SDE Sheet

Graphs.

Date : Page :

→ data structure with node and edges

Types of graphs

→ undirected : edges have no direction

→ directed : edges have direction

undirected graph: degree → no. of adjacent nodes

total degree = $2 \times E$

Directed graph: Indegree → no. of incoming edges

Outdegree → no. of outgoing edges

path: sequence of nodes/vertex in which all vertex appears only once

Graph representation in C++

Input

n m

[m lines u v]

n = no. of vertex

m = no. of edges.

u = start of edge

v = end of edge

① Adjacency matrix.

$(n+1) \times (n+1)$ matrix.

if there is edge between u and v.

adj[u][v] = 1 and adj[v][u] = 1 ∵ in case of undirected.

only for small values

SC → $O(N^2)$

② Adjacency list.

vector<int> adj[6] → array of vectors.

(1-2)

(1-4)

(1-3)

(3-4)

0	
1	{2, 4, 3}
2	{1}
3	{1, 4}
4	{1, 3}

SC → $(N+2E)$

for weighted graph, use pair

Date: Page:

Connected components in a graph.

→ if a graph has disconnected, each part is called component

→ connected graph has only 1 component.

Q3. Breadth First search Traversal technique

→ generate adjacency list

→ for any node, we visit all others adjacent nodes of it

→ traverse adjacent nodes first and then move to next nodes

Start from 1st nodes. as well as maintain visited bool array.

→ use queue

TC → O(N) + O(N)

SC → O(N)

Q2. Depth First search Traversal technique.

→ go to max depth untill all nodes are not visited, then come back

→ traverse adjacent of adjacent first.

→ use of stack / recursion.

→ use of visited matrix too.

TC → O(N) + O(N)

SC → O(N).

Q4. Cycle detection in undirected graph.

→ create visited array along with adjacency list.

→ carry previous node too so as to ensure it shouldn't take prev node in consideration.

→ if there are no more adjacent nodes return false, but if encountered already visited node, return true

TC → O(N) SC → O(N)

check for cycle BFS.

→ create visiting array assigning all unvisited

→ use of queue so that we could save nodes ~~and their parent~~

→ visit a node, pop it out of queue and mark it as visited.

→ if you reach visited node, return <node, par> null else return false

TC: $O(N+E)$ SC: $O(N+E) + O(N) + O(N)$

use any of the opt for solution

Q5. Bipartitive graph.

→ a graph that can be colored with exactly 2 colors and no 2 adj nodes must have same color.

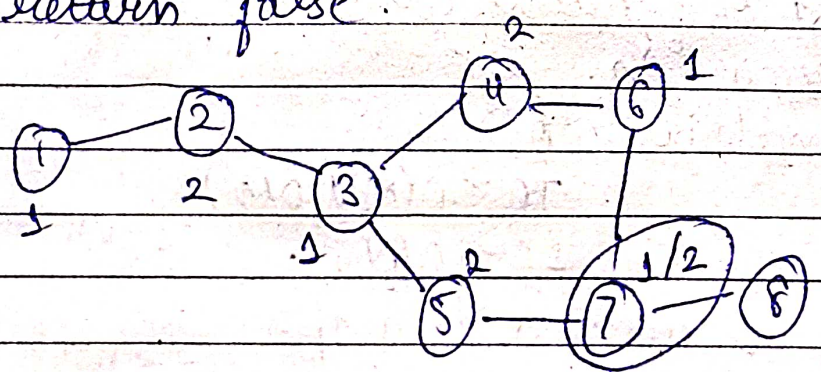
→ cycle length should not have ~~even~~ odd length cycle

BFS algo.

→ make use of array as well as queue.

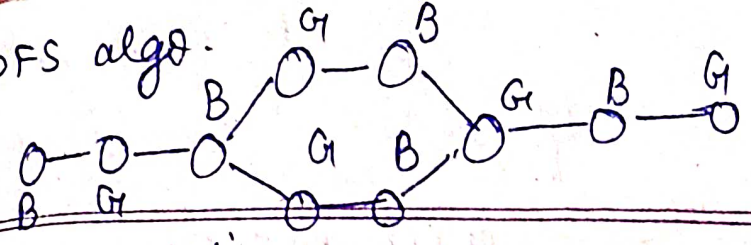
→ at each step check whether ~~node~~ is colored or not / adjacent nodes have opposite color or not.

→ if you found any node violating rule, return false.



exception thus this graph is non bipartite.

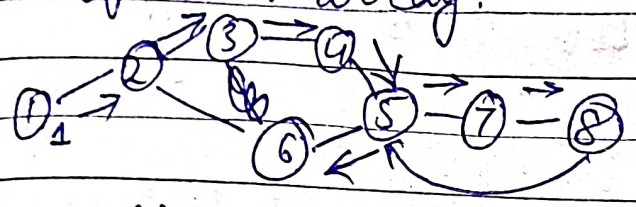
DFS algo.



Date : Page :

→ no 2 adjacent nodes must not have same color,
→ contradiction in color filling helps us find whether graph is bipartite or not.
observations : check for cycle size. : graph with no odd length cycle is always bipartite.

use of color array.



	1	2	1	2	1	2	2	1
0	1	2	3	4	5	6	7	8

DFS(1) → check for adjacent nodes
DFS(2) color the node and call its DFS
DFS(3)
DFS(4)
DFS(5)
DFS(7)
DFS(8)

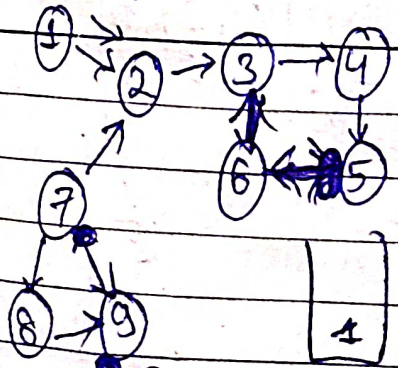
→ DFS(6)
DFS(2) → color == DFS(6) ⇒ color
Thus this graph is not bipartite.

True
True

Ans false (this graph is not bipartite)

TC : O(N+E) SC : O(N+E) + O(N) + O(N)
color array auxiliary space

Q6. cycle detection using DFS.



1	2	1	1	2	1	1	1	1
1	2	3	4	5	6	7	8	9

→ unsimilar to that of undirected one

DFS(1)
↓
DFS(2)
↓
DFS(3) ⟷ DFS(6)
↓ ↓
DFS(4) DFS(5)
↓
DFS(5)

undirected doesn't work

but we can't call it as cycle

```
for (i=1 to n){
    if(!vis(i))
    {
      if(cycleCheck(i)
        return T.;
    }
}
```

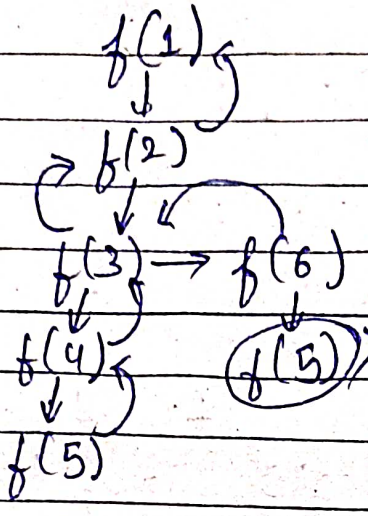
create a visited array

vis

1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1

dfs vis

1	2	3	4	5	6	7	8	9
0	0	0	0	0	0			



→ when you go back and recursive call is done, then mark dfs call back.

dfs vis(5) = 0

TC : O(N+E)

SC → O(2N)

Q7. Topological sort.

→ linear ordering of vertices

→ for every edge u → v, u appears before v

→ It is only possible for Directed acyclic graph

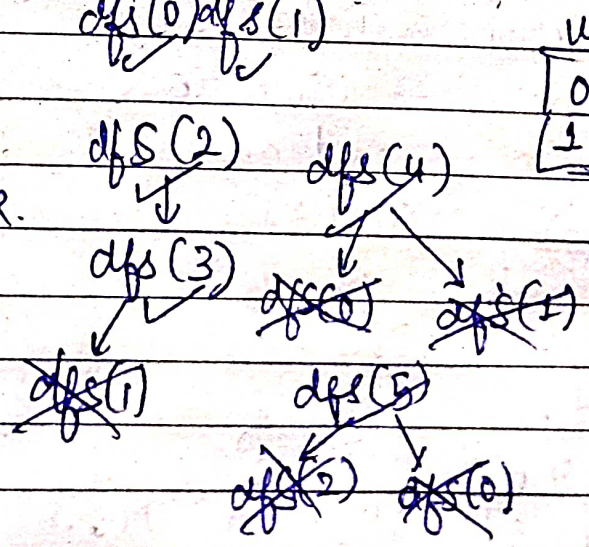
→ DAG : Directed + cycle in graph.

DFS algorithm.

```
for (0 → n)
    if(!vis[i])
        dfs(i)
```

→ use of stack and visited array

→ whenever DFS is over, push element to stack.



vis

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

5
4
2
3
1
0

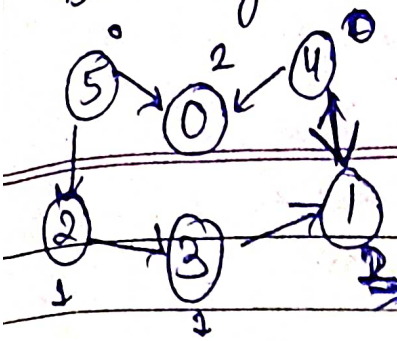
ans ↑

TC : O(N+E)

SC : O(N+N)

ASC : O(N)

BFS algorithm (Kahn's Algorithm)



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

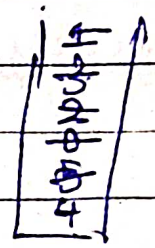
Date: Page:

→ find indegree of each vertex and save in array.

- use of queue
→ push all vertices with indegree 0.
→ pop element, do BFS and deduce the indegree of adjacent values by 1. and again check for indegree 0.
→ follow this untill queue is empty

TC: $O(N) + O(E)$

SC: $O(N) + O(N)$



Ans.

4	5	0	2	3	1

Q.8. Cycle detection using BFS in directed (Kahn's)

→ Kahn's algorithm
→ Reverse logic and thus we generate topological sort
→ counter variable will keep track of all the queue elements

logic: in cyclic graph there will be a situation when all the remaining vertices will have indegree greater than 0. and that will restrict traversal

TC: $O(N) + O(E)$ SC: $O(N) + O(N)$

Q9. Shortest distance in undirected graph.

→ if there is no weight assume it to be 1

→ given source

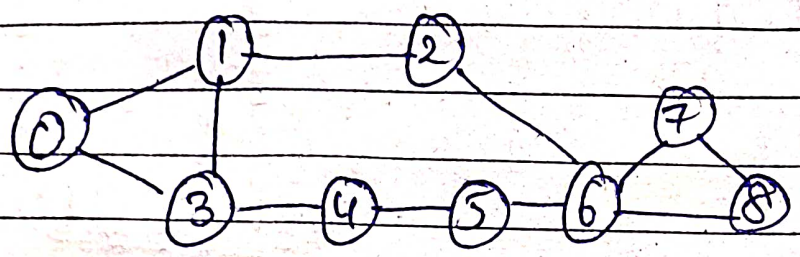
→ find shortest distance between all nodes and sou

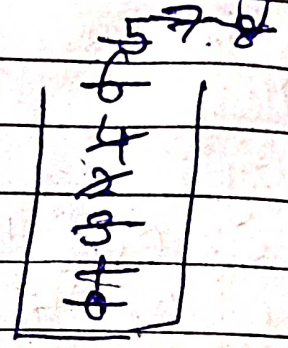
BFS algorithm.

→ take distance array of size n and make each as INT-MAX

→ take queue data structure holding nodes

→ add source node to queue marking dist[source] = 0.





node = ~~0~~ ~~1~~ ~~3~~ ~~2~~ ~~4~~ ~~6~~ ~~5~~ ~~7~~ 8

dist = ~~0~~ ~~1~~ ~~1~~ ~~2~~ ~~2~~ ~~3~~ ~~3~~ ~~4~~ 4

	0	1	2	3	4	5	6	7	8
dist	~~0~~	~~∞~~	~~0~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~

Ans

0	1	2	1	2	3	3	4	4

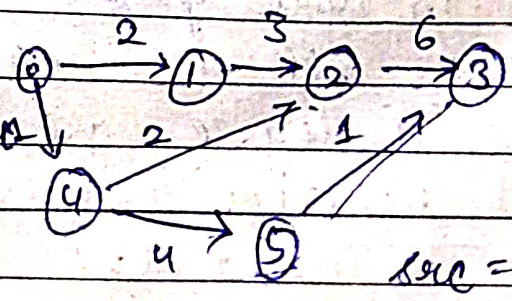
TC → O(N+E) SC → O(N) + O(N)

Q10. Shortest Path in Directed Acyclic graph. (weight

→ given source

→ find shortest distance from source to all other points.

→ use of pairs.



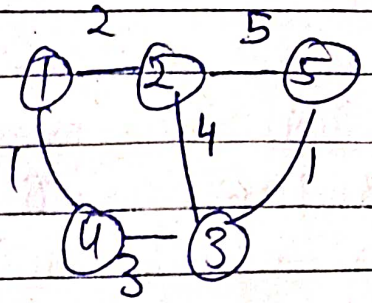
src = 0.

Task 1: find Topological sort for the graph. create distance array for storing result.

→ now find the shortest distances to all adjacent nodes. adding element from stack

TC → O(N+E) × 2. SC → O(2N) Date: Page: ASC → O(N)

Q11. Dijkstra's Algorithm



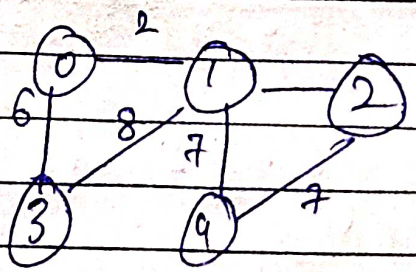
→ given graph (undirected) and source.
→ find shortest distance from source to all other points
→ use priority-queue → {dist, node}
→ priority queue should be min heap.
→ use as distance array initially INT_MAX.
dist[source] = 0.
→ iterating with BFS

shorter dist of adj node
~~(2,5) (5,5)~~
~~(1,3)~~
~~(1,4)~~
~~(2,2)~~
~~(0,1)~~

TC : O(N+E) ~~log~~ ≈ O(NlogN)
SC : O(N) + O(N)

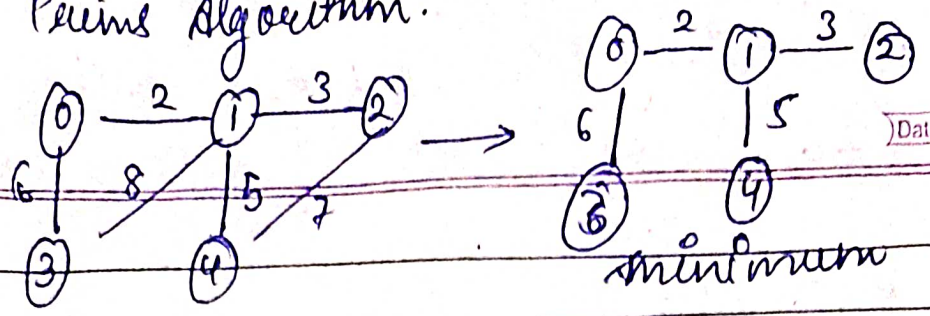
dist	0	1	2	3	4	5
	0	2	4	1	5	

Q12.
Minimum Spanning Tree :



spanningTree : when you can connect a node into a tree with n nodes and n-1 edges
→ and every node is reachable to every other node
** minimum : minimum cost of weights of edges
→ if we can draw tree from graph with n nodes and n-1 edges such that every nodes are reachable to every other nodes as well as we get minimum sum of all edges weight, then we call that tree as minimum spanning Tree

Prims Algorithm.

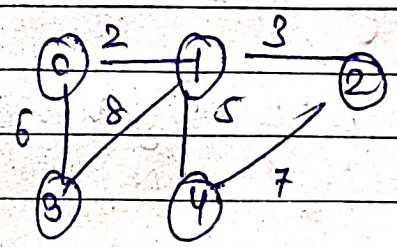


minimum spanning tree

→ start with source
→ find minimum edge attached with source and connect with other end node
→ again check min edge attached to the nodes
→ do this untill all graphs are connected
→ do not pick any edge that creates cycle

use of 3 array.

~~key~~



key

0	1	2	3	4
0	2	∞	3	∞

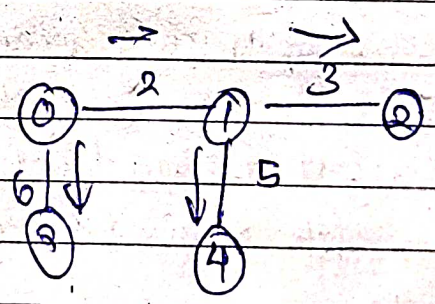
mst

0	1	2	3	4
T	F	T	T	T

get min possible key val.

parent

0	1	2	3	4
-1	0	1	0	1



get the edges: use parent node.

Brute force: TC: $O(N^2)$ SC: $O(3N)$

Optimized: use of heap data structure
priority queue for implementation
priority-queue < pair< int, int>, vector < pair< int, int>>, greater < pair< int, int>>> pq;

TC: $O(N \log N)$ SC: $O(3N)$

→ disjoint set.

- used in kruskals algo as well as to detect cycle in graph.

Date: Page:

2 operations ① Find Parent () ② Union ()

n = 7 → all nodes belong to single component

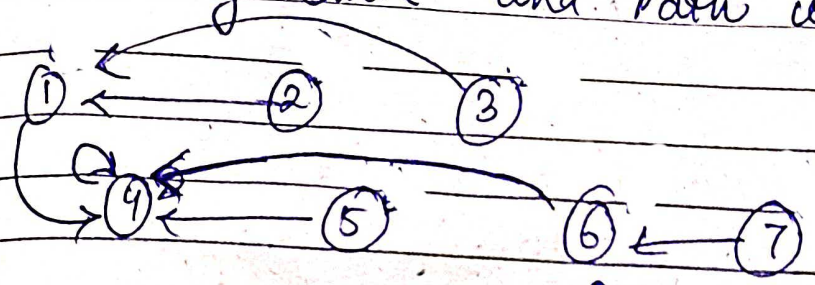
→ everyone is its parent thus FindPar(3) = 3.

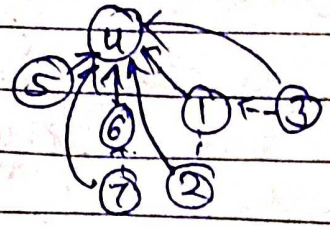
→ union(1, 2) : union components of 1 and 2

union (1) = 1 union (2) = 1

→ you can make any one as parent after union operation.

union by Rank and Path compression





Rank array

1	2	3	4	5	6	7
~~0~~	0	0	~~0~~	0	~~0~~	0

→ initial = 0.

union(1,2) → 1→1, 2→2 — you can make any one of them as parent, but make sure the incremented one's rank will increase

union(2,3) 2→1, 3→3 → lesser rank node gets connected to higher rank.

union(4,5) 4→4, 5→5

union(5, 6) 5→4, 6→6 (same rank)

union(6,7) 6→6, 7→7

union(3,7) 3→1, 7→6→4, 7→4 — Path compression. minimize movements to find parent of any node.

TC: $O(4\alpha) \approx O(4)$

SC: $O(N)$

tree.

→ store graph in linear data structure

So as to ~~sort~~ sort the edges on basis of weight.

→ greedily pick up shortest edge

→ find whether 1 and 4 belong to same edge.

→ you should use disjoint set

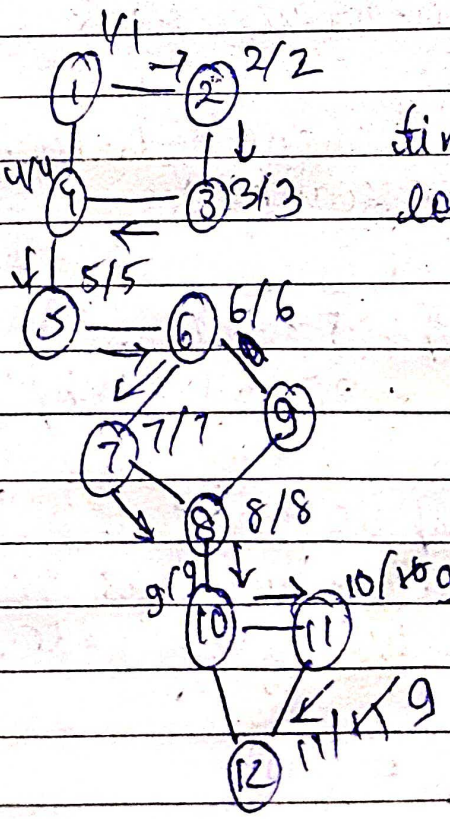
→ dont combine vertices in same component

TC → $O(M \log M) + O(M \times O(4\alpha)) \approx O(M \log M)$

SC → $O(M) + O(N) + O(N) \approx O(N)$

Bridges in graph

→ Those edges are called Bridges that after removal, divides graph into 2 different components.



use tin [ ]

low [ ]

tin = time of insertion

low = lowest time of insertion

if low [it] > tin [node]

→ it is bridge

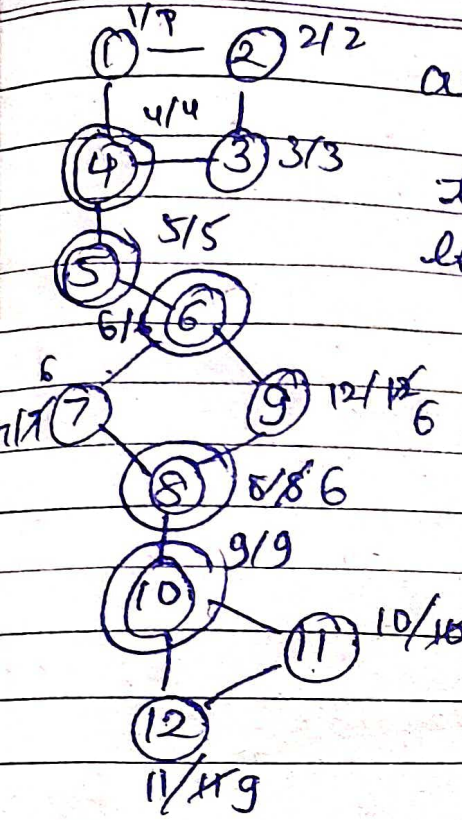
you can only update the low once dfs call is over

TC: $O(N+E)$ SC: $O(2N)$

Articulation Point

* node on whose removal graph is divided into 2 components.

Date : Page :



articulation points: 4, 5, 6, 8, 10.

tin [ ]
low [ ]

if(low[adj] >= tin[node] && parent!=-1)

→ there should be a parent for the node that you're checking

→ 1 has to be articulation pt but it has no parent

→ (parent != 1 as well as child > 1)

TC : O(N+E) SC : O(N)