DAA - TUTORIAL - 5

Q1) Difference b/w DFS and BFS

BFS

(1) BFS stands for breadth first search

(2) BFS uses queue

(3) BFS can be used to find single source shortest path in an unweighted graph.

(4) BFS is more suitable for searching vertices which are closer to the given source

(5) The time complexity of BFS is $O(V+E)$ when adjacency list is used and $O(V^2)$ when adjacency matrix is used.

(6) Siblings are visited before children.

(7) BFS requires more memory

Applications

(1) Crawlers in search engine

(2) GPS navigation system

(3) Find shortest path & minimum spanning tree for an unweighted graph

(4) Broad casting

(5) peer to peer networking

DFS

(1) DFS stands for Depth first search

(2) DFS uses stack

(3) In DFS we might traverse through more edges to reach destination vertex.

(4) DFS is more suitable when the solutions are away from source

(5) time complexity of DFS is also $O(V+E)$ when adjacency list is used and $O(V^2)$ when adjacency matrix is used.

(6) children are visited before siblings.

(7) DFS requires less memory.

Applications

(1) Deleting cycles in graph

(2) topological sorting

(3) to check if a graph is bipartite

(4) path finding

(5) Finding strongly connected components in a graph

Q2) Which data structure are used to implement BFS + DFS and why?

Sol=> Queue is used to implement BFS
Stack is used to implement DFS

BFS => BFS algorithm traverse 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 => DFS algorithm traverse a graph in a depth ward 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 space and dense graphs? Which representation of graph is better for separated dense graphs?

Sol => Dense graph

If the number of 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 is of vertices is connected by one edge.

Sparse graph

The sparse graph is completely the opposite.
If the no of edges is close to the minimum number of edges than that graph is a Sparse graph.

There is no distinction between sparse graph & dense graph.

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

(*) For sparse graph, adjacency list are good and generally preffered.

Q4) How can you detect cycles in a graph using BFS and DFS

BFS

(1) Compute in degree (number of incoming edges) for each of the vertex present in the graph and initialize the count of visited node as 0.

(2) Pick all the vertices with in degree as 0 and add them into a queue (Enqueue operation)

(3) Remove a vertex from the queue (Dequeue operation) and then
   → Increment count of visited nodes by 1
   → Decrease in degree by 1 for all its neighbouring nodes
   → If in-degree of a neighbouring nodes 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 number of nodes in the graph, then the graph has cycles, 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[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);
             cnt++;
          }
       }
  }
 if cnt != V
   return true;
  else
  return false;
}
```

DFS

1) Create a graph using the given number of edges and vertices.
2) Create a recursive function that initializes the current index or 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. Recursively call the function for those vertices. If the recursive function returns true, return 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 the vertices and if any function returns true, return true. Else if for all vertices and 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;
      recStack[i] = false;
   }
```

```
for (int i = 0; i < V; i++)
    if ( ! visited [i] && is cyclic util (i, visited, recstack))
        return true;
    return false;
}
```

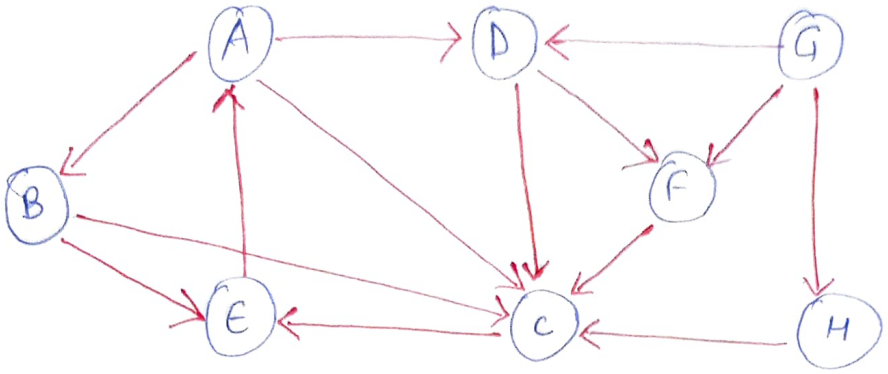
Ques 5 => What do you mean by disjoint set data structure? Explain 3 operations along with examples which can be performed on disjoint set.

Sol => 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 portion of a set into disjoint subsets. It provides operations for adding new sets, merging sets and finding a representative number of sets.

Operations =>

1) Making new sets - The Makeset 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) replace the set containing x and set containing y with their union.
Union first uses find to determine the root of the tree containing x & y.
3) Finding set representative - The find operation follows the chain of parent pointer from a specified query node x until it reaches a new element. This root element represents the set to which x belongs and may be x itself. Find returns the root element it reaches.

Ques 6) Run BFS & DFS on graph below



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

1) For BFS – {queue}

	visited
A	
{B, C, D}	{A}
{D, ~~C~~, E}	{A, B}
~~{C, E, F}~~ ~~{D, E}~~	{A, B, D}
{E, F}	{A, B, D, ~~C~~}
{F}	{A, B, D, ~~C~~, E}
	{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~~	E	F		
	C	E				

=) {A, B, D, C, E, F}

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

Sol=> In disjoint set union algorithm there are two main functions, ie connect() and 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.

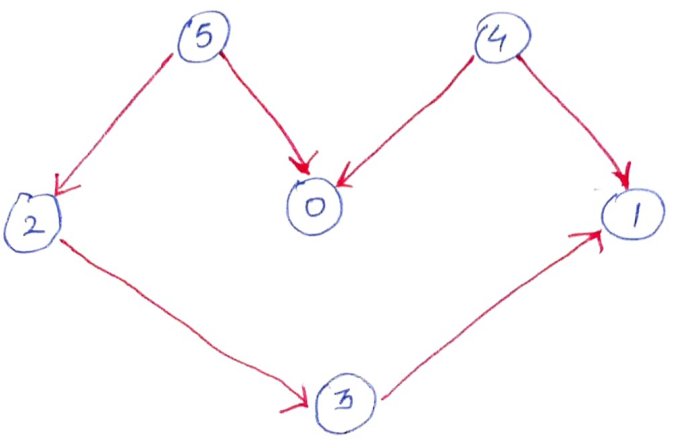
After completing the above step for every edge, print the total number of distinct top-most parents for each vertex.

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 connectcomponents(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);
}
                                    //output = 3
```

Ques 8) Apply topological sorting and DFS on graph having vertices from 0 to 5



```
class graph
{
    int V;
    list < int > * adj;
    void topologicalSortUtil ( 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 :: add Edge (int v, int w)
   {
     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 ] )
       topologicalSort 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 sortutil (i, visited, stack);
  while (stack.empty () == false)
  {
    cout<< stack.top ()<< " ";
    stack. pop ();
  }
}
```

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

Sol => Yes, Heap data structure can be used to implement priority queue.
Heap data structure provides an efficient implementation of priority queue.
Few graph algorithms where priority queue is used -

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

→ Prims Algorithm to store key of node & 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.

Ques 10) What is the difference b/w Min Heap & Max Heap ?

Min Heap	Max Heap
1) In min heap the key present at 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
2) In min heap the minimum element is present at the root	(2) In max heap the maximum element is present at the root.
3) Min heap uses ascending priority	(3) Max heap uses descending priority
4) In construction of min heap smallest element has priority	(4) In construction of max heap 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