WATCH THIS LECTURE ON VIDEO

Goal of the lecture:
To make the lecture content so obvious to the learners that they never need to revise, and don't forget this even if they have amnesia.

Graph: Introduction

After this lecture, the learner will find the basics of graph, the various graph terminologies and two main of the main graph algorithms **very intuitive and obvious**:

- DFS (Depth First Search)
- BFS (Breadth First Search)

Along with various other essential things about graph.

Not Bragging, just telling it to learners so that they learn confidently with faith in the teacher.

About Me

Hi, I am Utkarsh Gupta.

Upcoming Google Software Engineer.

I am one of the best Competitive Programmers in India.

Subscribe to my YT Channel for more content

Achievements:

India Ranks 2, 2, 2, 3 in Google Kickstart Round A,B,C,D respectively.

Grandmaster on Codeforces (India Rank 2)

7 star coder on Codechef

Watch me do Leetcode Weekly Contest in less than half time

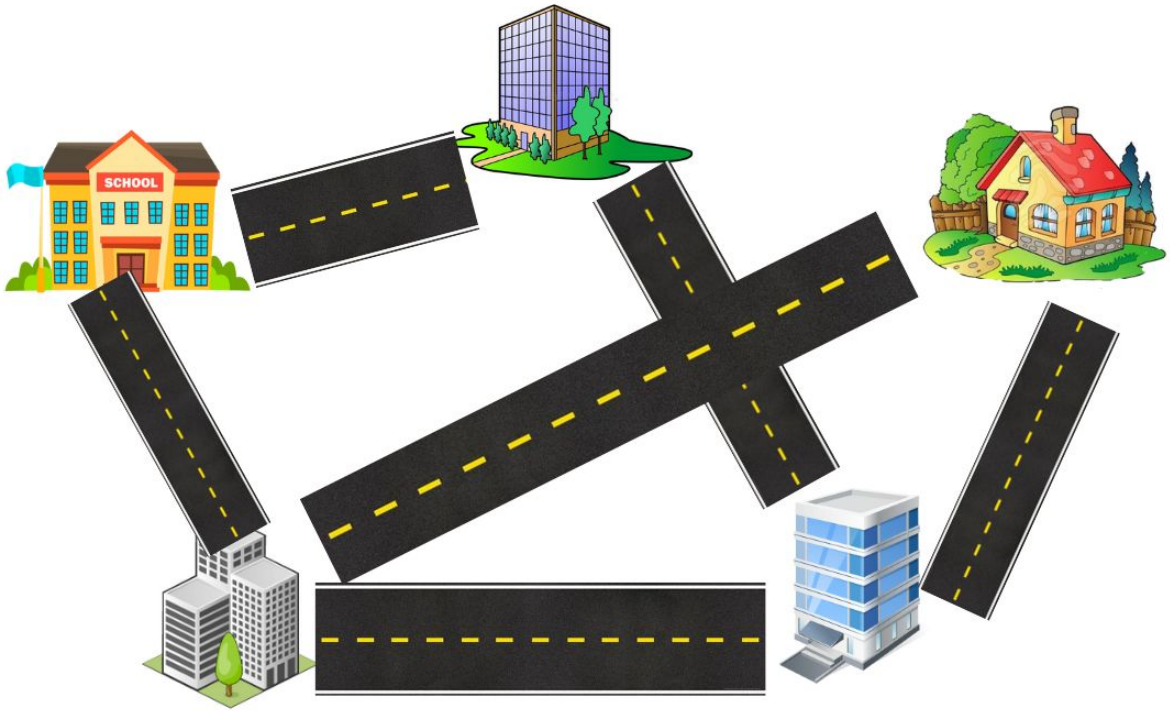
Prerequisites

- Should have some problem solving experience
- Should know Standard Template Library in C++, or equivalently Java containers like ArrayList, etc
- Should know recursion
- Should pay attention to this lecture

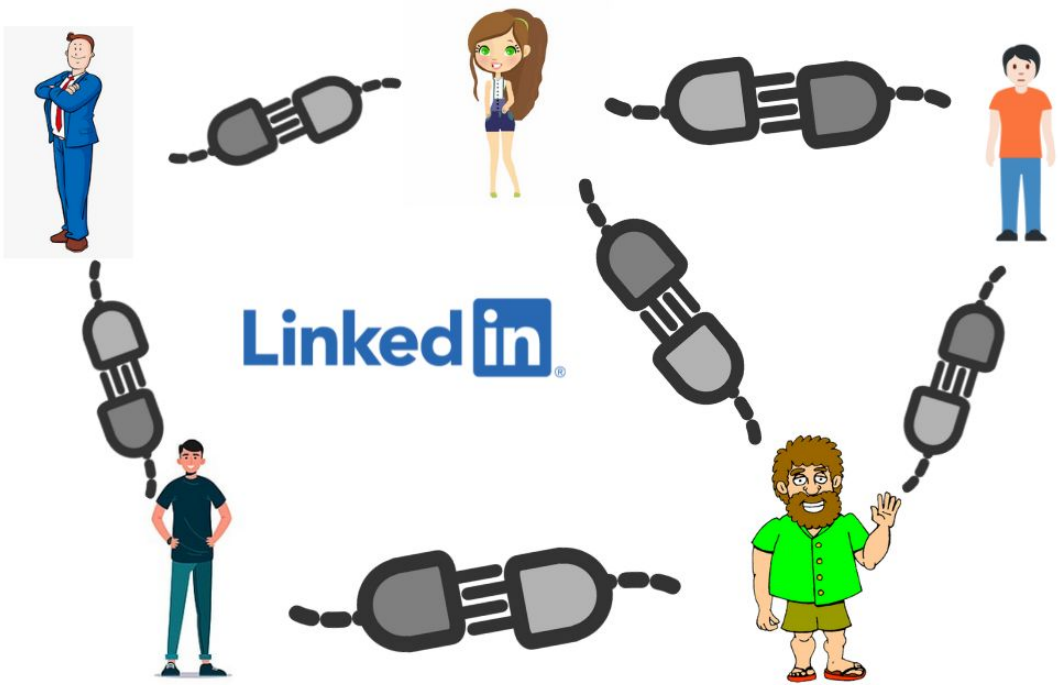
Why do we need graphs?
Why did anyone think of graphs?

And wait,
What are graphs?

A city with roads and buildings



LinkedIn Connections between people



Notice the similarity?

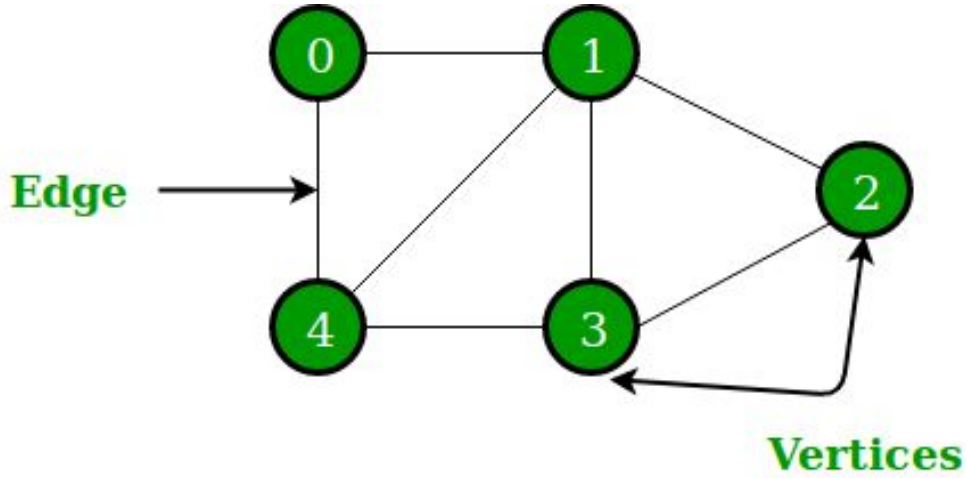
Such similarity can be found in a lot of places. Any “network” has such similarities. Even the problems and solutions to the problems based on such networks can be extremely similar.

For example, these 2 problems are almost equivalent:

- Minimum how many roads you have to change to get from home to school?
- A person on linkedin is your 1st, 2nd, or 3rd connection?

Whenever programmers and mathematicians find such similarities in unrelated problems, they come up with an abstract structure to generalize it. In this case, Graphs.

Graph



“Vertices” or “Nodes”: Represent Entities such as buildings, or people on Linkedin
“Edges”: Represent the relationships between the entities

The vertices along with edges together, are called a graph.
The vertices that are joined by an edge are called “adjacent”. For example, 1 and 3 are adjacent.

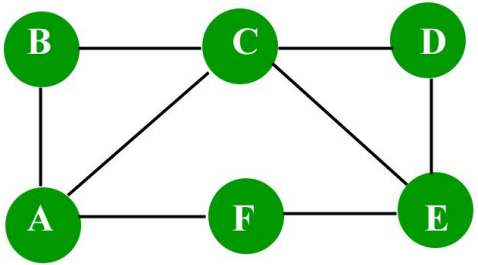
Types of Graphs

- Directed and Undirected
- Weighted and Unweighted

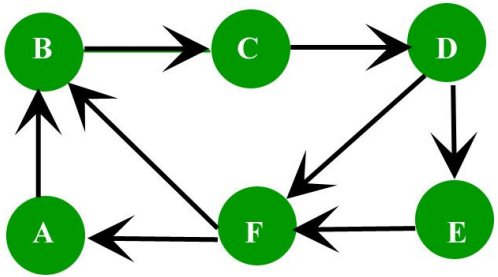
Note: Other types of graphs like Trees, DAG, Bipartite Graphs etc will be covered later in the series.

Directed and Undirected Graphs

- In directed graphs, the connections are one-directional, for example, A city with one-way roads, or the linkedin “followers” system. If A is a follower of B, it doesn’t mean that B is also a follower of A.
- In undirected graphs, the relations are symmetric and non-directional, for example, LinkedIn connections. If A is a connection of B, then B is also a connection of A.



Un-directed graph



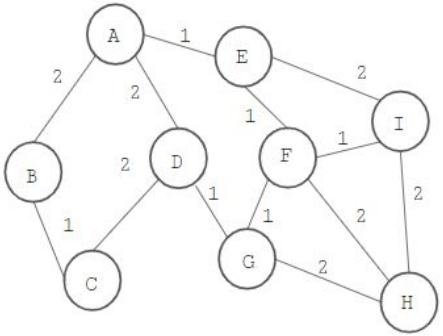
Directed Graph

Arrow-heads represent information about the direction of relation. For example, B follows C.

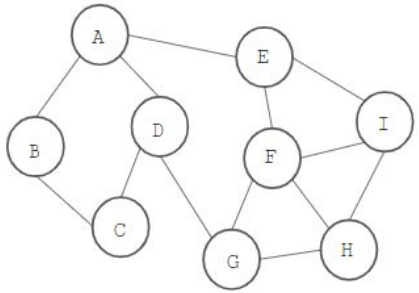
Weighted and Unweighted Graphs

- In weighted graphs, there is some extra information called “weight” on the edges of the graph. This weight denotes some additional information about the relations. For example, if a graph represents a metro-network, the weights on edges can represent the prices to get from point A to point B, etc.
- In unweighted graphs, there is no additional information stored on the edges.
  NOTE: *From a problem-solving POV, an unweighted graph is equivalent to a weighted graph in which all edges have the same weight.*

Weighted graph



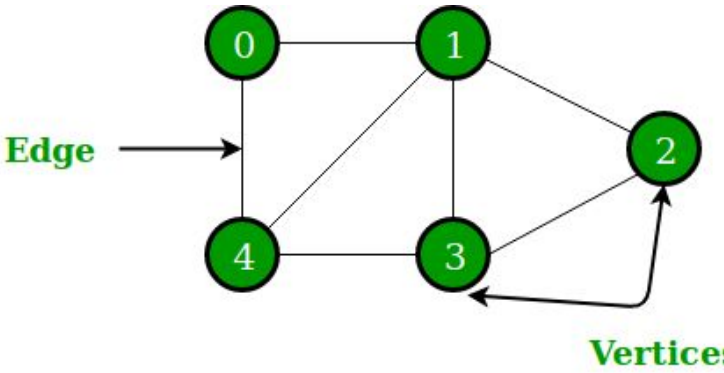
Unweighted graph



How do we represent a graph in a computer program?

Adjacency Matrix Representation

If there are V vertices, take a matrix of size VxV and store which vertices are adjacent to which vertices.



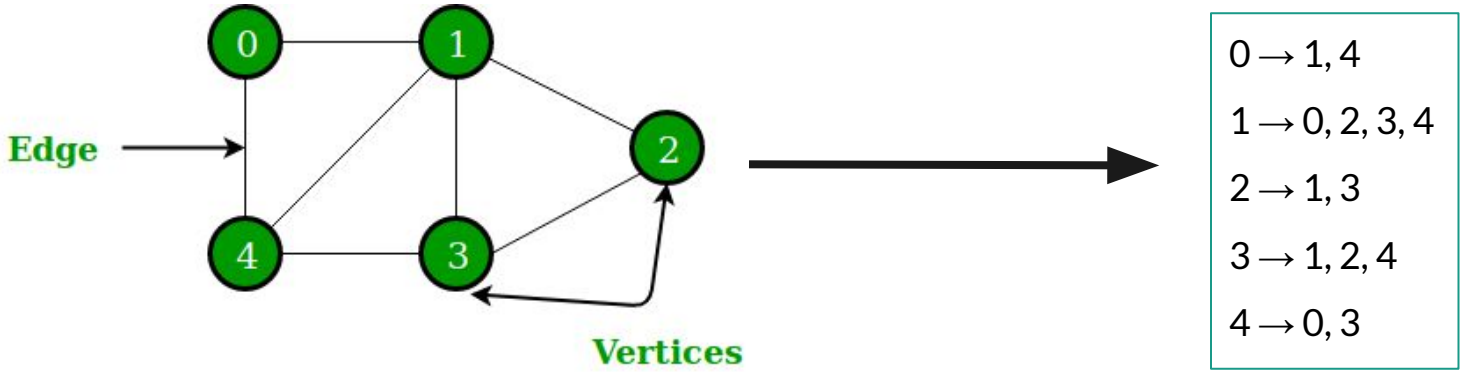
	0	1	2	3	4
0	false	true	false	false	true
1	true	false	true	true	true
2	false	true	false	true	false
3	false	true	true	false	true
4	true	true	false	true	false

- If the graph is directed, the matrix would be asymmetrical.
- If the graph is weighted, store the weight in matrix instead of true/false.
- Drawback: **Takes $O(V^2)$ memory even if number of edges is small.**

Adjacency List Representation

Graphs, in which the number of edges aren't a lot compared to the number of vertices, are called "Sparse Graphs". Adjacency List representation is necessary when dealing with Sparse Graphs.

For each of the V vertices, keep a list of vertices which are adjacent to it.

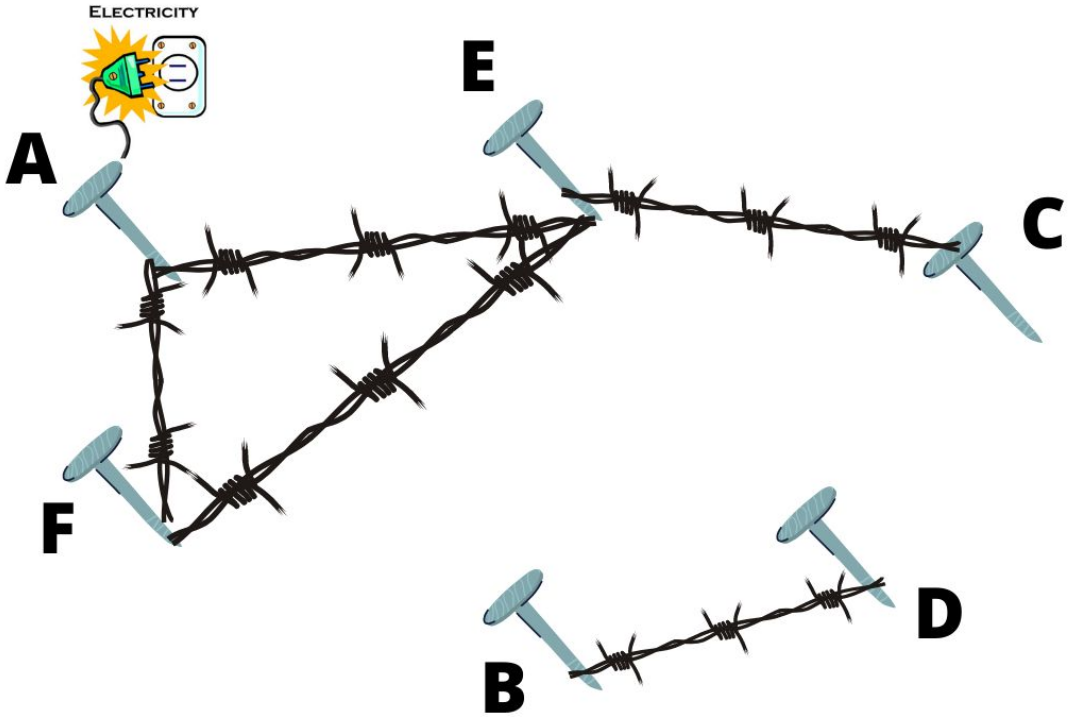


- If the graph is weighted, store a pair of {vertex, weight} for all outgoing edges.
- Advantage: **Takes O(V + E) memory.** (V lists and every edge adds 2 items to the lists in total)

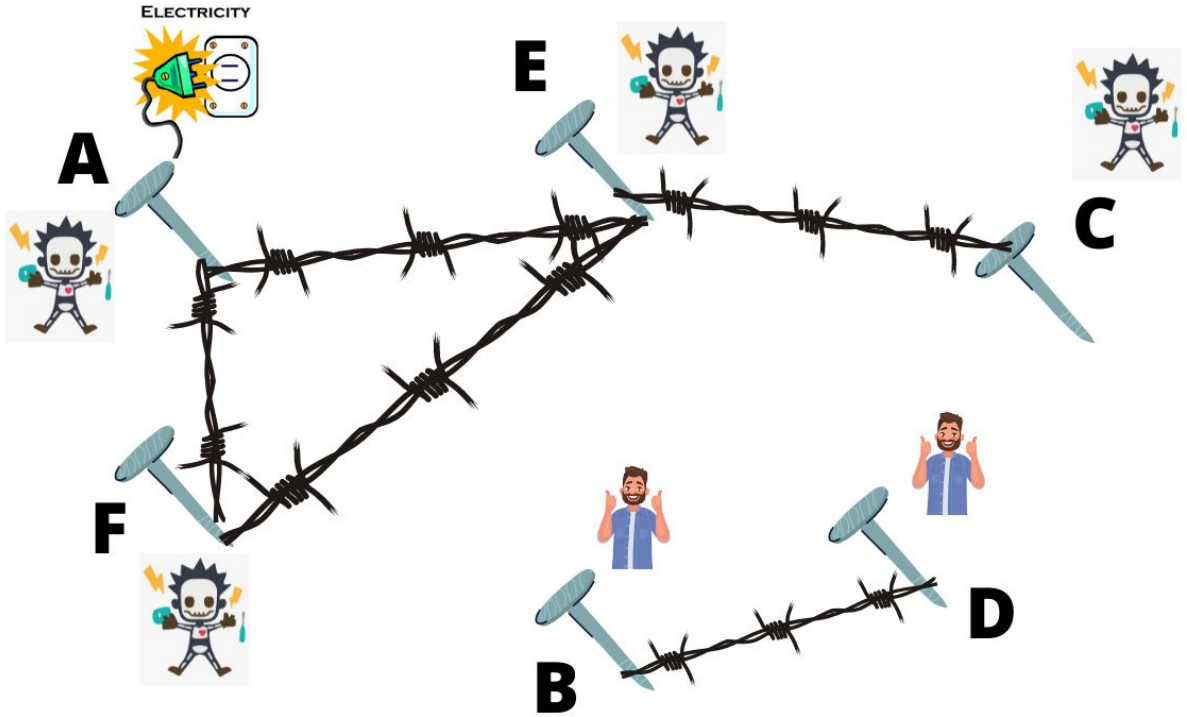
Note: The number of items in the adjacency list of a vertex is called the "degree" of that vertex.

Let’s solve a problem before learning more terminology:

Which of the metal nails will give you an electric shock upon touching?



This introduces the concept of “Connectivity” in a graph.

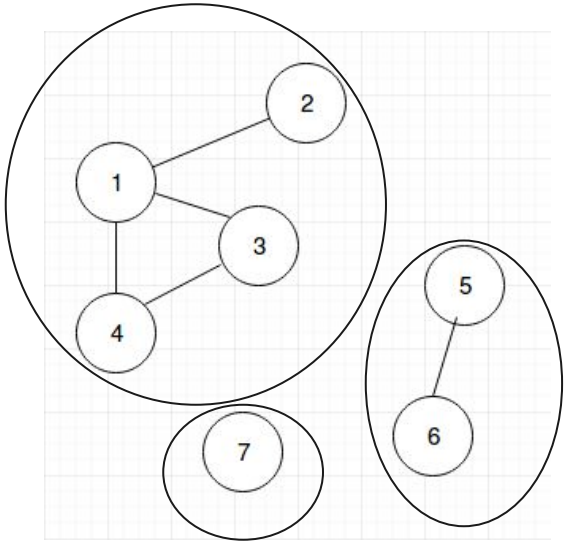


Terminology related to Connectivity

Path: A sequence of **distinct** vertices such as $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow ... \rightarrow A_n$, such that there is an edge from A_x to A_{x+1}

Connectivity: Vertex X is connected to vertex Y if there is at least 1 path from X to Y.

Connected Component: A *maximal* group of vertices such that each of them is connected to one other.



There are 3 connected components in this graph:

- 1,2,3,4 is a connected component.
- 5,6 is a connected component.
- 7 is a connected component.

Examples of paths:

- $2 \rightarrow 1 \rightarrow 4 \rightarrow 3$
- $5 \rightarrow 6$
- $3 \rightarrow 1 \rightarrow 2$

How to find a connected component?

Start from a vertex, all vertices adjacent to that vertex are in the same connected component.

All of the vertices adjacent to those are also in the same connected component and so on.

Let's think recursively:

```
visit(X):
        for each Y in adj_list[X]:
                visit(Y)
```

pseudocode of a very basic idea

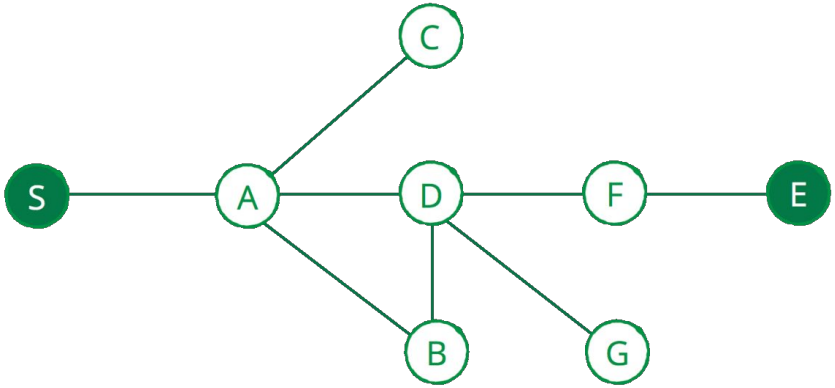
Ever been lost in a maze?
Try making some marks in the places you've already been,
so you don't get stuck in infinite loops.

DFS (Depth First Search)

```cpp
const int N = 100005;
vector<int> adj_list[N]; // list of vectors
// fill values in adj_list
bool visited[N];

void dfs(int current){
    visited[current] = true;
    for(int next_vertex : adj_list[current]){
        if(visited[next_vertex]) continue;
        dfs(next_vertex);
    }
}
```

It is called Depth First Search or Depth First Traversal, because we firstly go as deep as possible in a certain direction before going elsewhere



If we call **dfs(5)**, all the vertices in the same connected component as vertex **5**, will be visited once, and marked as visited.

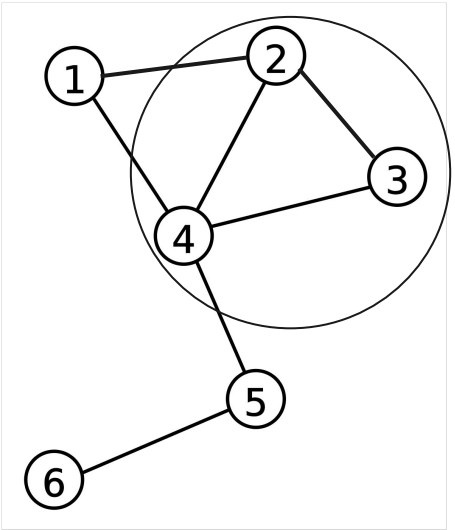
As per the problem, we might add more code to the dfs() function to perform various types of computations.

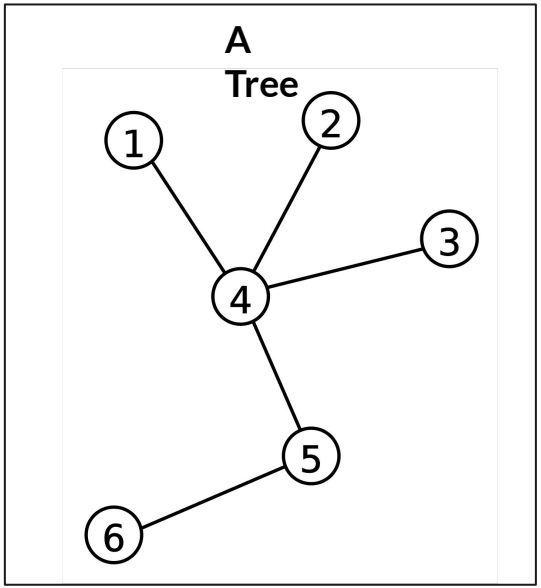
Time Complexity of DFS?

It is **O(V + E)**, where V is the number of vertices in the connected component and E is the total number of edges in that connected component, because every vertex is visited only once, and every edge is considered only twice - once from while visiting each of the end points of the edge.

Cycles in a Graph and Tree

- A cycle is like a path that starts and ends at the same vertex.
- For example, **2 → 3 → 4 → 2**
- A Connected Graph without cycles is called Tree. (A Disconnected Graph without cycles is called Forest).

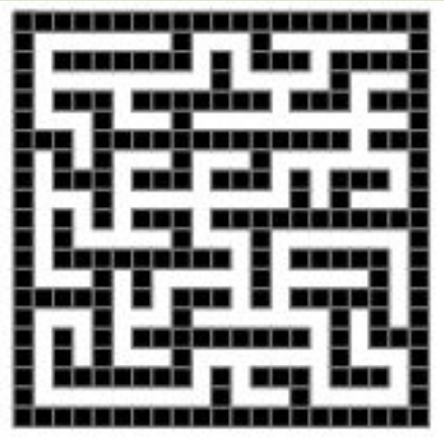




Try out these problems:

- Given a graph, count the number of connected components in the graph.
- https://www.spoj.com/problems/ABCPATH/ (Problem on a directed graph)
- Count Connected Components in a Grid
- Find Path in a Grid
- Find Cycle in Graph (Explained in video)

Hint: Grid/Maze is also a graph. Every square in the grid is a vertex and implicitly there are upto 4 edges. L,R,U,D



When working with grids, you do not need to keep an adjacency list, you can find all 4 adjacent vertices easily:

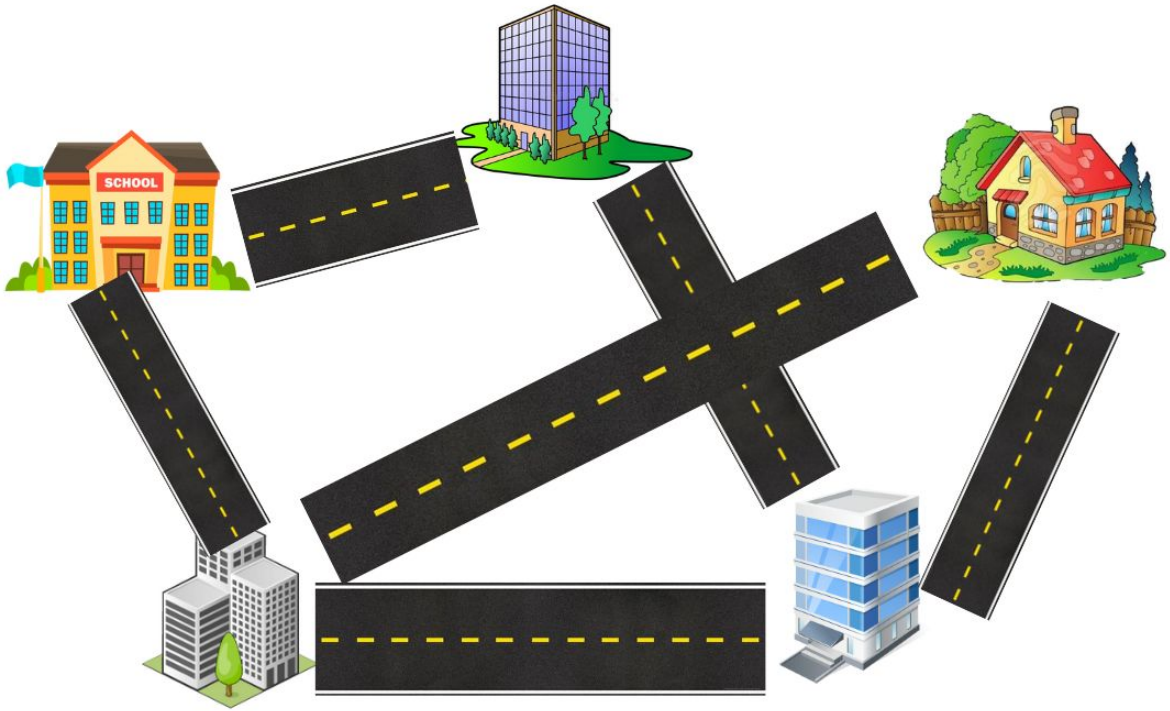
For (x,y) the adjacent vertices are:

- (x , y+1)
- (x+1 , y)
- (x-1 , y)
- (x , y-1)

```
// You can store the 4 graph
// moves in this way
int dx[4] = {1,0,-1,0};
int dy[4] = {0,1,0,-1};
```

For our next algorithm, let's look at another very common problem.

Minimum number of roads to get from house to school



Shortest Path Problem in Unweighted Graph

In general, the problem from the last slide is called “Shortest Path Problem”. And since we only care about the number of roads and not the length of the roads, we consider this as an unweighted graph.

The general idea to solve this problem is pretty intuitive:

- Start with the source vertex, the “distance” for that is 0.
- All vertices adjacent to source have distance 1.
- All unvisited vertices which are adjacent to at least one of vertices with distance 1 have distance 2.
- All unvisited vertices which are adjacent to at least one of vertices with distance x have distance (x+1).
- Keep repeating previous step until the destination is found.

But how do we implement this in a computer program?

We need to learn a data structure called queue for this.

Prerequisite: Queue

Queue is a FIFO data structure. First In First Out.

The elements that are pushed into the queue first, are popped out first.

You do not need to implement queue, **it is already there in STL C++**

```
queue<int> q;
q.clear(); // clear the queue
q.size(); // get the size of the queue
q.empty(); // returns true if the queue is empty
q.push(x); // insert x into the queue
q.front(); // returns the value in the queue that was pushed earliest
q.pop(); // erase the front from the queue
```

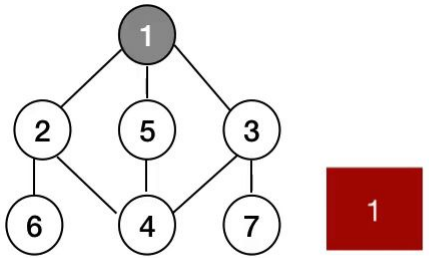
Push and Pop are also known as enqueue and dequeue.

BFS (Breadth First Search)

```
for(int i = 1; i <= n; i++)
    dist[i] = INF;

dist[source] = 0;
queue<int> q;
q.push(source);
while(!q.empty()){
    int current = q.front();
    q.pop();
    for(int next : adj_list[current]){
        if(dist[next] == INF){
            dist[next] = dist[current] + 1;
            q.push(next);
        }
    }
}
// dist[x] contains distance from source to x
```

We are using a queue because of the FIFO principle. The vertex which is seen first is nearer to the source, so it is best if it is popped first, this guarantees that no vertex with a higher distance from the source is popped before a vertex with a lower distance.



Time Complexity of BFS

It is also **O(V + E)**. The reasoning is similar to DFS, a vertex is visited only once and an edge is considered only twice at most.

Try out these problems:

- Find and print the shortest length path (Hint: store for each vertex its parent)
- Minimum moves of knight on chessboard to reach from 1 point to another
- Reach a destination before competitors
- Minimum possible moves to win Snakes and Ladder
- Will BFS work if the graph is directed?
- Will BFS work if the graph is weighted?

Few more problems (not necessarily using DFS/BFS):

- Minimum number of new edges to make the graph connected
- Check if the graph has a cycle of odd length.
- Check if the graph can be divided into 2 groups such that there is no edge between vertices from the same group (Hint: Read about bipartite graphs)

That's it folks.

In the future, we'll talk about a bunch of other things, such as Trees, Weighted Graphs, Bipartite Graphs, Directed Graphs, and much more.