01 Shortest Paths

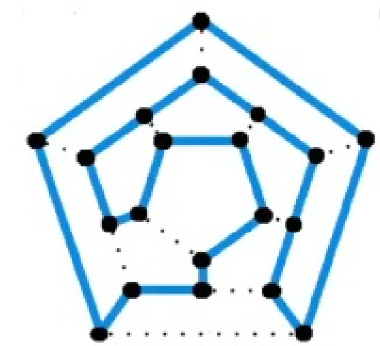
Google Maps for
Navigating Shortest Paths,
Computing Flight times & Costs

02 Social Networks

LinkedIn, Instagram, Facebook, Quora

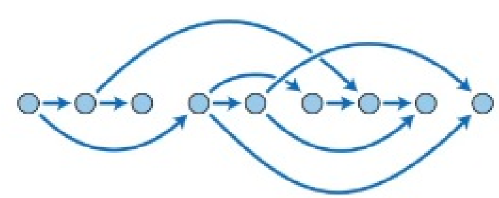
03 Shortest Cyclic Route

Min Cost Round Trip for School Vans,
Amazon Delivery Vans



04 Dependency Graphs

Resolving dependencies on Servers,
Software Installation etc

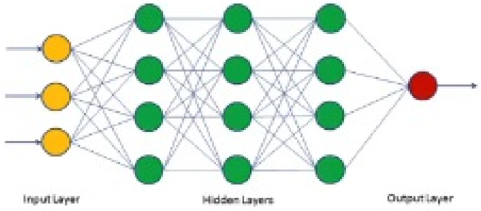


05 Routing Algorithms

Internet Routing

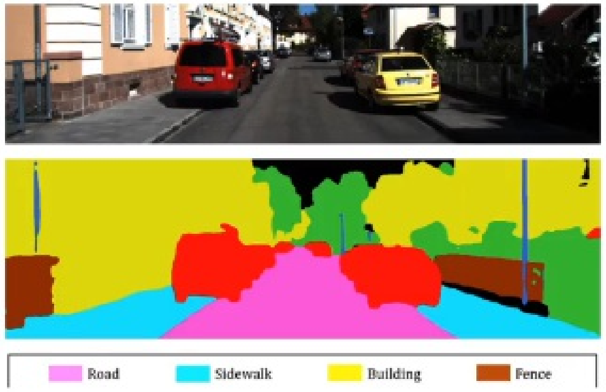
06 Computation Graphs

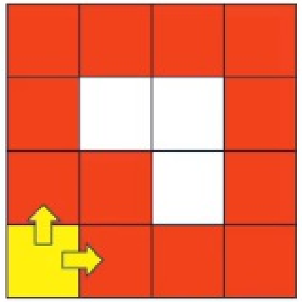
Deep Learning, Computations are done by optimising a graph like structure



07 Computer Vision

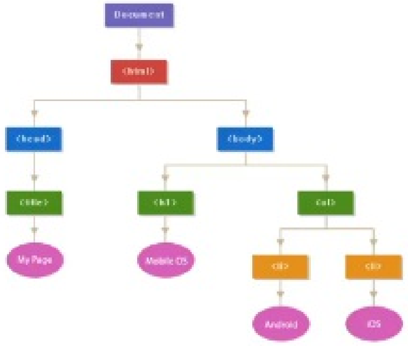
Image Segmentation, Flood Fill etc





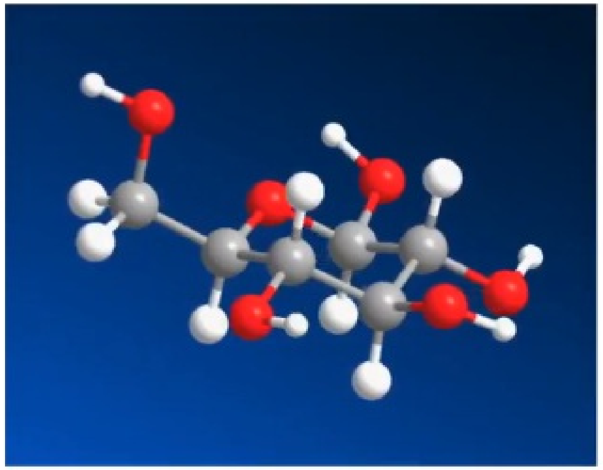
08 Web Crawlers

- Web Crawlers using BFS to crawl web
- Web Page is a DOM Tree, a tree is a graph without cycle,



09 Physics & Chemistry

Atomic & Molecular Structure,
Computer Processing

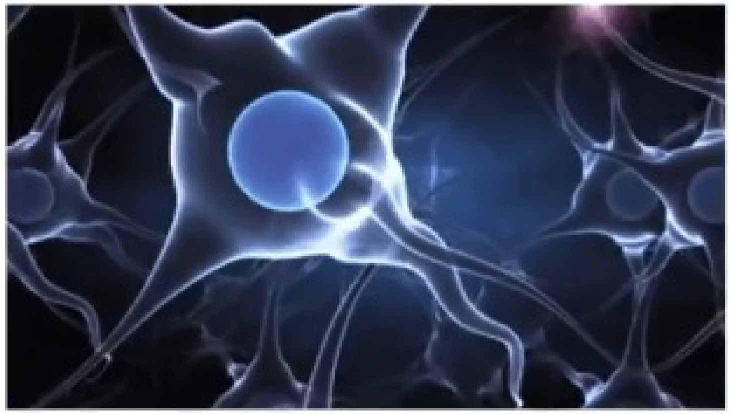


10 Graph Databases

Neo4j - Graph based database used in recommendtation engines, fraud detections & AI applicatons

Lot more Applications

Linguistics, Social Sciences,Biology & Neuroscience and more



Data Structure covered

1. Array + Array List
2. Stack
3. Queue
4. Linked list (with Rev Path)
5. Generic Tree
6. Binary Tree
7. BST
8. Priority Queue
9. Hash Map

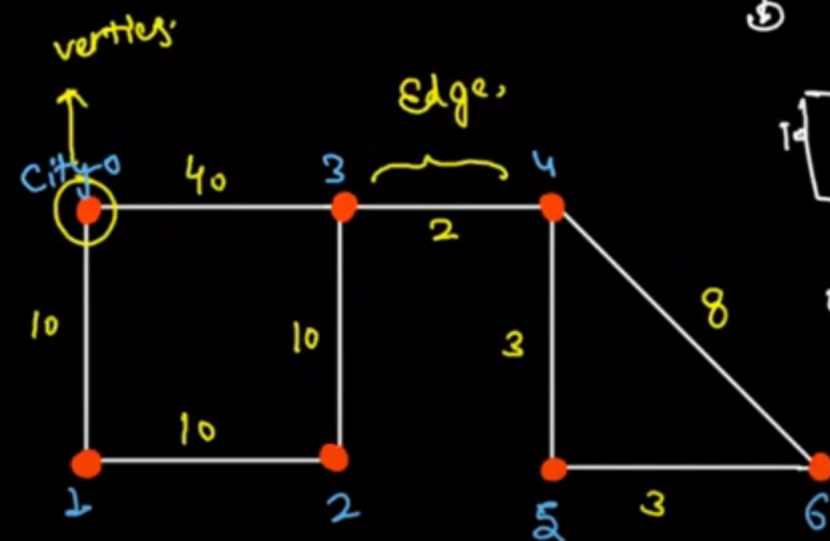
Linear Data Structure (1–4)

Non linear D.S. (5–7)

How to travel upwards

graph.

1. Nodes / vertices.
2. Edge
3. weights / Cost



Vertices → cities

Edge → connection b/w cities / Roads

wt → distance

Problem ① — Can i from city 0 to city 6

Problem ② → city 0 to city 6 with min toll.

Problem ③ → Connect all cities with min dist

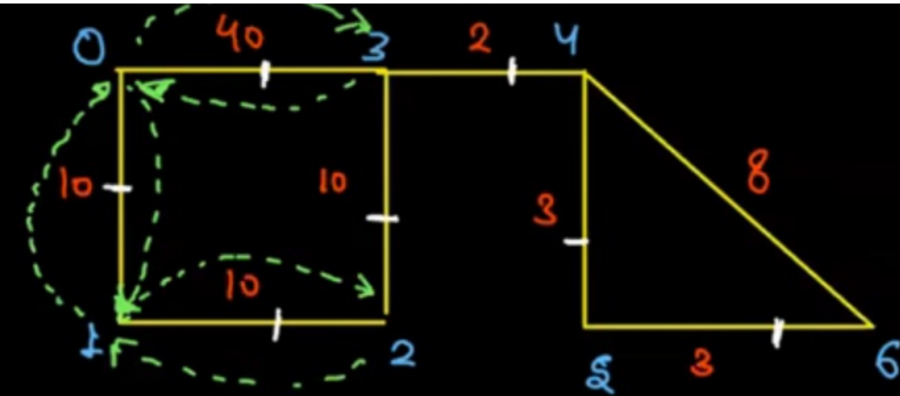
16:18 / 3:50:16

Implementation of graph:

① Adjacency Matrix ✗

② Adjacency List ☆

Undirected graph.



① Adjacency Matrix (n×n), n = no of vertices.

	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

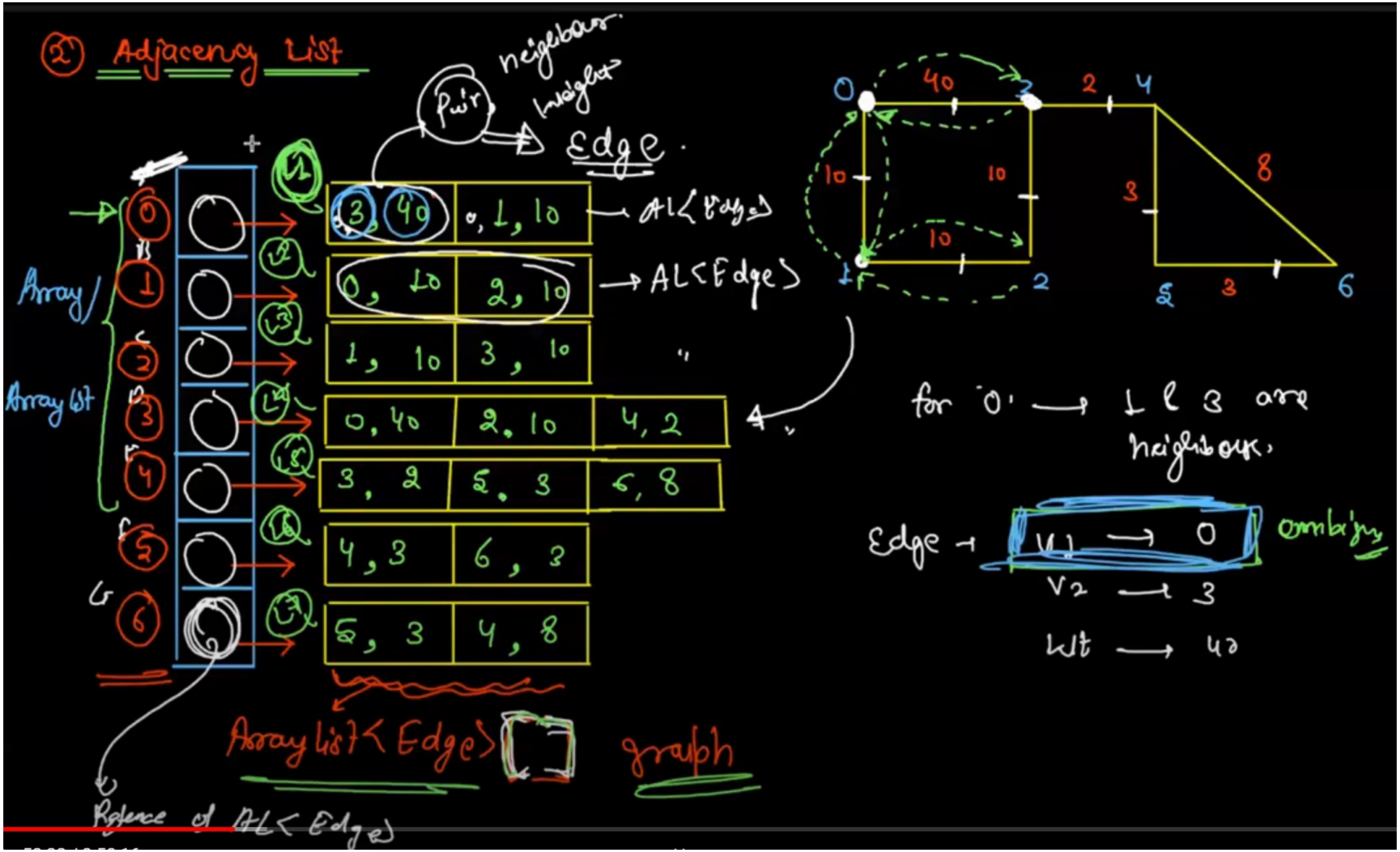
0 → 1 → 10
0 → 3 → 40
1 → 2 → 10
2 → 3 → 10
3 → 4 → 2
4 → 5 → 3
5 → 6 → 3
4 → 6 8

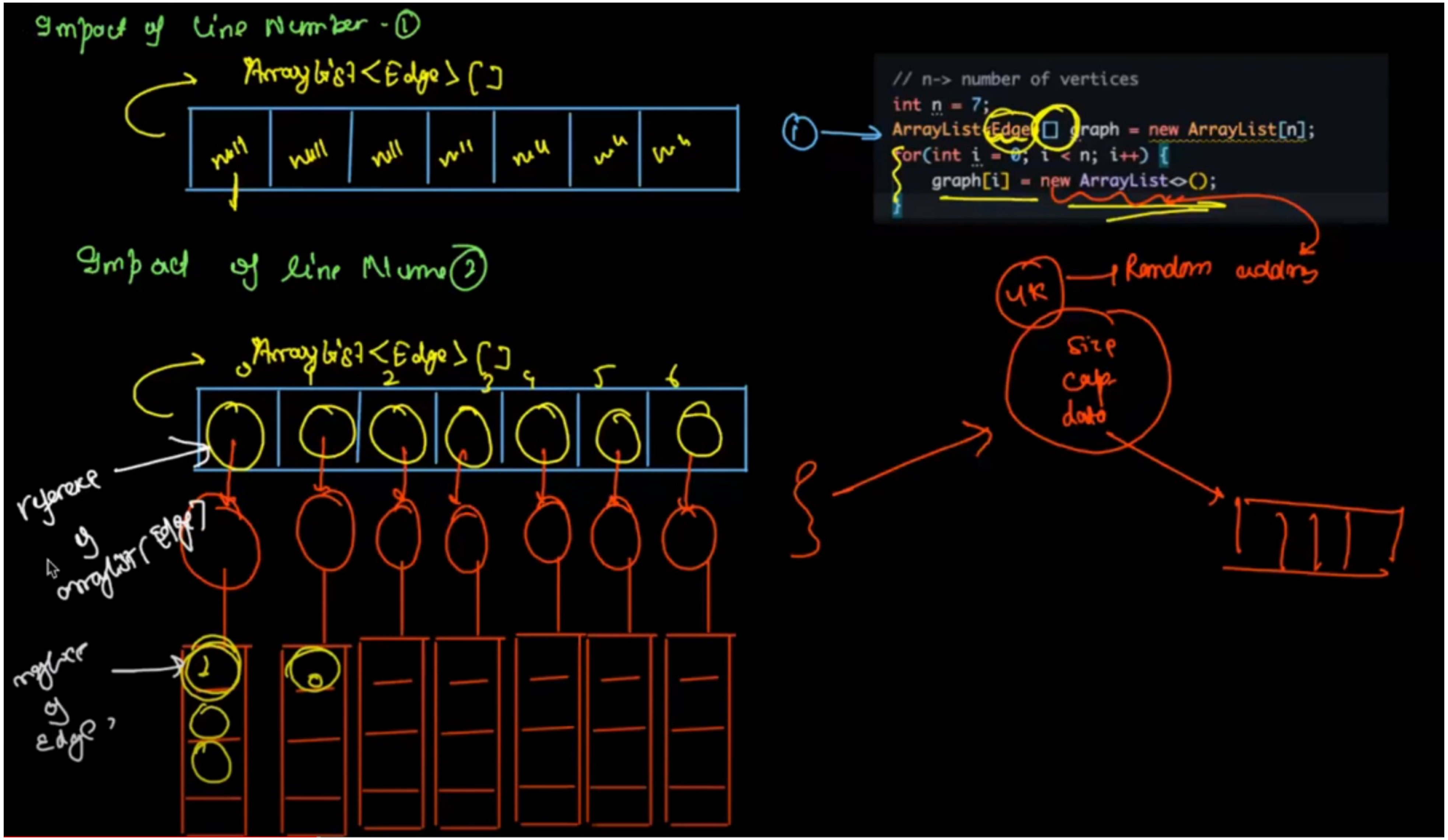
Drawbacks of Adjacency matrix

① wastage of memory.

② Limited availability for no. of vertices.

→ 10 nodes


② Adjacency List
pair
neighbour
weight
Edge
Array/
Array list
0, 1, 10
3, 40
→ AL<Edge>
0, 10
2, 10
→ AL<Edge>
1, 10
3, 10
0, 40
2, 10
4, 2
3, 2
5, 3
6, 8
4, 3
6, 3
5, 3
4, 8
Arraylist<Edge>[ ]
graph
Reference of AL<Edge>
for '0' → 1 & 3 are neighbours.
Edge → v1 → 0
v2 → 3
wt → 40
ambiguity


Impact of Line Number - 1
ArrayList<Edge>[]
null
Impact of line Number 2
ArrayList<Edge>[]
0 1 2 3 4 5 6
reference of arraylist[Edge]
arraylist of Edge
// n-> number of vertices
int n = 7;
ArrayList<Edge>[] graph = new ArrayList[n];
for(int i = 0; i < n; i++) {
graph[i] = new ArrayList<>();
}
4K
Random address
size
cap
data

```java
public static void fun() {

    // n-> number of vertices
    int n = 7;
    ArrayList<Edge>[] graph = new ArrayList[n];
    for(int i = 0; i < n; i++) {
        graph[i] = new ArrayList<>();
    }
```

```java
int[][] data = {
        {0, 1, 10},
        {0, 3, 40},
        {1, 2, 10},
        {2, 3, 10},
        {3, 4, 2},
        {4, 5, 3},
        {4, 6, 8},
        {5, 6, 3}
};
```

```java
public static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {
    graph[v1].add(new Edge(v1, v2, wt));
    graph[v2].add(new Edge(v2, v1, wt));
}
```

```java
for(int[] arr : data) {
    addEdge(graph, arr[0], arr[1], arr[2]);
}
```

```java
for(int i = 0; i < data.length; i++) {

    addEdge(graph,  data[i][0], data[i][1], data[i][2]);
}
```

```java
public static void display(ArrayList<Edge>[] graph) {
    for(int v = 0; v < graph.length; v++) {
        System.out.print("[" + v + "] -> ");
        for(int n = 0; n < graph[v].size(); n++) {
            Edge e = graph[v].get(n);
            System.out.print("[" + e.v1 + "-" + e.v2 + " @ " + e.wt + "], ");
        }
        System.out.println();
    }
}
```

Display :- →

order is Random & depend on addEdge.

[0] → (0←1 @ 10), (0←3 @ 40)

[1] → (1↔0 @ 10), (1↔2 @ 10)

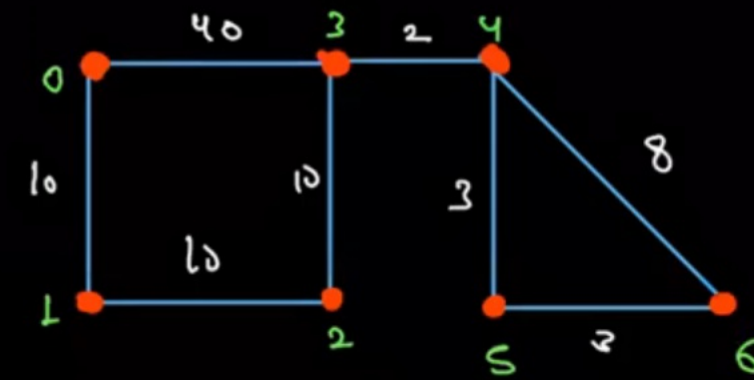
[2] → (2↔1 @ 10), (2–3 @ 10)

[3] → (3↔0 @ 40), (3→4 @ 2), (3→2 @ 10)

[4] → (4←3 @ 2), (4→5 @ 3), (4→6 @ 8)

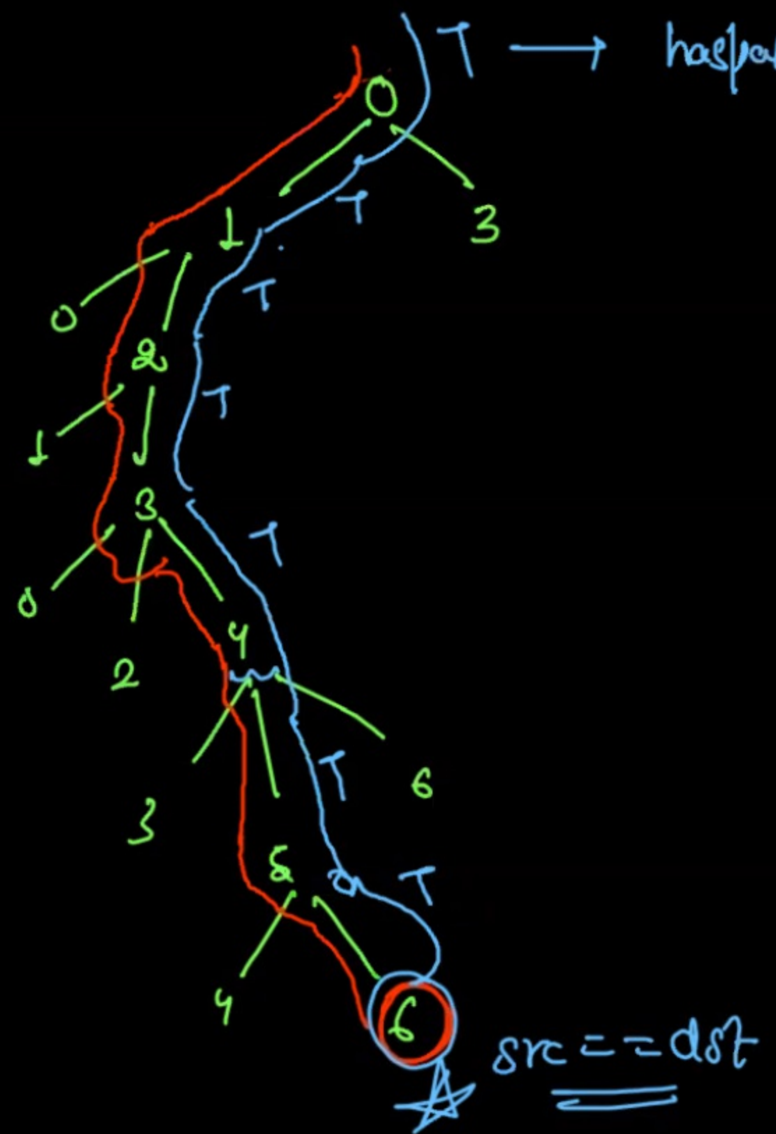
[5] → (5–4 @ 3), (5–6 @ 3)

[6] → (6–5 @ 3), (6–4 @ 8)



Haspath

dst = 6



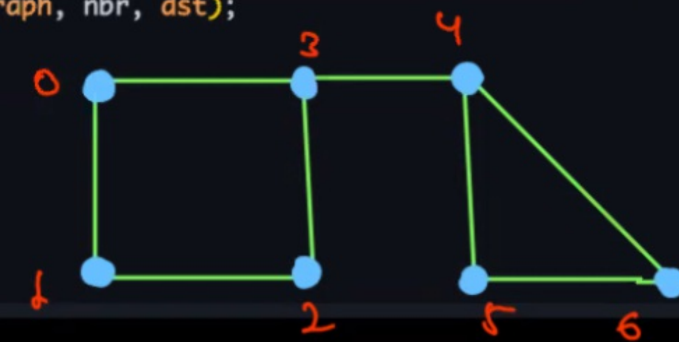

```java
public static boolean hasPath(ArrayList<Edge>[] graph, int src, int dst) {
    if(src == dst) {
        return true;
    }

    for(Edge e : graph[src]) {
        int nbr = e.v2;
        boolean res = hasPath(graph, nbr, dst);
        if(res == true) {
            return true;
        }
    }

    return false;
}
```

marking → visited (array → boolean)

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



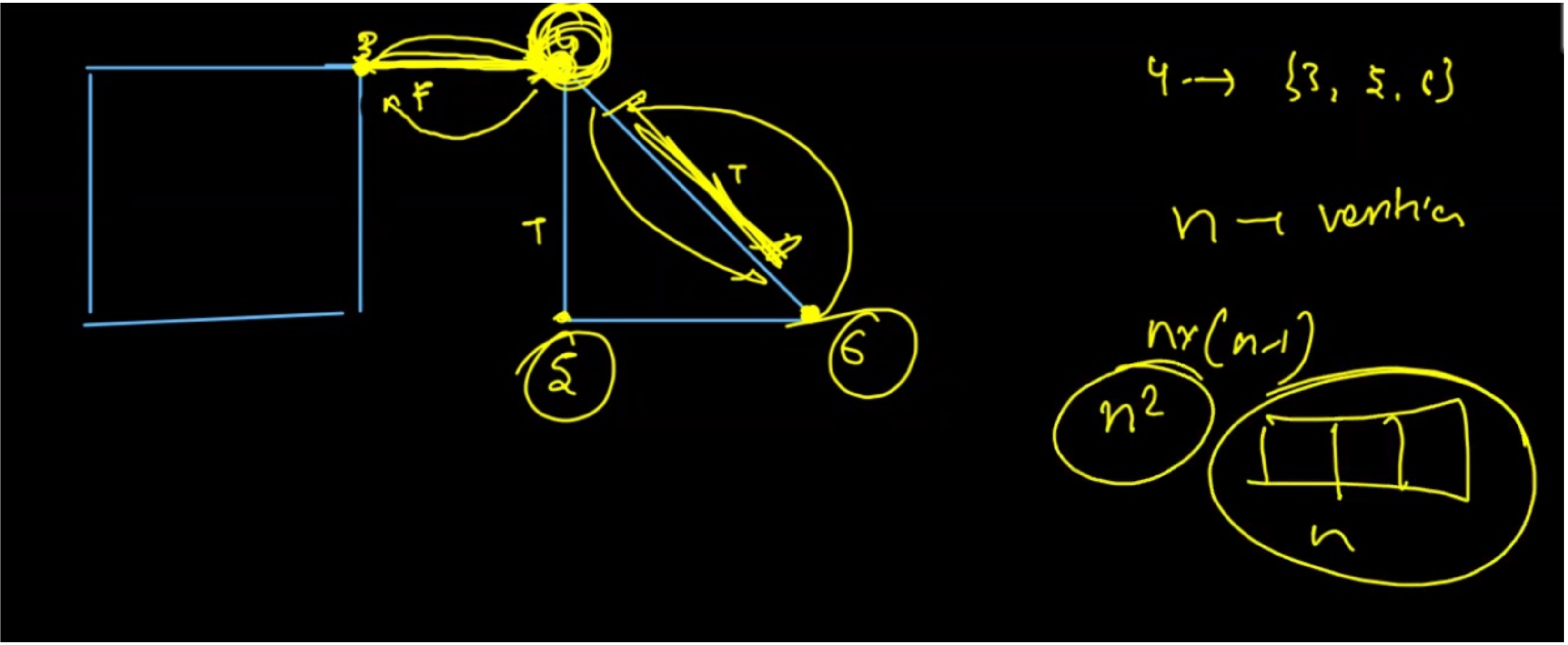
After mapping of visited array →

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

    vis[src] = true;
    for(Edge e : graph[src]) {
        int nbr = e.v2;
        // if neighbour is unvisited, move toward it
        if(vis[nbr] == false) {
            boolean res = hasPath(graph, nbr, dst, vis);
            if(res == true) {
                return true;
            }
        }
    }

    return false;
}
```

this base case is for find.



if you have n vertices
then you can have max n*(n-1)/2 edge

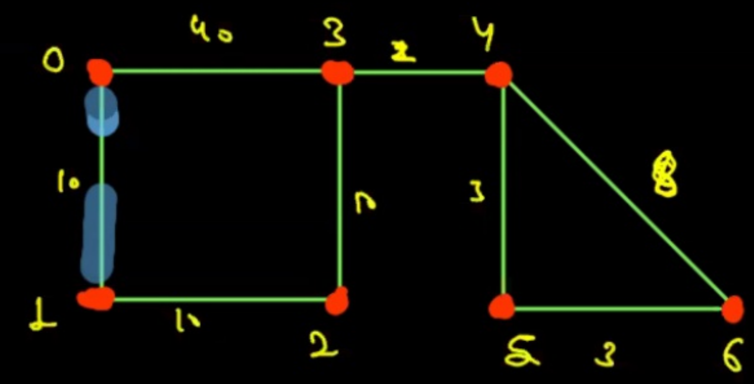
```java
// dfs -> depth first search
public static boolean hasPath(ArrayList<Edge>[] graph, int src, int dst, boolean[] vis)
    if(src == dst) {
        return true;
    }

    vis[src] = true;
    for(Edge e : graph[src]) {
        int nbr = e.v2;
        // if neighbour is unvisited, move toward it
        if(vis[nbr] == false) {
            boolean res = hasPath(graph, nbr, dst, vis);
            if(res == true) {
                return true;
            }
        }
    }
```

All path between src to dst:

src = 0

dst = 6



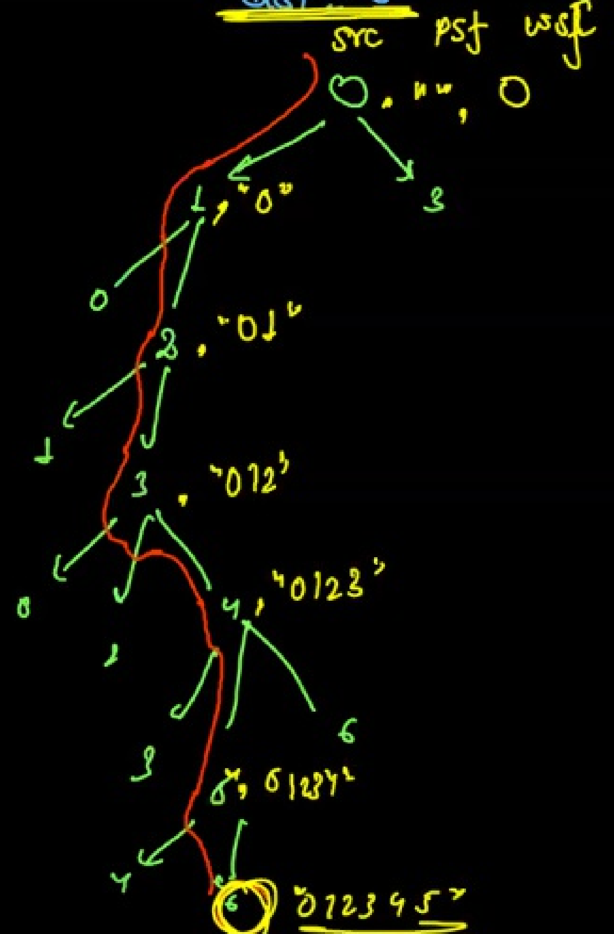
								wt sum
0	1	2	3	4	5	6)	@	38
0	1	2	3	4	6)		@	40
0	3	4	5	6)			@	
0	3	4	6)				@	

sum of wt

All path between src to dst

src = 0

dst = 6



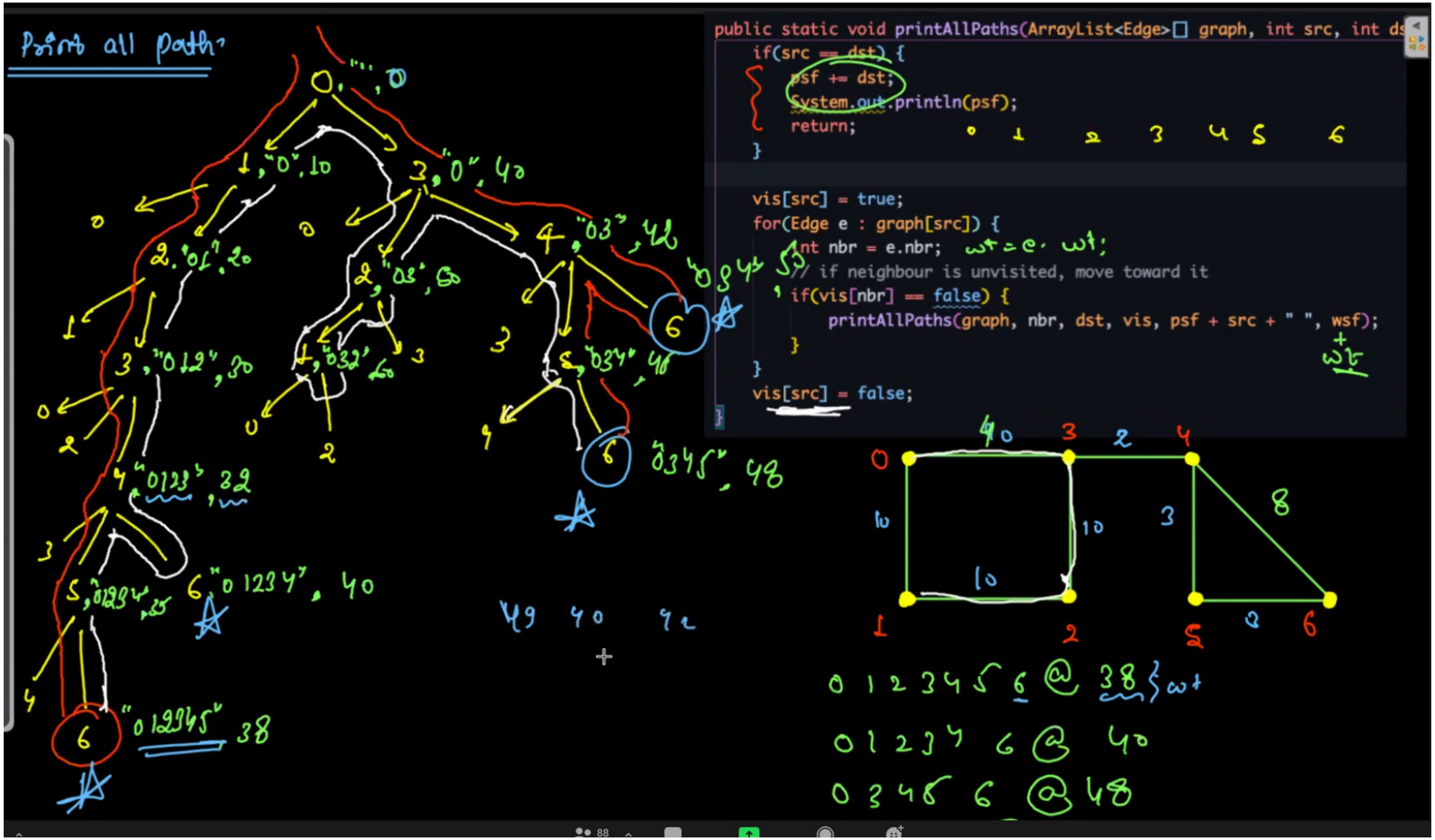

```
public static void printAllPaths(ArrayList<Edge>[] graph, int src, int d
    if(src == dst) {
        psf += src;
        System.out.println(psf);
        return;
    }

    vis[src] = true;
    for(Edge e : graph[src]) {
        int nbr = e.nbr;
        // if neighbour is unvisited, move toward it
        if(vis[nbr] == false) {
            printAllPaths(graph, nbr, dst, vis, psf + src, wsf);
        }
    }
}
```

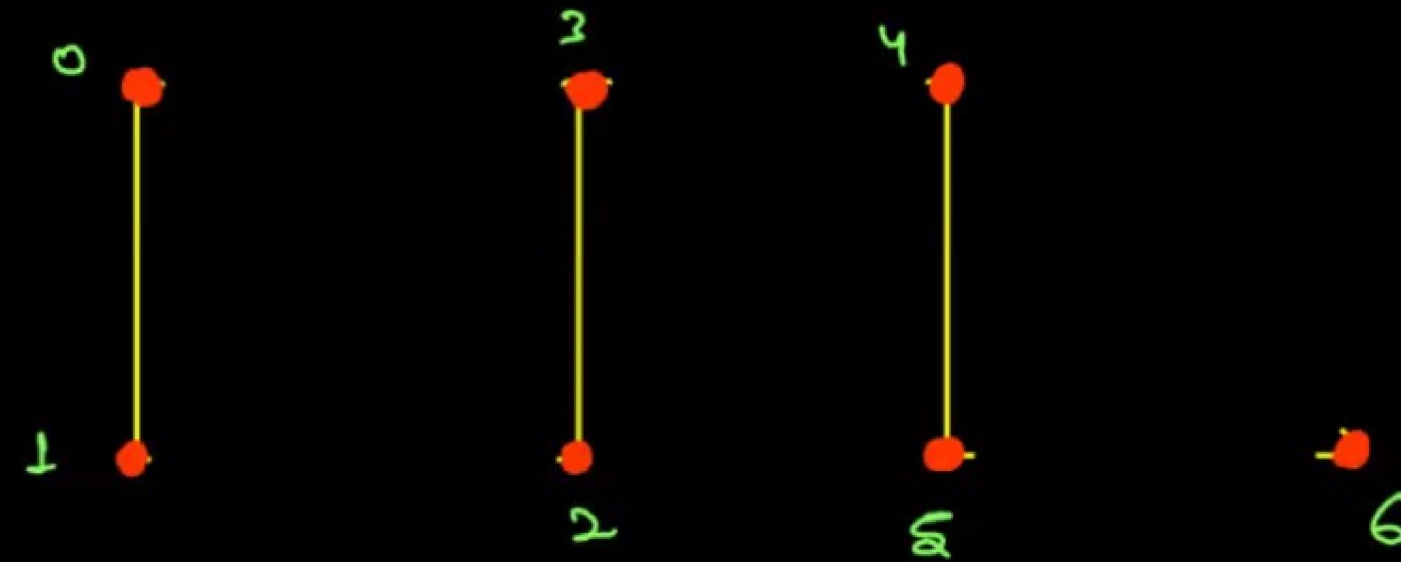
0	1	2	3	4	5	6
T	T	T	T	T	T	T

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

    vis[src] = true;
    for(Edge e : graph[src]) {
        int nbr = e.nbr;
        // if neighbour is unvisited, move toward it
        if(vis[nbr] == false) {
            printAllPaths(graph, nbr, dst, vis, psf + src + " ", wsf);
        }
    }
    vis[src] = false;
}
```


Print all paths
public static void printAllPaths(ArrayList<Edge>[] graph, int src, int ds
if(src == dst) {
psf += dst;
System.out.println(psf);
return;
}
vis[src] = true;
for(Edge e : graph[src]) {
int nbr = e.nbr; wt = e.wt;
// if neighbour is unvisited, move toward it
if(vis[nbr] == false) {
printAllPaths(graph, nbr, dst, vis, psf + src + " ", wsf);
}
}
vis[src] = false;
0 1 2 3 4 5 6 @ 38 wt
0 1 2 3 4 6 @ 40
0 3 4 5 6 @ 48

connected components



$\longrightarrow \{[0, 1], [2, 3], [4, 5], [6]\}$