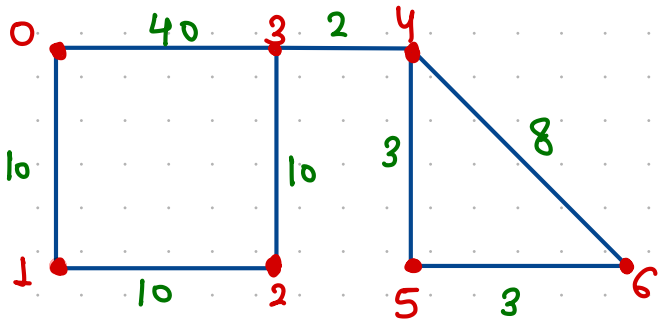
Graphs:

→ to organise data in a structured manner

D.S already covered

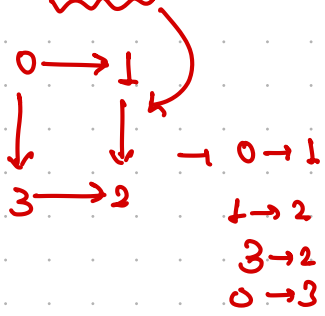
1. Arrays & AL
2. stacks & Queues
3. LinkedList
4. Trees

→ get, Set, Retrieve



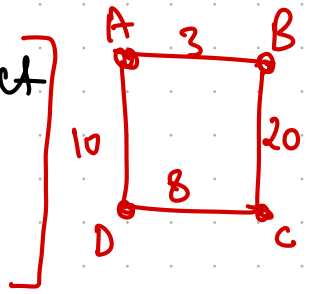
terms used in graphs →

1. Node / vertices → {0, 1, 2, 3, 4, 5, 6}
2. Edges → {0-3, 0-1, 1-2, 2-3, 3-4, 4-5, 5-6, 4-6}
3. weight / cost → {10, 40, 3, 2, 8}
4. directed / undirected → there is no direction in Edges.

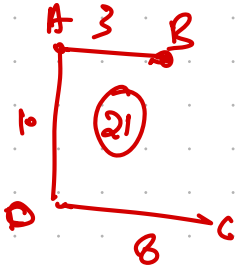


use of graph in Real life:

1. Suppose all nodes are representation of city. Now we have to reach from 0 to 6 with shortest distance
2. We have to move from 0 to 6 with less entrance of city
3. Assume all nodes as server, you have to connect all servers using wire, connect all server with min. length of wire.



min cost →

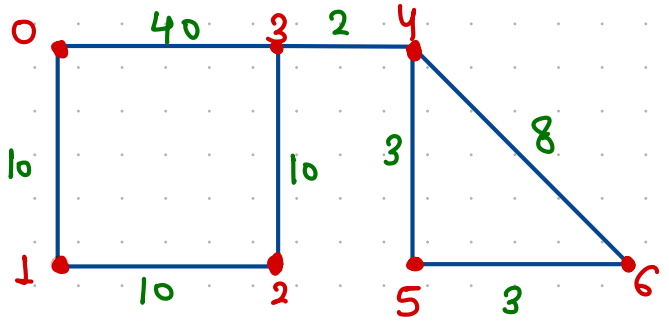


Implementation of graph: (undirected graph)

① Adjacency matrix

② Adjacency List

no. of vertex in graph = $n = 7$



① Adjacency matrix :→ $n \times n$ → matrix

	0	1	2	3	4	5	6
0	-1	10	-1	40	-1	-1	-1
1	10	-1	10	-1	-1	-1	-1
2	-1	10	-1	10	-1	-1	-1
3	40	-1	10	-1	2	-1	-1
4	-1	-1	-1	2	-1	3	8
5	-1	-1	-1	-1	3	-1	3
6	-1	-1	-1	-1	8	3	-1

→

Edge	wt.
0 – 1	→ 10
0 – 3	→ 40
1 – 2	→ 10
2 – 3	→ 10
3 – 4	→ 2
4 – 5	→ 3
4 – 6	→ 8
5 – 6	→ 3

Drawback of Adjacency matrix :

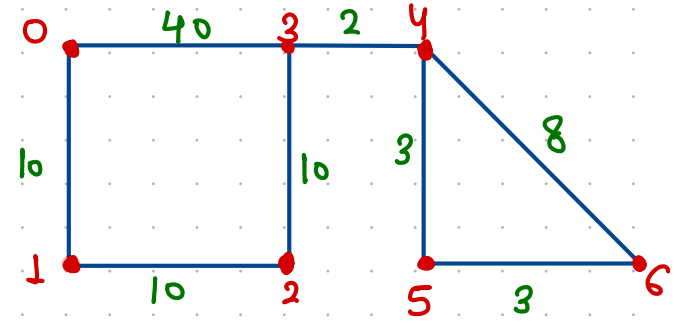
① wastage of memory

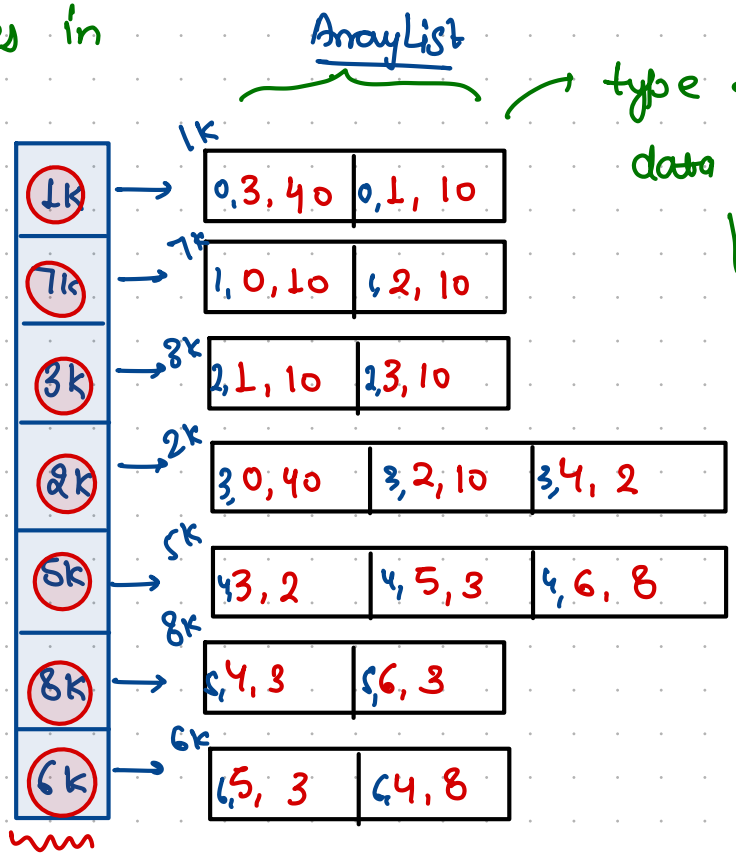
② Limited availability of vertices. (more than 10^4 nodes)

③ Naming rather than indexing.

Adjacency List:

n → no. of vertices in graph





Array of size 'n'

Array List of size n
↳ hold an Array List of type Edge.

Edge is wrapper class

Structure of Edge →

① src.] → ambiguous / needless
② nbr
③ wt

for 0 → 3 & 1 are neighbour

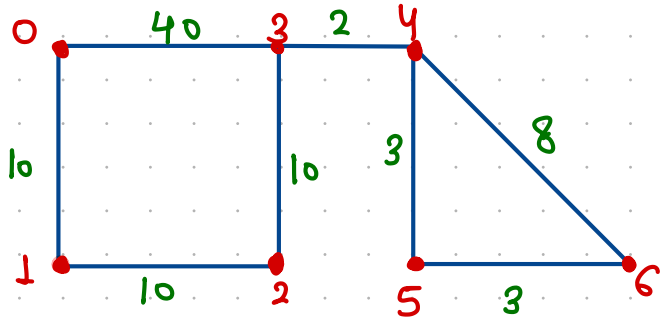
for 3 → 0, 2 & 4 are ".

Edge → v1 → src. vertex
v2 → nbr. vertex
wt → weight

ArrayList<Edge>[] graph;

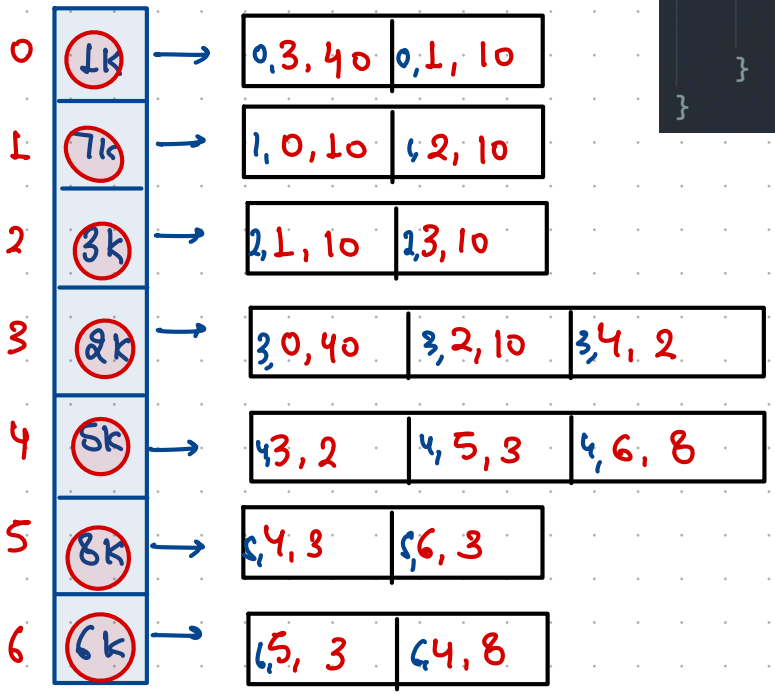
Edge :

0 - 3 40
0 - 1 10
1 - 2 10
2 - 3 10
3 - 4 2
4 - 5 3
5 - 6 3
4 - 6 8



How to display that information :

```
public static void display(ArrayList<Edge>[] graph) {
    for(int v = 0; v < graph.length; v++) {
        System.out.print("[" + v + "] -> ");
        for(Edge e : graph[v]) {
            System.out.print("[" + e.src + "-" + e.nbr + "@" + e.wt + "], ");
        }
        System.out.println();
    }
}
```




```
[0] -> [0-1@10], [0-3@40],
[1] -> [1-0@10], [1-2@10],
[2] -> [2-1@10], [2-3@10],
[3] -> [3-0@40], [3-2@10], [3-4@2],
[4] -> [4-3@2], [4-5@3], [4-6@8],
[5] -> [5-4@3], [5-6@3],
[6] -> [6-4@8], [6-5@3],
```

Has Path ? :

src, dst

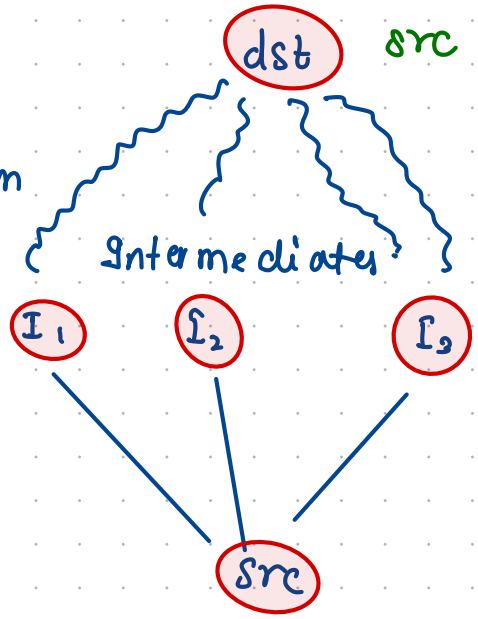
Is there any path between src to dst ?

NOTE: Haspath is DFS Algorithm.

Floodfill

↓

Application

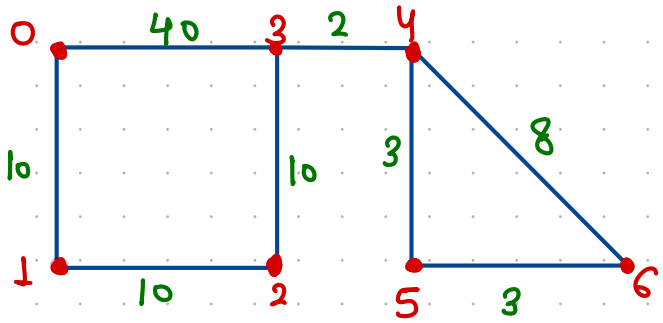


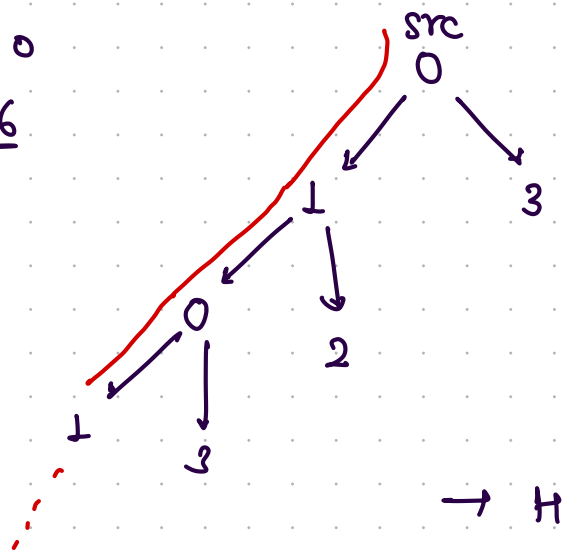

```java
public static boolean hasPath(ArrayList<Edge>[] graph, int src, int dst) {
    if(src == dst) return true;

    for(Edge e : graph[src]) {
        int nbr = e.nbr;
        boolean res = hasPath(graph, nbr, dst);
        if(res == true) {
            return true;
        }
    }
    return false;
}
```

src = 0

dst = 6

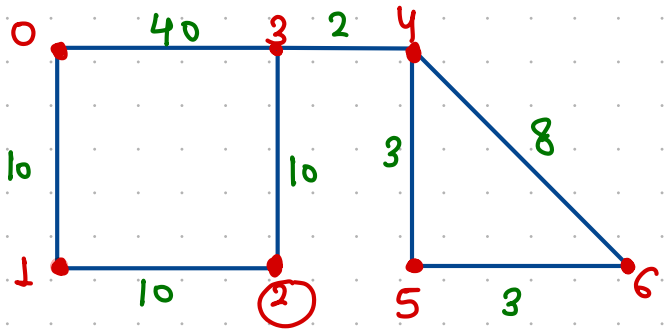




Here we are going towards already visited vertex.

→ How to avoid this problem.

→ Manage visited nbrs.



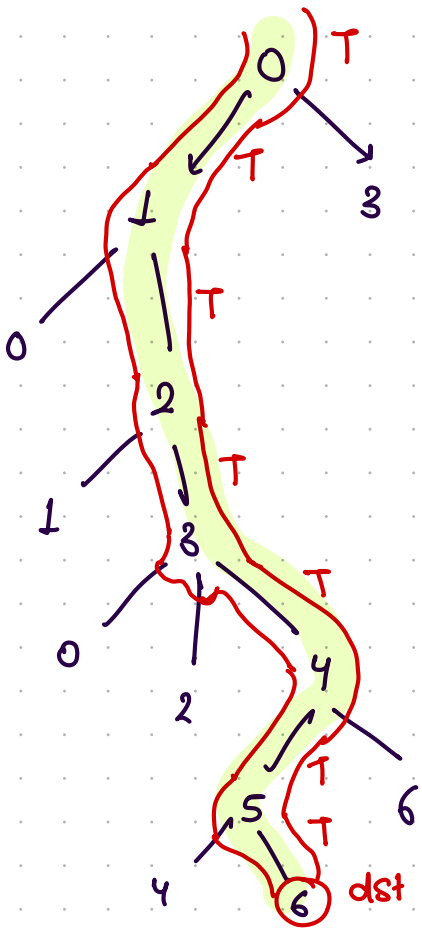
vis →

0	1	2	3	4	5	6
T	T	T	T	T	T	

src = 0
dst = 6

```
public static boolean hasPath(ArrayList<Edge>[] graph, int src,
    int dst, boolean[] vis) {
    if(src == dst) return true;

    vis[src] = true;
    for(Edge e : graph[src]) {
        int nbr = e.nbr;
        if(vis[nbr] == false) {
            boolean res = hasPath(graph, nbr, dst, vis);
            if(res == true) {
                return true;
            }
        }
    }
    return false;
}
```

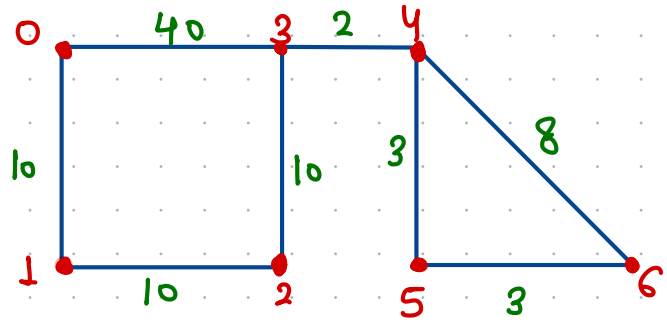


After mapping visited, we can avoid already visited vertex and we can reach at dst point

Print all paths:

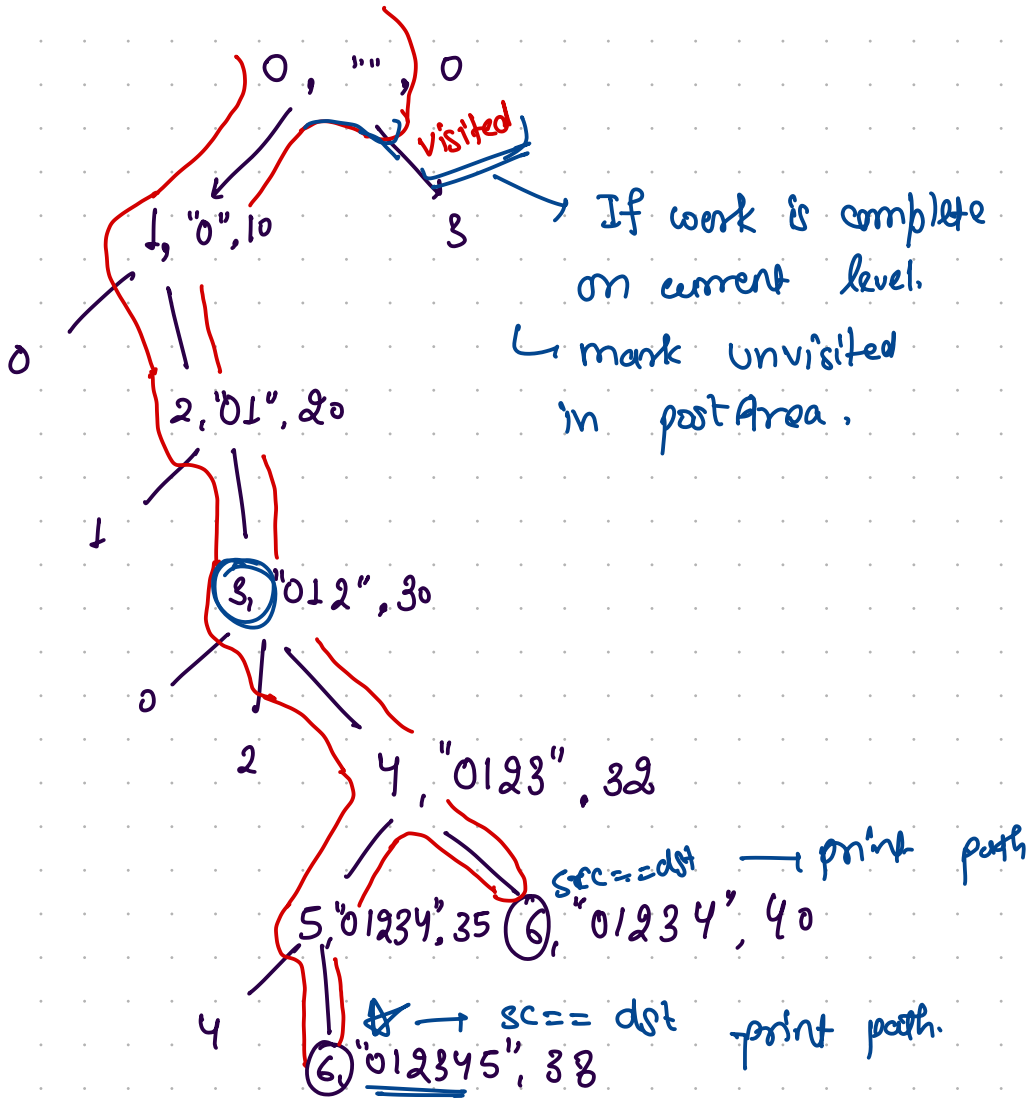
src = 0

dst = 6



	0	1	2	3	4	5	6
vis →	T	T	T	T	T	T	T

src, psf, wsf



If work is complete on current level.
↳ mark unvisited in postArea.

src == dst → print path

✱ → src == dst print path.

psf → path so far

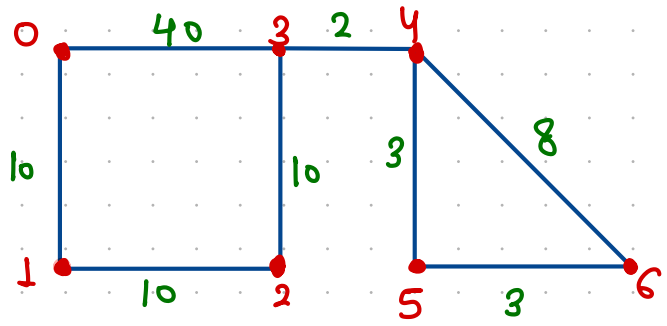
wsf → weight so far

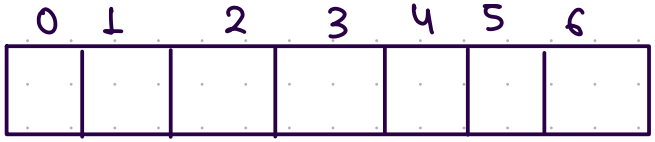
path1 → 0 1 2 3 4 5 6 @ 38

path2 → 0 1 2 3 4 6 @ 40

All paths → Expected

- 0 1 2 3 4 5 6 @ 38
- 0 1 2 3 4 6 @ 40
- 0 3 4 5 6 @ 48
- 0 3 4 6 @ 50



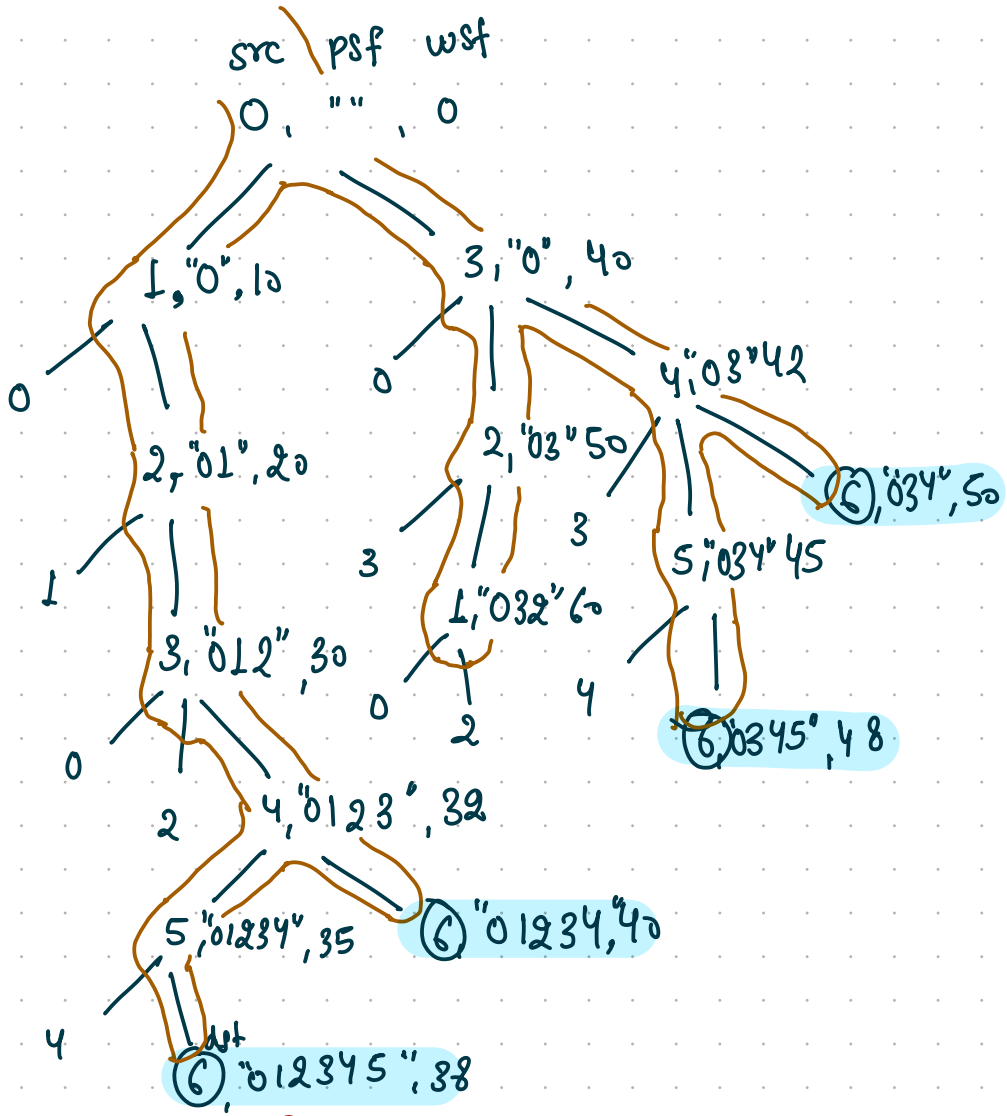


src=0
dst=6

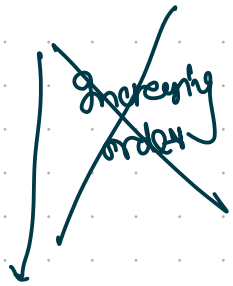
```java
public static void printAllPath(ArrayList<Edge>[] graph, int src,
    int dst, boolean[] vis, String psf, int wsf) {
    if(src == dst) {
        System.out.println(psf + src + " @ " + wsf);
        return;
    }

    vis[src] = true;
    for(Edge e : graph[src]) {
        if(vis[e.nbr] == false) {
            printAllPath(graph, e.nbr, dst, vis, psf + src, wsf + e.wt);
        }
    }
    vis[src] = false;
}
```

Not in final answer of all paths.



012346 @ 38
01246 @ 40
03456 @ 48
0346 @ 50



max path] → cost maximum, min path, ceil path, floor path

factor

Leetcode Submission

1971. Find if Path Exists in Graph

Easy 875 51 Add to List Share

There is a **bi-directional** graph with `n` vertices, where each vertex is labeled from `0` to `n - 1` (**inclusive**). The edges in the graph are represented as a 2D integer array `edges`, where each `edges[i] = [u_i, v_i]` denotes a bi-directional edge between vertex `u_i` and vertex `v_i`. Every vertex pair is connected by **at most one** edge, and no vertex has an edge to itself.

You want to determine if there is a **valid path** that exists from vertex `source` to vertex `destination`.

Given `edges` and the integers `n`, `source`, and `destination`, return `true` *if there is a* ***valid path*** *from* `source` *to* `destination`, *or* `false` *otherwise.*

Example 1:

```java
class Solution {


    private boolean hasPath(ArrayList<Integer>[] graph, int src, int dst, boolean[] vis) {
        if(src == dst) return true;
        vis[src] = true;
        for(int nbr : graph[src]) {
            if(vis[nbr] == false) {
                if(hasPath(graph, nbr, dst, vis) == true) {
                    return true;
                }
            }
        }
        return false;
    }

    public boolean validPath(int n, int[][] edges, int src, int dst) {
        ArrayList<Integer>[] graph = new ArrayList[n];
        for(int i = 0; i < n; i++) {
            graph[i] = new ArrayList<>();
        }

        for(int[] edge : edges) {
            graph[edge[0]].add(edge[1]);
            graph[edge[1]].add(edge[0]);
        }

        boolean[] vis = new boolean[n];
        return hasPath(graph, src, dst, vis);
    }
}
```