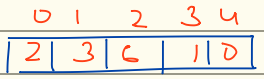
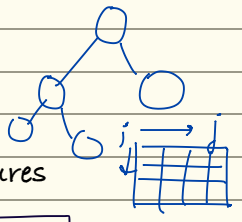
Graphs - Complete Notebook

codes, explanation, notes,
Implementation, algorithms and
questions from leetcode and codeforces ,

By Master The Codes

Graphs





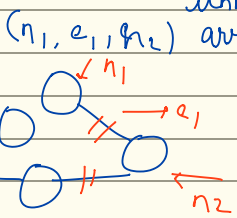
They are a non-linear Data Structure.
They contain Nodes and Edges

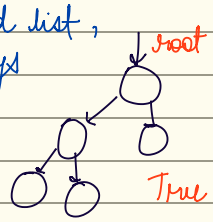
Data Stuructures

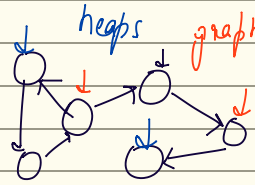
- Linear: queue, stack, linked list, arrays
- Non-linear: graphs, trees, matrix, heaps

Nodes → they store data

Edges → they connect the two nodes

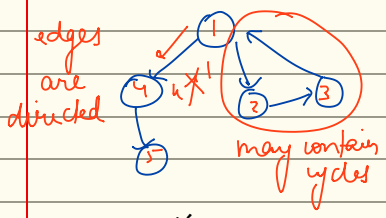




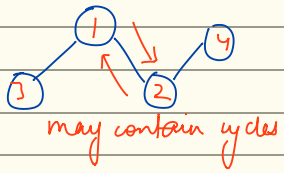


Types of Graphs

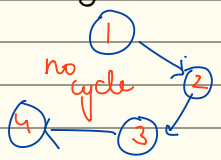
Uni-directional



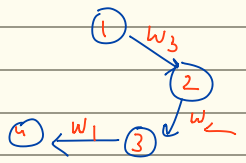
Bi directional



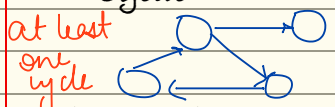
Acyclic



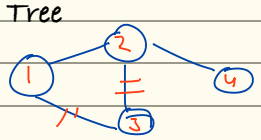
Directed



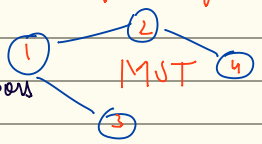
Cyclic



Minimum Spanning Tree

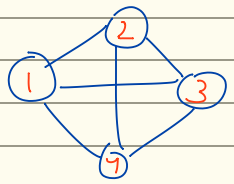


i → j i, j ∈ G



for n nodes, MST has n-1 edges

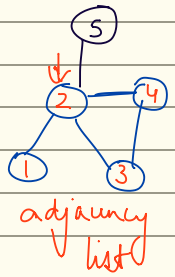
Complete



1 → {2, 3, 4}

i → (n-1) nodes

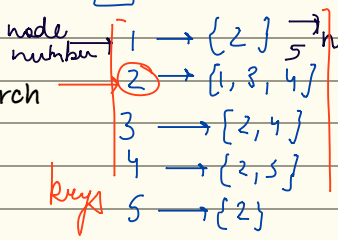
for each node i, n-1 edges



Basic Operations

1. Create
O(n)

map + list
K → V → List



2. Search
O(1)

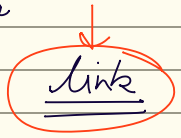
3. Add
O(1)

K, V → again

4. delete
O(1)

codes (Java)

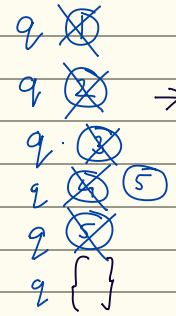
Like share and subscribe



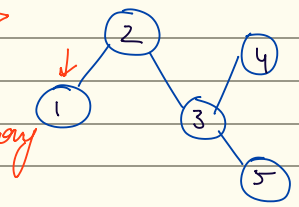
Algorithms for Traversal

queue
FIFO

Bfs



queue<node>
adj. list
visited array
[bitmask, array]



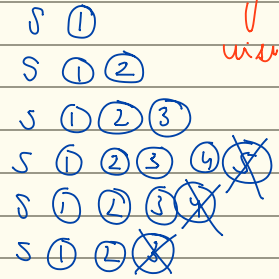
1 → [2]
2 → [1, 3]
3 → [4, 2, 5]
4 → [3]
5 → [5]

q is empty → bfs ends

LIFO
stack

Dfs

stack
adj. list
visited array
bitmask + array

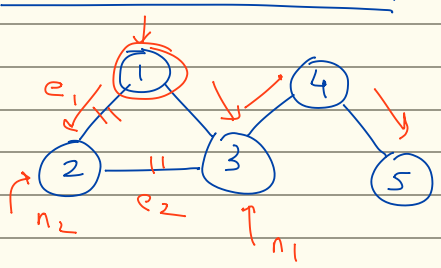


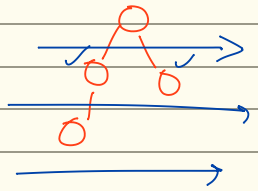
S → {} dfs ends

Graphs - Part 3

Traversals

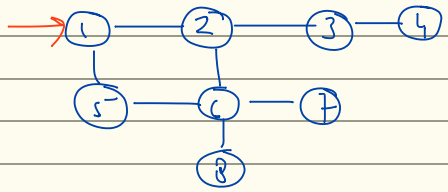
Breadth First Search (BFS)





Algorithm

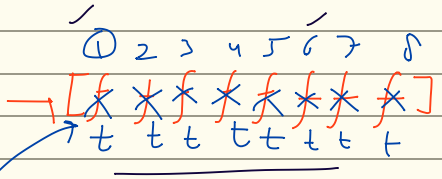
1. Create visited array (n) (n+1)
2. Enter source node into the queue
3. Check all neighbors of the node (visited)
4. If neighbor is not visited, add into the queue
5. Do this till queue becomes empty



queue

q (1)
q (2) (5)
q (5) (3) (6)
q (3) (7)
q (6) (4)
q (4) (7) (8)
q (7) (8)
q (8)
q { } → empty()

→ algorithm stops



add()
pop()
isEmpty()

bfs

adjacency list

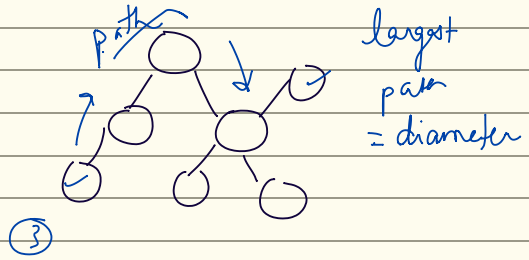
1 - {2, 5}
2 - {1, 6, 3}
3 - {2, 4}
5 - {1, 6}
4 - {3}
6 - {2, 5, 7, 8}
7 - {6}
8 - {6}

list of vertices of graph

Applications

1. Count number of nodes
2. Count diameter of graph
3. Count the depth of the graph
4. Perform greedy calculations

→ sorting
min max
best answer
(immediate)



largest path = diameter

Graphs - Part 4

Traversals

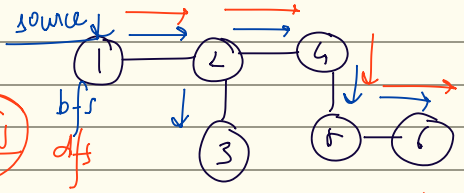
Depth First Search (DFS)

trees, matrix, graphs → Non linear DS

stack → Recursion

bfs

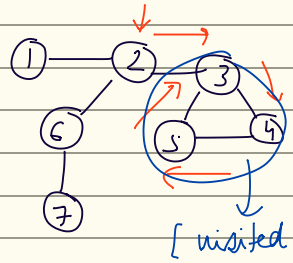
3rd video



dfs - branch wise

one branch at a time

true graph
X cycles
cycles ✓

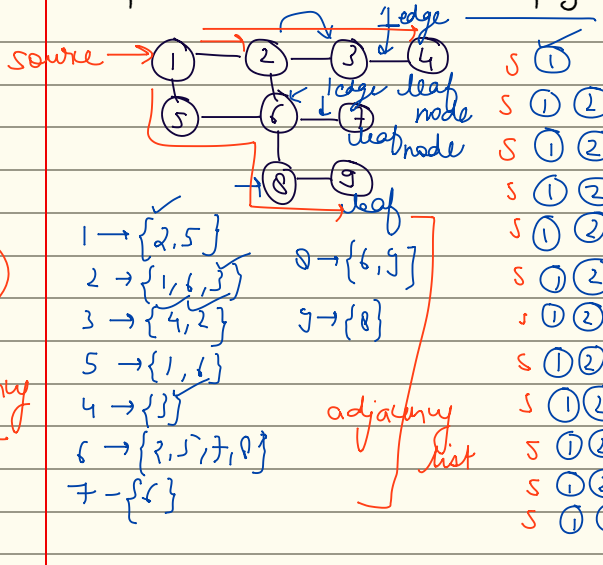


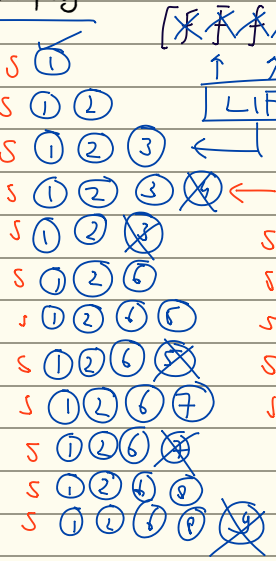
dfs may get struck in a cycle

[visited]

Algorithm

1. Create visited array
2. Enter source node into the stack (Recursion)
3. Check a neighbour of the node
4. add into the stack and mark as visited
5. When all neighbours are visited backtrack → Recursion
6. Repeat till the stack becomes empty






```
func (adj list, curr, visited)
{
  visited[curr] = true;
  for(int neighbor of curr)
  {
    func(adj list, neighbor, visited);
  }
}
```

1st video → adjacency list

O(n) time
O(n) space → stack

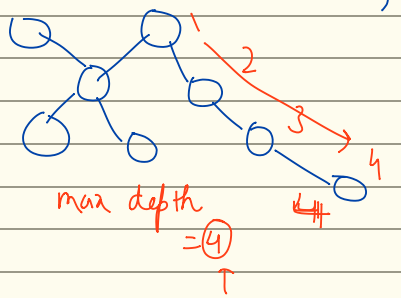
≃n
O(stack) → O(n) → Recursion

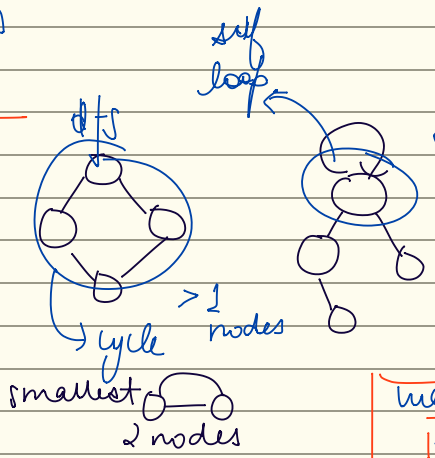
Applications

Like and share ↑

1. Count number of nodes
2. Count max depth of the graph
3. Count the number of paths
4. Perform greedy calculations
5. Detect cycles in the graph

Recursion → make a separate a video





we never use bfs for cycle detection

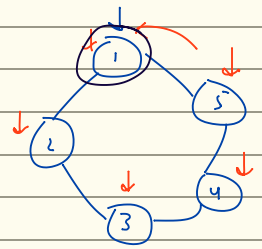
dfs is better than bfs in cycle detection

Graphs - Part 5

Cycle Detection Algoritm

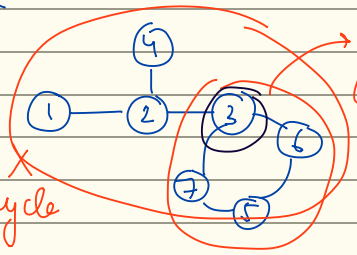
Algorithm

1. Start from a node
2. Mark the current node as visited
3. Move to a neighbour
4. If the neighbor is visited aside from the parent, then we have a cycle.

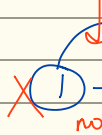


DFS / BFS

cycle



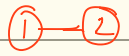
Is this a cycle?



not cycle

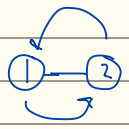
1 — 2 2 — 1

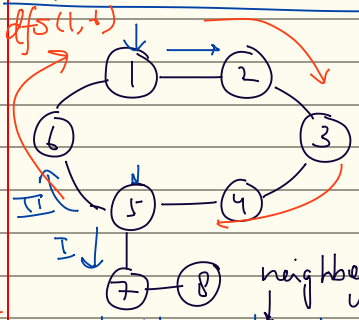
a graph must contain 3 nodes at least to have a cycle



Like share and subscribe

Depth First Search (DFS)

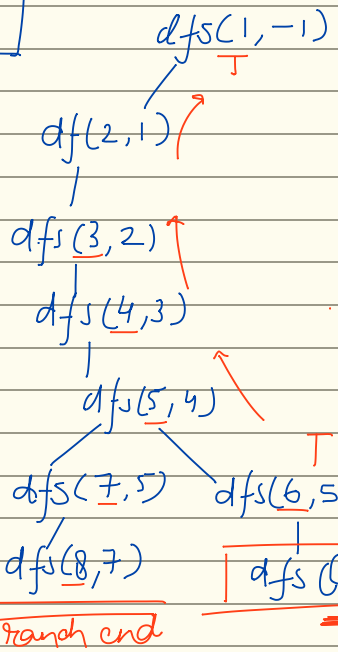




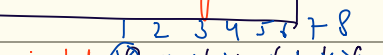
1 → {2,6} 2 → {1,3}
3 → {2,4} 4 → {3,5}
5 → {4,6,7} 7 → {5,8}
6 → {1,5} 8 → {7}

adjacency list

dfs



visited



Imp

If a visited node is found, then it should not be a parent

1st index

anytime visited node at first index, cycle is found

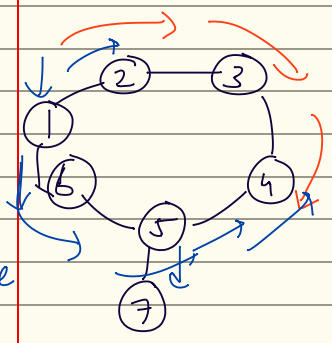
parent

dfs(1,1)

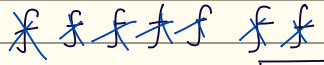
parent of every branch

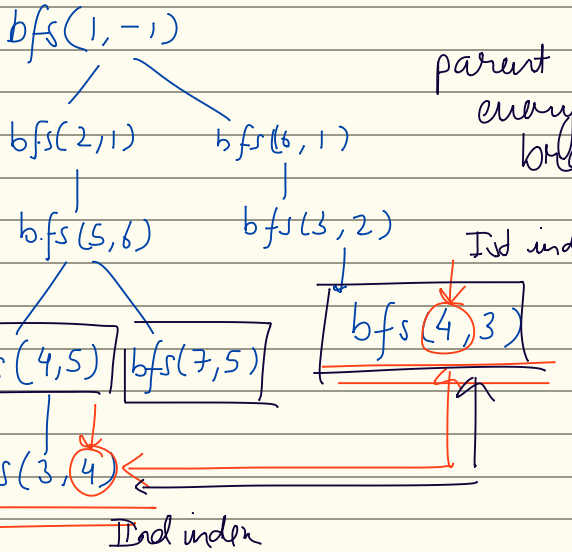
Breadth First Search (BFS)

two paths are reaching the same node → cycle



1 2 3 4 5 6 7





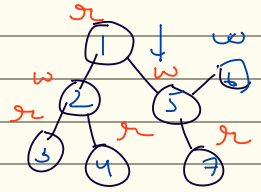
list, queue, array

extra work

DFS is better than BFS

Graphs

Bipartite Graphs - Graph Coloring with Two Colors



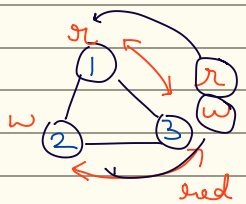
two colors

red white
yellow black

color → {0, 1}

any two neighbors cannot have same color

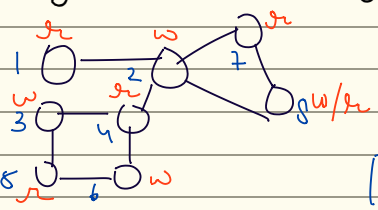
Imp
↳ neighbors should be of different color



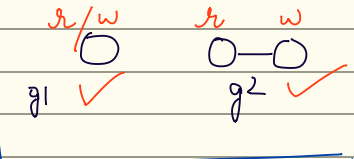
Properties of Bipartite Graphs

1. Always contain an ~~odd~~ even length cycle.

finding an odd length / even length ?
↓ dfs



bfs → coloring

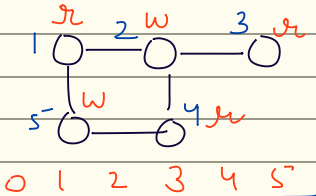


If a cycle exist then it has to an even length cycle

any graph, that contains a cycle of odd length
↳ not a bipartite

Coloring a Bipartite Graph

Use a color array

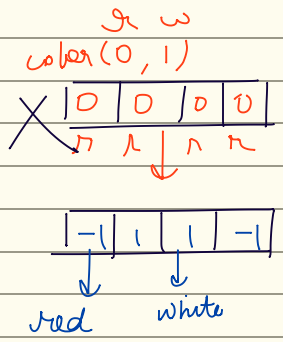


bfs → uncoloured (-1, 1)

no. of vertices → n+1 → Size

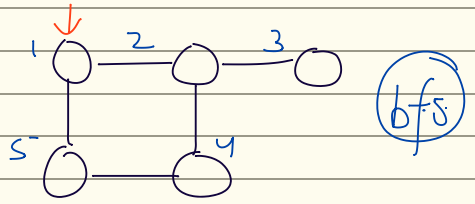
	-1	1	-1	-1	1
0	1	2	3	4	5

Color array



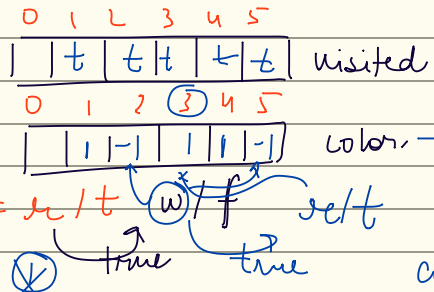
Algorithm

1. Create two arrays, one visited and one colour.
2. Perform bfs
3. While doing bfs, color the nodes with alternate colour every time you reach a new depth
4. If any two neighbouring nodes get same color, then graph is not Bipartite
5. Otherwise, when all the nodes get coloured the graph is Bipartite.



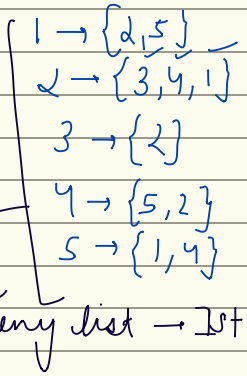
bfs

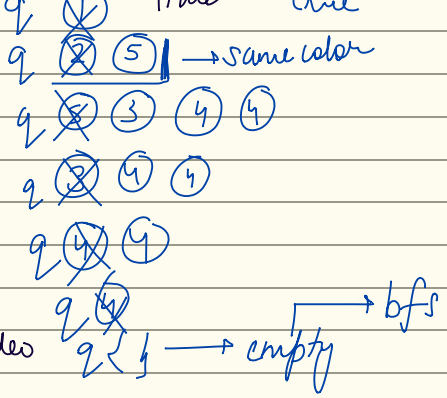
like share subscribe



boolean variable

color = true → red
color = false → white





color[3]
↓
1 → red
-1 → white