26/5/22

DAA TUTORIAL-5

1) What is difference between DFS and BFS. Please write the applications of both the algorithms.

BFS	DFS
1. BFS stands for Breadth first search	1. DFS stands for Depth for search.
2. BFS uses queue data structure for finding shortest path	2. DFS uses stack data structure
3. BF's can be used to find single source shortest path in an unweighted graph	3. In DFS we might traverse through more edges to reach destination vertex.
4. BF's is more suitable for searching vertices which are closer to the given source	4. DF's is more suitable when there are solutions away from source
5. The time complexity of BFS is $O(V+E)$ when adjacency list is used & $O(V^{\wedge}2)$ when adjacency matrix is used, where V stands for vertex & E stands for Edge	5. Time complexity of DFS is also $O(V+E)$ when adjacency list is used and $O(V^{\wedge}2)$ when adjacency matrix is used.
6. Siblings are visited before children	6. Children are visited before siblings.
7. BFS requires more memory	7. DF's requires less memory.

Applications of BFS

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.

Applications 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.

2) 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:- Breadth first search (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:- Depth first search (DFS) algorithm traverse a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when dead end occur in any iteration.

3) 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 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 maximum number of edges), then it is a sparse graph

There is no distinction between 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 and generally preferred.

4) How can you detect a cycle in a graph using BFS & DFS.

Sol:- BFS

1. Number of incomming edges for each of the vertex present in graph and initialize the count of visited nodes 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 neighboring 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 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:: is cycle ( )
{
  vector <int> in_degree (V,0);
  for (int v=0; v<V; v++)
      {
        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 number of edges and vertices
2. Create a recursive function that initializes the current index of 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 recursible 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)
{
  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] && isCyclicUtil(i, visited, recstack))
            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.

Sol:- 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 partion of a set into disjoint subsets. (It provides a partion 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 newset containing only the new element and the new set is added to the data structure.

2. Merging two sets :-

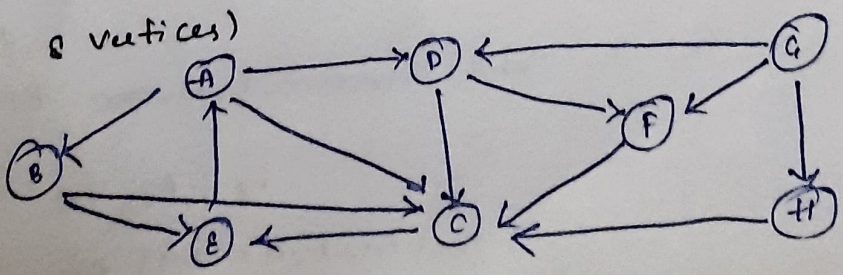
The operation union(x,y) replace the set containing x and set containing y with their union.

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 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.

6) Run BFS & DFS on graph shown below (Graph with 8 vertices)



Sol:- Let 'A' be the source Node & 'F' be the goal node

1. For 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}

2. For DFS (stack.

visited	A	B	D	C	E	F
Stack	~~B~~ ~~D~~ ~~C~~	E ~~D~~ ~~C~~	F ~~C~~	~~E~~ F.	F	

⇒ {A, B, D, C, E, F}.

④ 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, i.e. connect() and root() function

connect(): connects an edge.

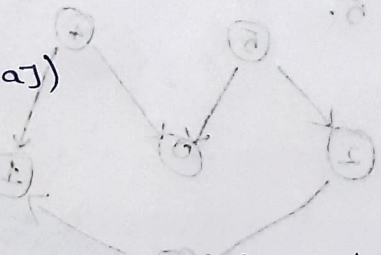
Root(): Recursively determine the topmost parent of a given edge.

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 the distinct top-most parents for each vertex

Pseudo code

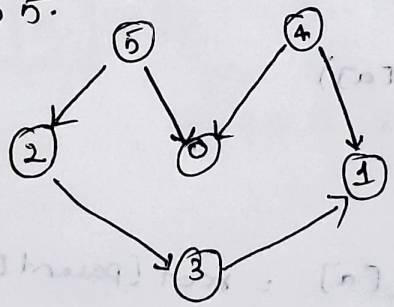
```
int Parent max;
int root (int a).
{
  if (a == parent[a])
    { return a;
    }
  return parent [a] = root (parent [a]);
}
Void connect (int a, int b)
{
  a = root (a);
  b = root (b);
  if (a != b) {
    Parent [b] = a;
  }
}
Void connected Components (int n)
{
  Set <int> s;
  for (int i = 0; i < 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], edges[i][1]);
  }
  connected components(N);          // output-3.
}
```

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

Sol:-




```
Class Graph {
  int V;
  list<int>* adj;
  void topologicalsortutil(int v, bool visited[], stack
                                  <int> & stack);
Public:
  Graph(int V);
  voidaddEdge(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 & why?

Sol:- Yes, Heapdatastructure 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 & list, Priority Queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm

→ Prims Algorithm To store keys of node & extract minimum key node at every step.

⇒ A* search algorithm 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.

10) what is the difference between min heap & Max heap.

Sol

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 root.
3. min heap uses the ascending priority	3. Max heap uses the descending priority

Min heap	Max heap
4. In the construction of Min heap, the smallest element has priority	4. In the construction of 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.