TUTORIAL-5

DESIGN & ANALYSIS OF ALGORITHMS

D. Uma Vyshnavi
CST-S PL-2
ROLL NO-37.

(1) What is difference between BFS & DFS. please write the applications of both the algorithms.

BFS

→ BFS stands for Breadth first search.

→ BFS uses queue data structure for finding the shortest path

→ BFS can be used to find single source shortest path in an unweighted graph.

→ BFS is more suitable for searching vertices which are closer to the given source.

→ The time complexity of BFS is O(V+E) when adjacency list is used and O(v^2) when Adjacency matrix is used where v stands for vertices & E stands for Edges

→ siblings are visited before the children

→ BFS requires more memory

DFS

→ DFS stands for Depth first search.

→ DFS uses stack data structure.

→ In DFS, we might traverse through more edges to reach a destination vertex from a source.

→ DFS is more suitable when there are solutions away from source.

→ Time complexity of DFS is also O(V+E) when adjacency list is used and O(v^2) when Adjacency matrix is used, where v stands for vertices & E stands for edges.

→ Children are visited before the siblings.

→ DFS requires less memory.

Applications of BFS.

- Crawlers in search engines
- GPS Navigation systems
- Find the shortest Path & minimum spanning tree for an unweighted graph
- Broadcasting
- Peer to peer Networking.

Applications of DFS:-

- Detecting cycle in a graph.
- Topological sorting
- To test if a graph is bipartite
- Path finding
- Finding strongly connected component of a graph.

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

Ans → Queue is used to implement BFS
→ stack is used to implement DFS.

BFS — Breadth first search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

DFS:- Depth first search (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.

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

Ans: Dense Graph:-

→ If the number of the edges is close to the maximum number 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 a few edges (the no: of edges is close to the maximum number of edges), then it is a sparse graph.

→ There is no distinction between the sparse and the dense graph.

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

★ For a sparse graph, adjacency list are good and generally preferred.

Q4) How can you detect a cycle in a graph u
BFS & DFS?

Ans:- BFS

1) Number of incoming edges for each of the vertex present
graph and initialize the count of visited nodes as 0.

2) pick all the vertices with in-degree as 0 and add them
a Queue (Enqueue operation)

3) Remove a vertex from the Queue (Dequeue Operation) and
→ 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 ze
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 no
in the graph has cycle, Otherwise not.

```
class Graph {
    int V;
    list <int> *adj;
public:
    Graph (int V);
    void addEdge (int v, int v);
    bool isCycle ();
};
Graph:: Graph (int v)
{
    this → v = v;
    adj = new <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[u])
        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 u=q.front();
   q.pop();
   top_order.push_back(u);
   list<int>::iterator itr;
   for(itr=adj[u].begin(); itr!=adj[u].end(); itr++)
   { if(--in_degree[*itr]==0)
   {
     q.push(*itr);
      int++;
    }}}
if(cnt!=v)
        return true;
else    return false;
```

Cycle Detection using DFS:-

1. Create the graph using the given number of edges and vertices.
2. Create a recursive function that initializes the current index of a vertex, visited and recursion stack.
3. Mark the current node as visited and also mark the index in recursion stack.
4. Find all the vertices which are not visited and are adjacent to the current node.
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 the vertices and if any function returns true return true, else if for all vertices and if any function returns false return false.

Pseudocode

```
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)
{
```

Q5) What do you mean by disjoint sets data structure? Explain 3 Operations along with examples, which can be performed on disjoint sets.

Ans:- → A disjoint-set data structure, also called a union-find data structure or merge-find set, is a data structure that stores a collection of disjoint sets.

→ Equivalently, it stores a partition of a set into disjoint subsets. It provides operations for adding new sets, merging sets and finding a representative member of a set.

OPERATIONS

1) Making new sets :- The Make set Operation adds a new element into a new set containing only the new element, and the new set is added to the data structure.

2) Merging two sets :-

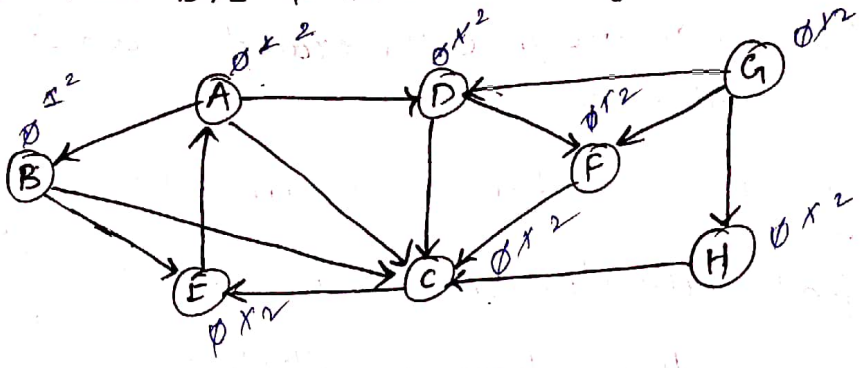
The Operation Union(x, y) replaces the set containing x and set containing y with their union.

Union first uses Find to determine the root of the trees containing x & y.

3) Finding set representatives :-

The Find operation follows the chain of parent pointers from a specified query node x until it reaches a root element. This root element represents the set to which x belongs and may be x itself. Find returns the root element it reaches.

Q6) Run BFS & DFS on the given graph.



BFS

Nodes	A	B	D	E	E	F	G	H
Parent.		A	A	A	B	D	D	G

DFS (stack)

visited	A	B	D	C	E	F
stack	B	E	F	E	F	
	D	D	E	F		
	E	C				

→ {A, B, D, C, E, F}.

BFS (Queue)

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

(7) Find the number of Connected components and vertices in each component using disjoint set data structure.

A: In Disjoint set union algorithm there are two main functions, i.e. connect() and root() function.

connect(): connects an edge.

Root(): Recursively determine the topmost parent of a given edge.

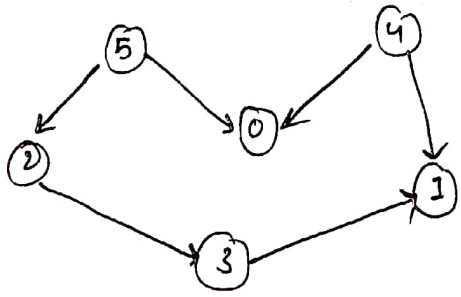
Pseudocode

```
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 connected Components (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<edge.size(); i++) {
  Connect(edges[i][0], edge[i][1]);
  }
 Connected component(N);
}
```

---

(8) Apply Topological sorting and DFS on graph having vertices from 0 to 5.




```
class Graph;
 int V;
 list<int> *adj;
  void topological sort util(int v, bool visited[], stack
                                        <int> & stack);
Public:
   Graph(int v);
   void addEdge(int v, int w);
   void topological sort();
};
```

```
Graph::Graph(int v)
{
  this->V=v;
  adj=new list<int>[v];
}
void Graph::addEdge(int v, int w)
{
  adj[v].push_back(w);
}
void Graph::topologicalSortUtil(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])
      topologicalSortUtil(*i, visited, stack);
  stack.push(v);
}
void Graph::topologicalSort()
{
  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)
      topologicalSortUtil(i, visited, stack);
  while(stack.empty()==false)
  {
    cout<<stack.top()<<" ";
    stack.pop();
  }
}
```

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

:→ Yes, Heap data structure can be used to implement priority queue.

* Heap data structures provide an efficient implementation of priority queues.

Few Graph algorithms where priority queue is used

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

* Prim's Algorithm:- To store keys of nodes and extract minimum key node at every step.

A* Search algorithm:- A* search algorithm finds 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 is the difference between Max and Min heap?

Sol:

Min heap	Max heap
1) In a Min-heap the key present at the root node must be less than & equal to among the keys present at all of its children.	1) In a Max-heap the key present at the root node must be greater than & equal to among the keys present at all of its children.
2) In a min-Heaps the minimum key element present at the root.	2) In a max-heap the maximum key element present at the root.
3) A Min-heap uses the ascending priority.	3) A max-heap uses the descending priority.
4) In the construction of a Min-heap, the smallest element has priority.	4) In the construction of a Max-heap, the largest element has priority.
5) The smallest element is the first to be popped from the heap	5) The largest element is the first to be popped from the heap.

D. Uma Vyshnavi

CST-SPL-2

Roll no:- 37.