
    }
}
```

Q6 Run BFS and DFS on graph shown on right side (Graph with 8 vertices).



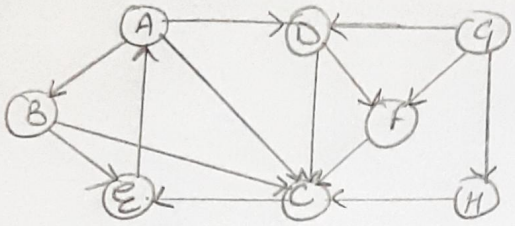
Node ⇒ (B) (E) (C) (A) (D) (F)

BFS

Parent ⇒ B B E A D

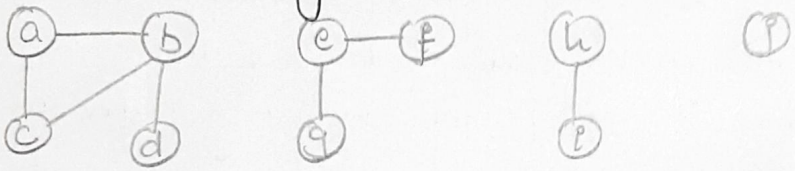
Unvisited Node ⇒ (G) and (H)

Path ⇒ B → E → A → D → F



DFS
Node Processed ⇒ B B C E A D E
Stack ⇒ B CE EE AE DE FE E
Path ⇒ B → C → E → A → D → F

Q7 Find out the number of connected components and vertices in each component using disjoint set data structure.

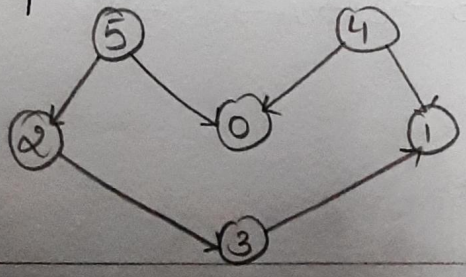


⇒ Vertices ⇒ {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
Edges ⇒ {a,b} {a,c} {b,c} {b,d} {e,f} {e,g} {h,i}

(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,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 component ⇒ 3

Q8 Apply topological sorting and DFS on graph having vertices from 0 to 5.



Visited:

0	1	2	3	4	5
f	f	f	f	f	f

Stack (empty)

Adjacency list: 0 → Null
1 → Null
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

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.
Stack → | 0 | 1 | 3 | 2 |

Step 4: Topological sort (4) visited [4] = true
'0' and '1' are already visited. No more recursion.
Stack → | 0 | 1 | 3 | 2 | 4 |

Step 5: Topological sort (5) visited [5] = true
'2' and '0' are already visited. No more recursion.
Stack → | 0 | 1 | 3 | 2 | 4 | 5 |

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

Q9. Heap data structure can be used to implement priority queue? Name few graph algorithms where you need to use priority queue and why?

⇒ 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 also has two types Max priority and Min priority.

Some algorithms where we need to use priority queue are:

(i) Dijkstra shortest path algorithm: When the path is sorted 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 key of nodes and extract minimum key node at every step.

Data Compression: It is used in Huffman's code which is used to compress data.

Q10. What is the difference between Max and Min heap?

⇒

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 root node must be greater than or equal to among the keys present all of its children.
* Use ascending priority.	* Use descending priority.