Thursday.
26/05/22.

→ Paper.

M T W T F S S
Page No.: 01
Date: 26
YOUVA

Assignment - 3

Q1 - What is difference between DFS & BFS. Please write the applications of both the algorithms.

Ans - 1 -

	DFS	BFS
i)	It stands for Depth First Search.	It stands for Breadth First Search.
ii)	Uses Stack Data Structure.	Uses Queue Data Structure for finding the shortest path.
iii)	We might traverse through more edges to reach a destination vertex from a source.	Can be used to find single source shortest path in an unweighted graph because in BFS we reach a vertex with min. no. of edges from a source vertex.
iv)	It is more suitable when there are soln.s away from source - more suitable for game or puzzle problems. We make a decision, then explore all paths.	BFS considers all neighbours first & ∴ not suitable for decision making trees. Used in games & puzzles.

through this decision & if this decision leads to win situation, we stop.

Q2- Which Data structures are used to implement BFS & DFS & why?

Ans -

i) The time complexity of BFS is $O(v+E)$ when adjacency list is used & $O(v^2)$ when adjacency matrix is used. where $v \rightarrow$ vertices, $e \rightarrow$ edges.	The time complexity of BFS is $O(v+e)$ when adjacency list is used & $O(v^2)$ when adjacency matrix is used.
ii) Siblings are visited later than children.	Siblings are visited before children
iii) DFS requires less memory.	BFS requires more memory.

Applications of BFS:

1. Crawlers in search engine.
2. GPS navigation system.
3. To find shortest path & minimum spanning tree for an unweighted graph.

4- Broad Casting
5- Peer to Peer Networking.

Application of DFS

1- Deleting cycles in graph
2 Topological Sorting
3. To check if a graph is bipartite.
4. Path finding.
5. Finding strongly connected components of a graph.

Q2- Which data structure are used to implement BFS & DFS & why?

Sol- Queue is used to implement BFS.
Stack is used to implement DFS.

BFS: It traverses a graph in a breadth-ward motion & uses a queue to remember to get the next vertex to start a search when a dead end occurs at any iteration.

DFS: It traverses a graph in a depth-ward motion & uses a stack to remember to get the next vertex to start a search when dead end occurs in any iteration.

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

1- Dense Graph
If the no. of edges is close to the maximum no. of edges in a graph then that graph is a dense graph.
In a dense graph, every pair of vertices is connected by one edge.

Sparse Graph
The sparse graph is completely the opposite. If a graph has only few edges (the no. of edges is close to the max. no. of edges), then it is a sparse graph.
There is no distinction b/w sparse graph & dense graph.

* For dense graph, adjacency matrices are the most suitable graph representation because in big-O terms they don't take up more space.

* For sparse graph, adjacency list are good & generally preferred.

4- How can you detect a cycle in a graph using BFS & DFS?

Sol- BFS

1. No. of incoming edges for each of the vertex present in graph & initialize the count of visited nodes as 0.

2. Pick all the vertices with in-degree as 0 & add them into a queue (Enqueue Operation.)

3. Remove a vertex from the Queue & then
→ Increment count of visited nodes by 1.
→ Decrease in degree by 1 for all its neighbouring nodes.
→ If in-degree of a neighbouring node is reduced to zero then add it to the queue.

4. Repeat Step 3 until Queue is Empty.

5. If the count of visited nodes is not equal to the no. of nodes then the graph has cycle, otherwise not.

```
class Graph
{
    int v;
    list <int> * adj;
public:
    Graph (int v);
```

```
    void addEdge(int u, int v);
    bool isCycle();
};
Graph::Graph(int v)
{
    this->v = v;
    adj = new list<int>[v];
}
void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v);
}
bool Graph::isCycle()
{
    vector<int> in-degree(v, 0);
    for(int u=0; u<v; u++)
    {
        for(auto v : adj[v])
            in-degree[v]++;
    }
    queue<int> q;
    for(int i=0; i<v; i++)
        if(in-degree[i]==0)
            q.push(i);
    int cnt=1;
    vector<int> top-order;
    while(!q.empty())
    {
        int v = q.front();
        q.pop();
```

```
top_order.push_back(u);
list<int>::iterator itr;
for (itr = adj[v].begin(); itr != adj[v].end(); itr++)
{
    if (--in_degree[*itr] == 0)
    {
        q.push(*itr);
        cnt++;
    }
}
}
if (cnt != V)
    return true;
else
    return false;
```

Cycle Detection using DFS:

1. Create the graph using the given no. of edges & vertices.
2. Create a recursive function that initializes the current index of vertex, visited & recursion stack.
3. Mark the current node as visited & also mark the index in recursion stack.
4. Find all the vertices which are not visited & are adjacent to the current node.

Recursively call the function for those vertices, if the recursive function returns true : return false true :

5. If the adjacent vertices are already marked in the recursion stack then return true.

6. Create a wrapper class, that calls the recursive function for all vertices & if any function returns true return true. Else if for all vertices & if any function returns false return false.

Pseudo code

```
Class Graph
{
  int v;
  list <int> * adj;
  bool isCyclicUtil (int v, bool visited[],
                     bool * rs);
  Public :
    Graph (int v)
      void addEdge (int v, int w);
      bool isCyclic();
};
Graph :: Graph (int v)
{
    this -> v = v;
```

```
    adj = new list<int>[v];
}
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}
bool Graph::isCyclicUtil(int v,
    bool visited[], bool* recStack)
{
    if (visited[v] == false)
    {
        visited[v] = true;
        recStack[v] = true;
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if (!visited[*i] && isCyclicUtil(*i, visited, recStack))
                return true;
            else if (recStack[*i])
                return true;
        }
    }
    recStack[v] = false;
    return false;
}
bool Graph::isCyclic()
{
    bool *visited = new bool[v];
    bool *recStack = new bool[v];
```

```
for(int i=0; i<v; i++)
{
    visited[i] = false;
    restack[i] = false;
}
for(int i = 0; i <v; i++)
{
  if (!visited[i] && iscyclicutil(i, visited, restack)
    return true;
  return false;
}
```

5- What do you mean by disjoint set data structure? Explain 3 operations along with examples which can be performed on disjoint sets.

A disjoint set data structure also called a union find data structure & merge - find set is a data structure that stores a collection of disjoint sets.

Equivalently, it stores a portion of a set into disjoint subsets. It provides operations for adding new sets; merging sets & finding a representative member of a set.

Operations:

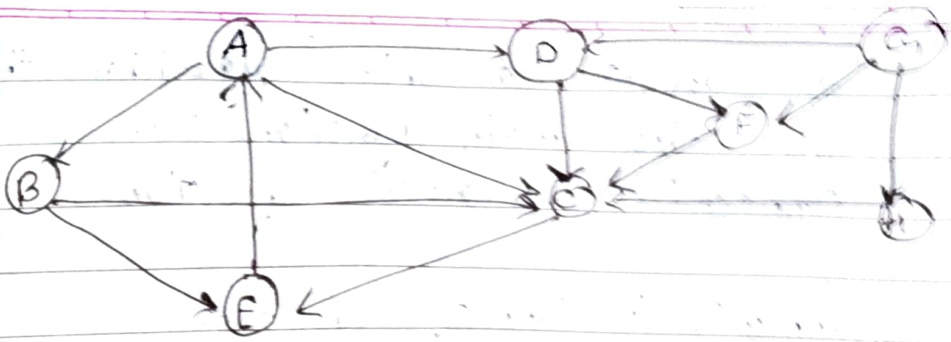
1- Making new sets: The Makeset operation adds a new element into a newset containing only the new element and the new set is added to the data structure.

2- Merging two Sets:
The operation union(x,y) replace the set containing x & set containing y with function their union.

Union first uses find to determine the root of the tree containing x & y.

3- Finding Set Representatives
The find operation follows the chain of parent pointer from a specified query code x until it reaches a new element. This root element represents the set to which x belongs & may be itself. Find returns the root element it reaches

Q6- Run BFS & DFS on graph shown below:



Let 'A' be the source node & 'F' be the goal node

1. For BFS : (queue)

	Visited
A	{A}
{B, C, D}	{A, B}
{D, C, E}	{A, B, D}
{C, E, F}	{A, B, D, C}
{E, F}	{A, B, D, C, E}
{F}	{A, B, D, C, E, F}

2. For DFS (Stack)

Visited	A	B	D	C	E	F
Stack	B	E	F	E	F	
	D	D	C	F		
	C	C				

=> { A, B, D, C, E, F }.

7. Find the no. of connected components & vertices in each component using disjoint set data structure.

Sol- In disjoint set union algorithm there are two main functions i.e., connect() & root() function

connect() : connects an edge
root() : recursively determine the topmost parent of a given edge.

For each edge {a,b}, check if a is connected to b or not, if found to be false connect them by appending their top parents.

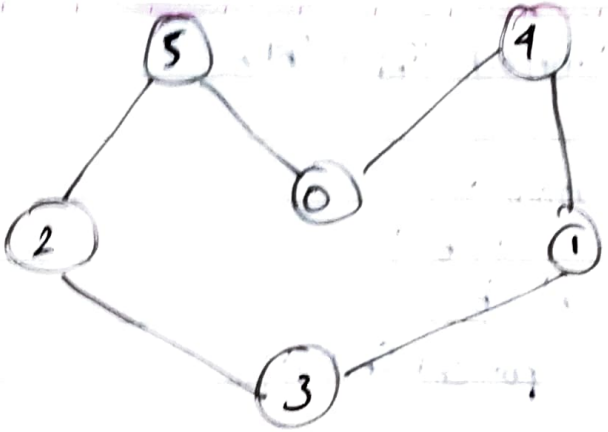
After completing the above step for every edge. print the total number of the distinct top-most parents for each vertex.

Pseudo code

```
int Parent max;
int root (int a)
{
    if (a == parent[a])
        return a;
    return parent[a] = root(parent[a]);
}
```

```
void connect (int a, int b)
{
    a = root(a);
    b = root(b);
    if (a != b)
        parent[b] = a;
}
void connectedComponents (int n)
{
    set <int> s;
    for (int i = 0; i < n; i++)
    {
        s.insert (root (parent[i]));
    }
    cout << s.size() << "\n";
}
void printAnswer(int N, vector<vector<int>>edges)
{
    for (int i = 0; i <= N; i++)
        parent[i] = i;
    for (int i = 0; i < edges.size(); i++)
        connect (edges[i][0], edges[i][1]);
    connectedComponents (N);
}
```

3- Apply topological sorting & DFS on graph having vertices from 0 to 5.



sol^n

```
Class Graph
{   int V;
    list < int >* adj;
    void topological sort util (int v, bool
                    visited [], stack < int >&

public:
    Graph ( int v).
    void add Edge (int v, int w);
    void topological sort ().
};

Graph :: Graph (int v)
{
    this -> V = v.
    adj = new list < int > [v].
}
void Graph :: add Edge (int v, i
{
    adj [v] . push_back (w).
}
```

```
void Graph :: topological sort util
        (int v, bool visited[ ], stack
        <int> & stack)
{ visited[v] = true;
  list <int> :: iterator i;
  for (i = adj[v].begin(); i != adj[v]
                          .end(); ++i)
    if (!visited[*i])
        topological sort util(*i, visited, stack);
    stack.push(v);
}

void Graph :: topological sort()
{
  Stack <int> Stack;
  bool* visited = new bool[V];
    for (int i=0; i<v; i++)
        visited [i] = false;

    for (int i = 0; i<v; i++)
        if (visited[i] == false)
            topological sort util (i, visited, sta
    while (stack.empty() == false)

    {
      cout << stack.top() << " ";
        Stack.pop();
    }
}
```

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

Ans Yes, Heap data structure can be used to implement priority queue.
Heap data structure provides an efficient implementation of priority queue.

Few graph algorithm where priority Queue is used.

-> **Dijkstra's Algo** where the graph is stored in the adjacency matrix list, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

-> **Prims Algo** to store keys of node & extract minimum key node at every step.

-> **A* search algorithm**. find the shortest path between two vertices of a weighted graph.
The priority queue is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.

Q.10 what ...
4 ma...
Mi...

Sol
1- In mi...
key pr...
root ...
less th...
among...
Present...
Its ch...
2. In mi...
minim...
is pr...
root ...
3. Min ...
ascen...
4. In th...
of ...
smalle...
has ...
5. The ...
is the ...
popped ...
heap ...

10 What is the difference between min heap & max heap.

Min Heap	Max Heap
In min heap the key present act root node must be less than or equal to among the keys present at all of its children	1- In 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.
In min heap the minimum element is present at the root.	2. In max heap the maximum element is present at the root.
Min heap uses the ascending priority	3. Max heap uses the descending priority
In the construction of min heap, the smallest element has priority	In the construction of Max heap, the largest element has priority.
5. The smallest element is the first to be popped from the heap.	The largest element is the first to be popped from the heap.

Name - Mohammad Anas Khan
Section - CST_SPL1
Roll No - 20
Student ID - 20021335