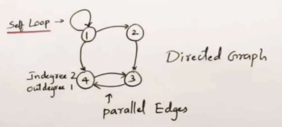
Graphs

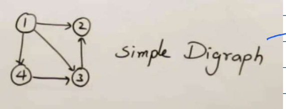
Monday 15 July 2024 11:40 AM

## Basic Terminology

→ Collection of vertices & edges.

$G = (V, E)$

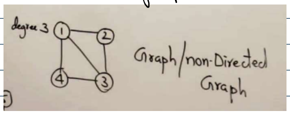


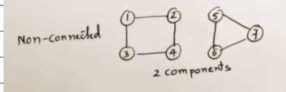


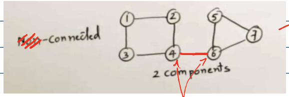
→ Doesn't contain self loops or parallel edges.

→ Two vertices connected by an edge are called ADJACENT VERTICES

→ In undirected graph : Total Degree = 2 X E

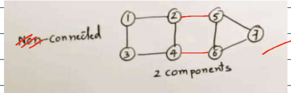




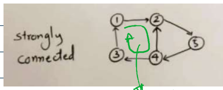


→ Graph has connected components

articulation points - vertices whose removal will split the graph into multiple components



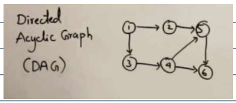
→ Bi-Connected components
There are many components but they are strongly Connected



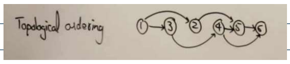
→ There is a PATH b/w any pair of vertices

set of vertices b/w a pair of vertices [neither vertices nor edges are repeated]

A sequence of distinct vertices



→ No cycles



→ only possible in DAG

→ edges may have weights if not assume unit weight (i.e. 1) (cost of traversing through a given edge)

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

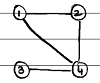
### Representation of Undirected Graph

Adjacency Matrix

Adjacency List

Compact list

## Representation of Undirected Graph



$G=(V,E)$

$|V|=n=4$

$|E|=e=4$

### Adjacency Matrix

$$\begin{array}{c|cccc} A & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & \textcircled{1} & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

Indicates an edge

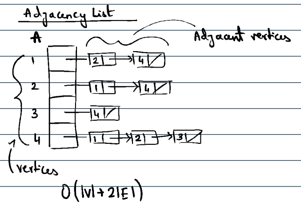
$4 \times 4$

$O(n^2)$

→ given a weighted graph we can fill A[i][j] with the corresponding weight instead of 1.

Cost Adjacency Matrix

### Adjacency List



$O(|V|+2|E|)$

$= O(n+2e)$

→ Here we need to modify the node structure to include weights

```
struct Neighbour {
    int vertex
    int weight
}
```

cost Adjacency List

→ Can use a Array of integer vectors in C++

vector<int> adj [size]

$O(2E)$

### compact list

$|V|+2|E|+1+\textcircled{1}$

Lets assume we start index from 1*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| / | 6 | 8 | 10 | 11 | 14 | 2 | 4 | 1 | 4 | 4 | 1 | 2 | 3 |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

$\underbrace{1\ 2\ 3\ 4\ 5}_{\text{vertices}}$ $\underbrace{6\ 7}_{1}$ $\underbrace{8\ 9}_{2}$ $\underbrace{10}_{3}$ $\underbrace{11\ 12\ 13}_{4}$

$|V|+2|E| = n+2e \rightarrow O(n)$

## Representation of Directed Graphs

→ Adjacency Matrix $O(n^2)$

→ Adjacency List $O(|V|+|E|) \rightarrow O(n) / O(E)$ can use Array of vectors (outgoing edges)

→ We can have an Inverse Adjacency list which stores incoming edges for each vertex.

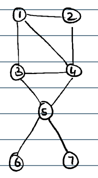
## Breadth First Search (level order)

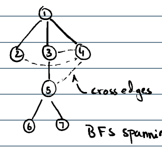
① Start traversal from any vertex

② visit a vertex, explore all its neighbors completely

→ Neighbors can be visited in any order

1. Visiting
2. Exploring





BFS spanning Tree

$$\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$
Visited = { 0, 0, 0, 0, 0, 0, 0, 0 }

Adjacency List

|  |  |
|---|---|
| 0 | { } |
| 1 | {2, 3, 4} |
| 2 | {1, 4} |
| 3 | {1, 4, 5} |
| 4 | {1, 2, 3, 5} |
| 5 | {3, 4, 6, 7} |
| 6 | {5} |
| 7 | {5} |

BFS: 1 2 3 4 5 6 7

one of many possibilites

work done : visiting all nodes

∴ $O(V)$

analytical

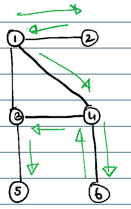
Program $O(V+2E)$

→ A Queue will help us with BFS

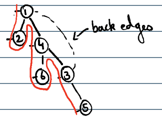
allows us to start traversal from any vertex

## Depth first Search

# Depth first Search



Lets start from 1:



O(V)
↑ analytical

DFS: 1 2 4 6 3 5

→ When we visit a node, suspend exploration of previous node
→ Recursion → Need to have visited array as static variable
↳ Internally uses a stack

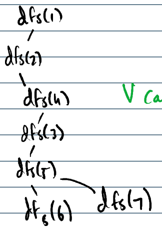
V ← # of calls

But we will scan through the whole list i.e. 2E

e.g.

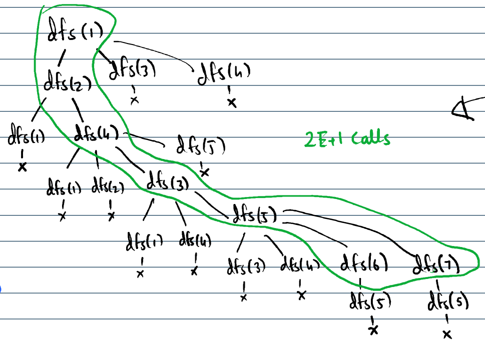
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| visited: | 0 | ~~0~~ 1 | ~~0~~ 1 | ~~0~~ 1 | ~~0~~ 1 | ~~0~~ 1 | ~~0~~ 1 | ~~0~~ 1 |

| | |
|---|---|
| 0 | { } |
| 1 | {2, 3, 4} |
| 2 | {1, 4} |
| 3 | {1, 4, 5} |
| 4 | {1, 2, 3, 5} |
| 5 | {3, 4, 6, 7} |
| 6 | {5} |
| 7 | {5} |

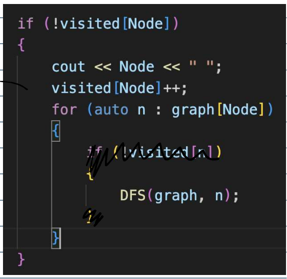


V calls

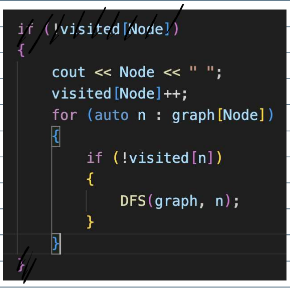
OR ← depending on condition in For loop



2E+1 calls



```
if (!visited[Node])
{
    cout << Node << " ";
    visited[Node]++;
    for (auto n : graph[Node])
    {
        ~~if (!visited[n])~~
        ~~{~~
            DFS(graph, n);
        ~~}~~
    }
}
```



```
~~if (!visited[Node])~~
~~{~~
    cout << Node << " ";
    visited[Node]++;
    for (auto n : graph[Node])
    {
        if (!visited[n])
        {
            DFS(graph, n);
        }
    }
~~}~~
```

Work done: Call made for each Node & traversing through the entire list

O(V+2E)