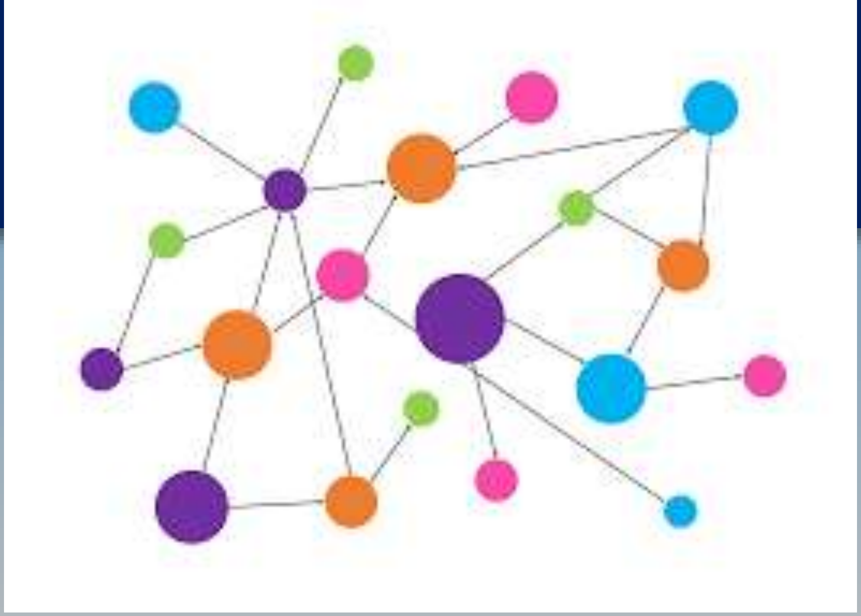
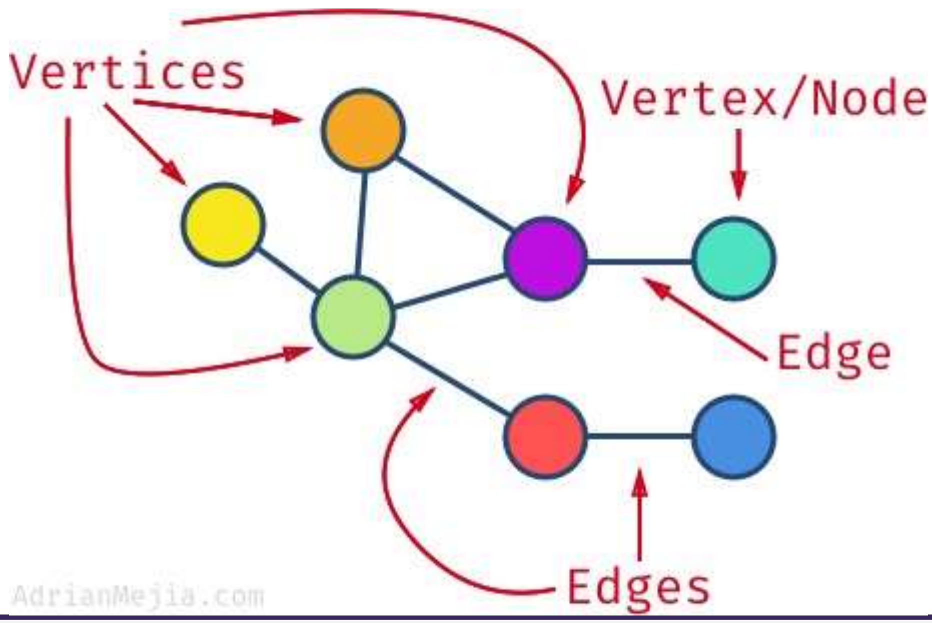
Graphs



What is a graph?

A **graph** is a **mathematical structure** for representing relationships.

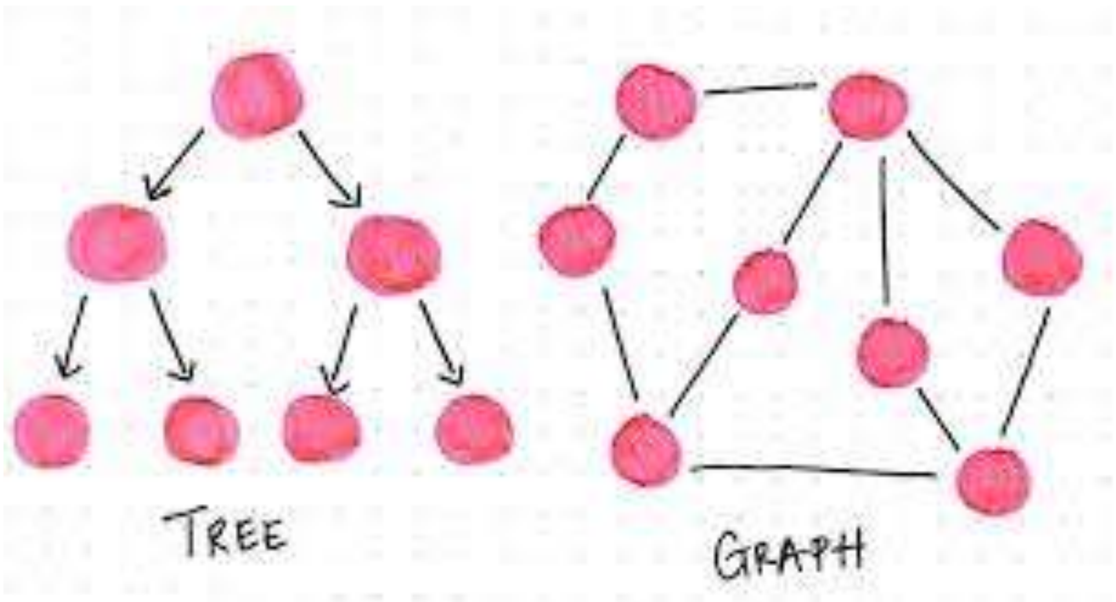
Graph consists of a set of **nodes** (*vertices*) and a set of edges between the vertices



Edges describes relationships among the vertices

Why Graphs?– When we have trees

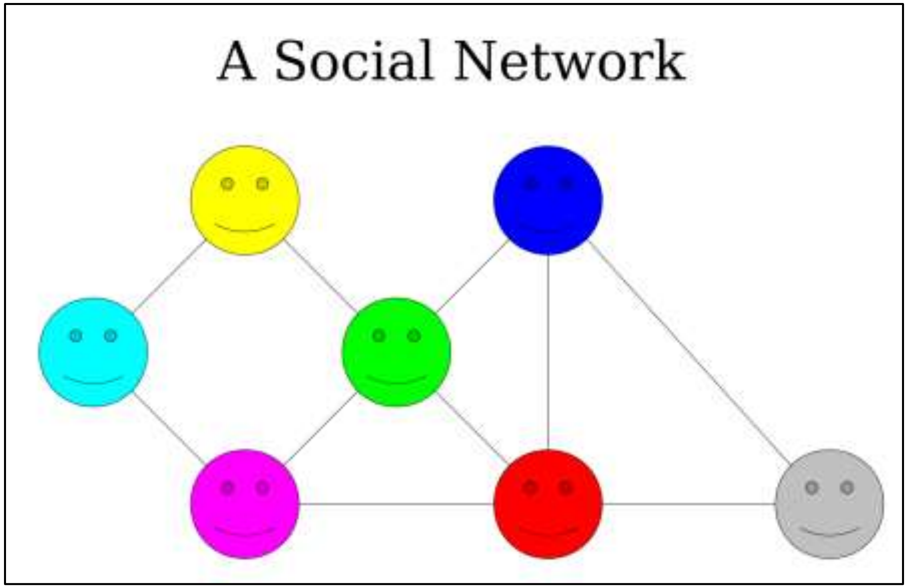
- Difference in a tree and a graph



Graph is a generalization of a tree

Why Graphs?– When we have trees

- **Limitation of trees ?**
  - represent relations of a hierarchical type only (parent-child).
  - Other relations are represented indirectly, such as a sibling.



A generalization of a tree, a *graph,* is a data structure in which this limitation is lifted.

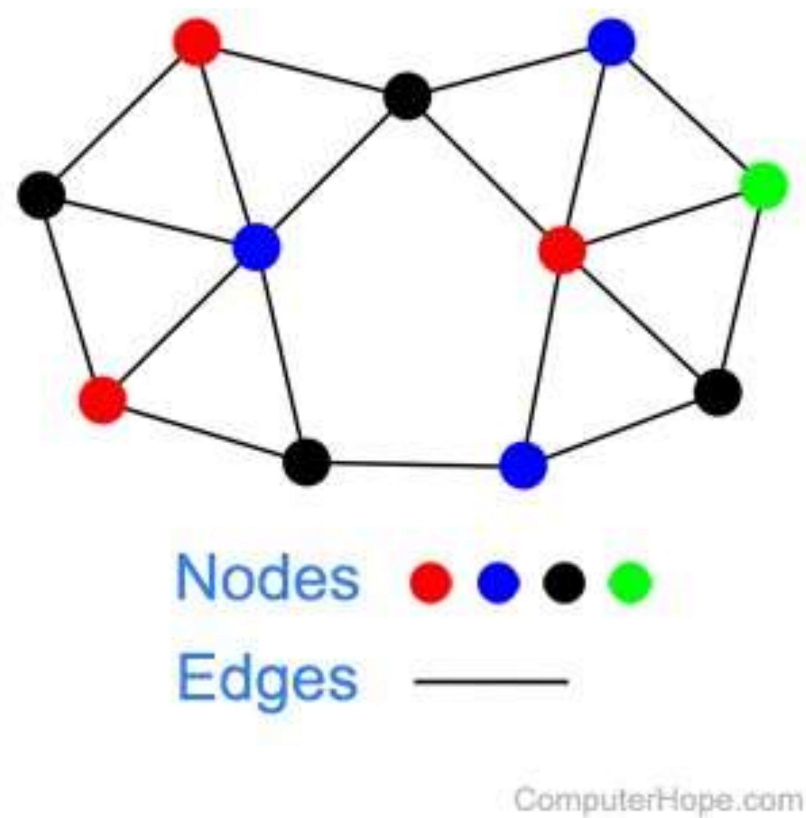
Formal definition of graphs

A *Simple* graph *G* is defined as **G=(V,E)**

V: a finite, nonempty set of vertices

E: a set of edges (pairs of vertices)

The number of vertices and edges is denoted by $|V|$ and $|E|$, respectively



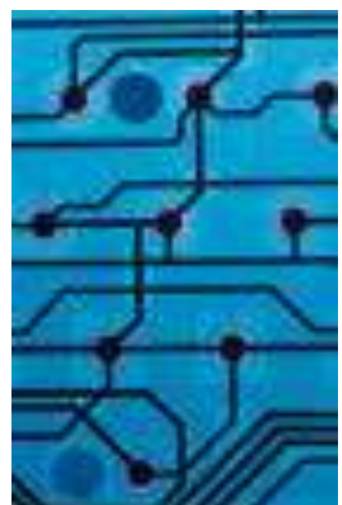
Why Graphs?

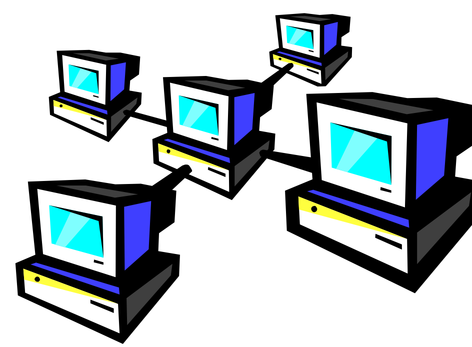
- We study graphs because
  - many problems can be modeled in terms of graphs
  - we have many off-the-shelf graph algorithms that we apply if we're able to formulate a problem as a graph problem.

Applications

- Graphs are versatile data structures that can represent various situations and events from diverse domains.

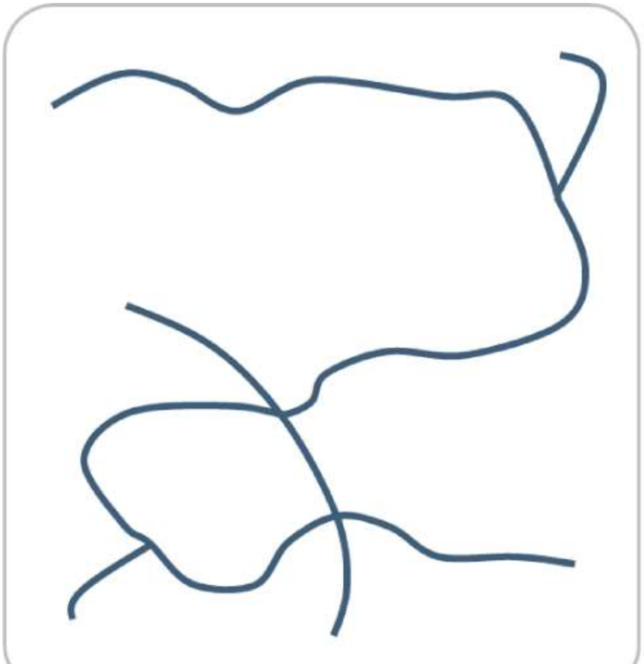
- Graph theory has grown into a sophisticated area of mathematics and computer science in the last 200 years since it was first studied.



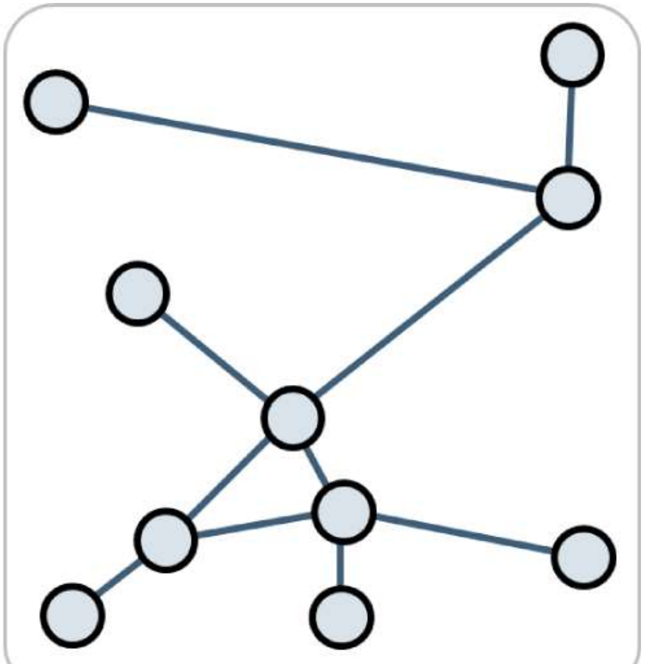


Applications–Road Network

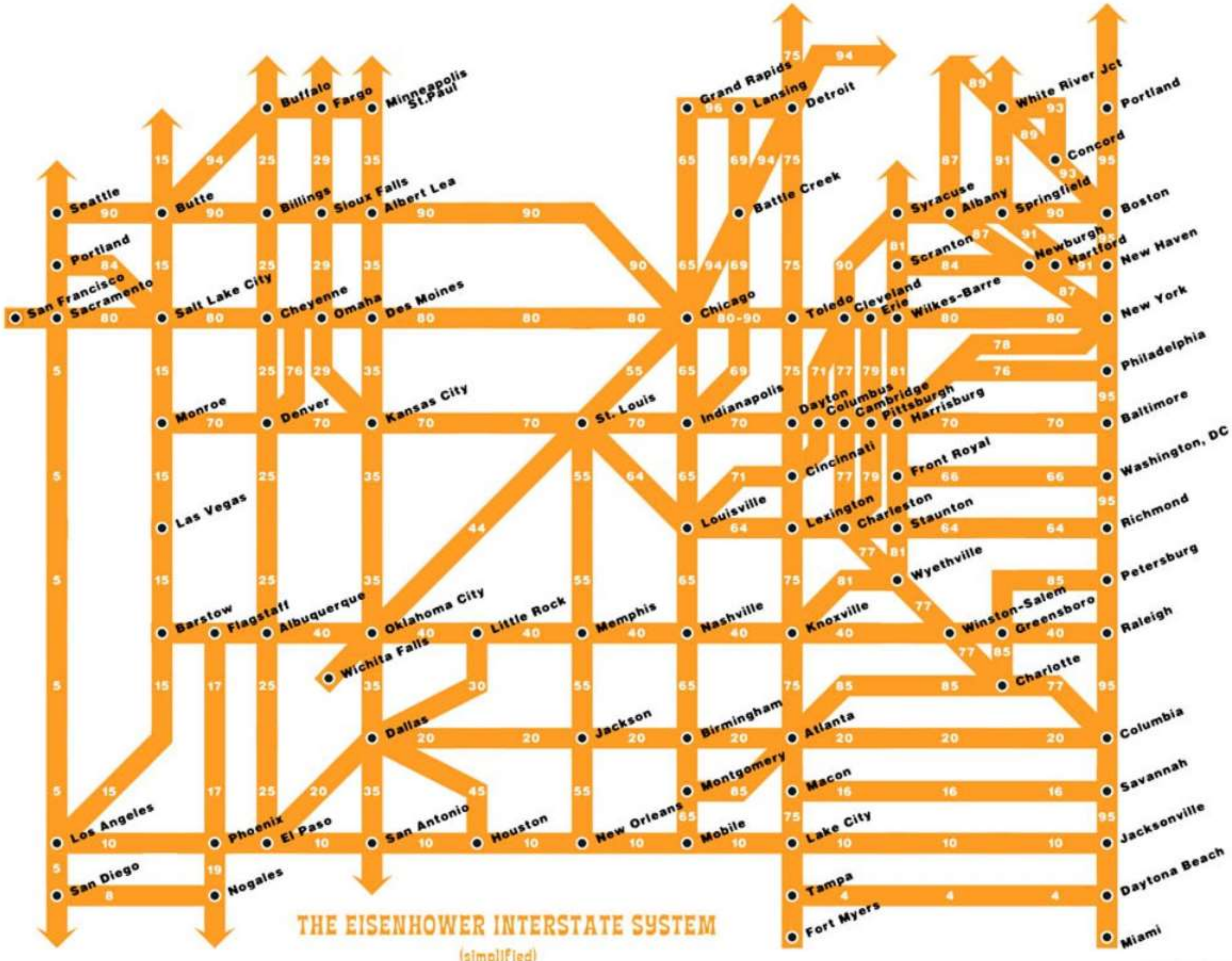
- Traffic flow can be modeled by a graph.
  - Each street intersection represents a vertex,
  - Each street is an edge.
  - Find the shortest route or use this information to find the most likely location for bottlenecks



Real Network



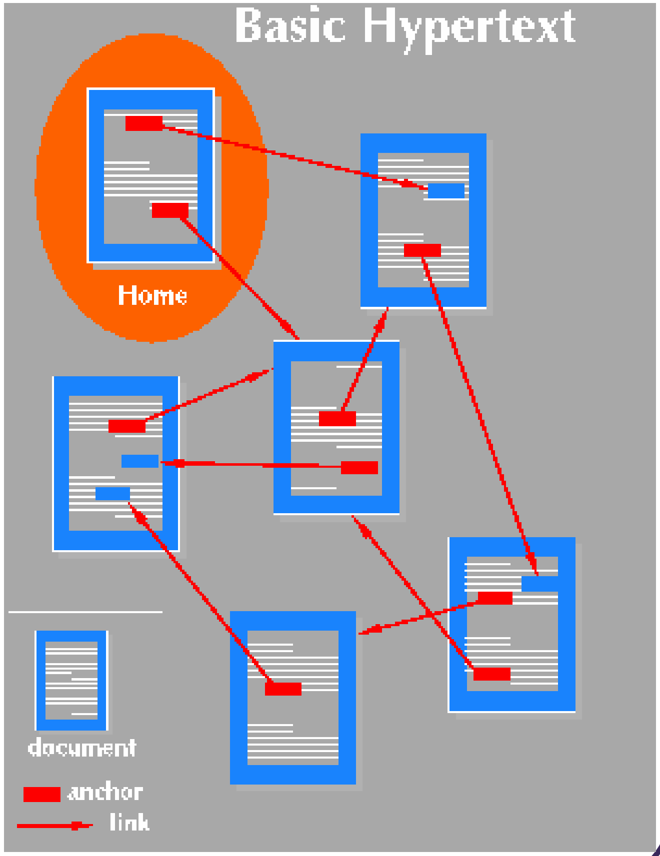
Graph Representation



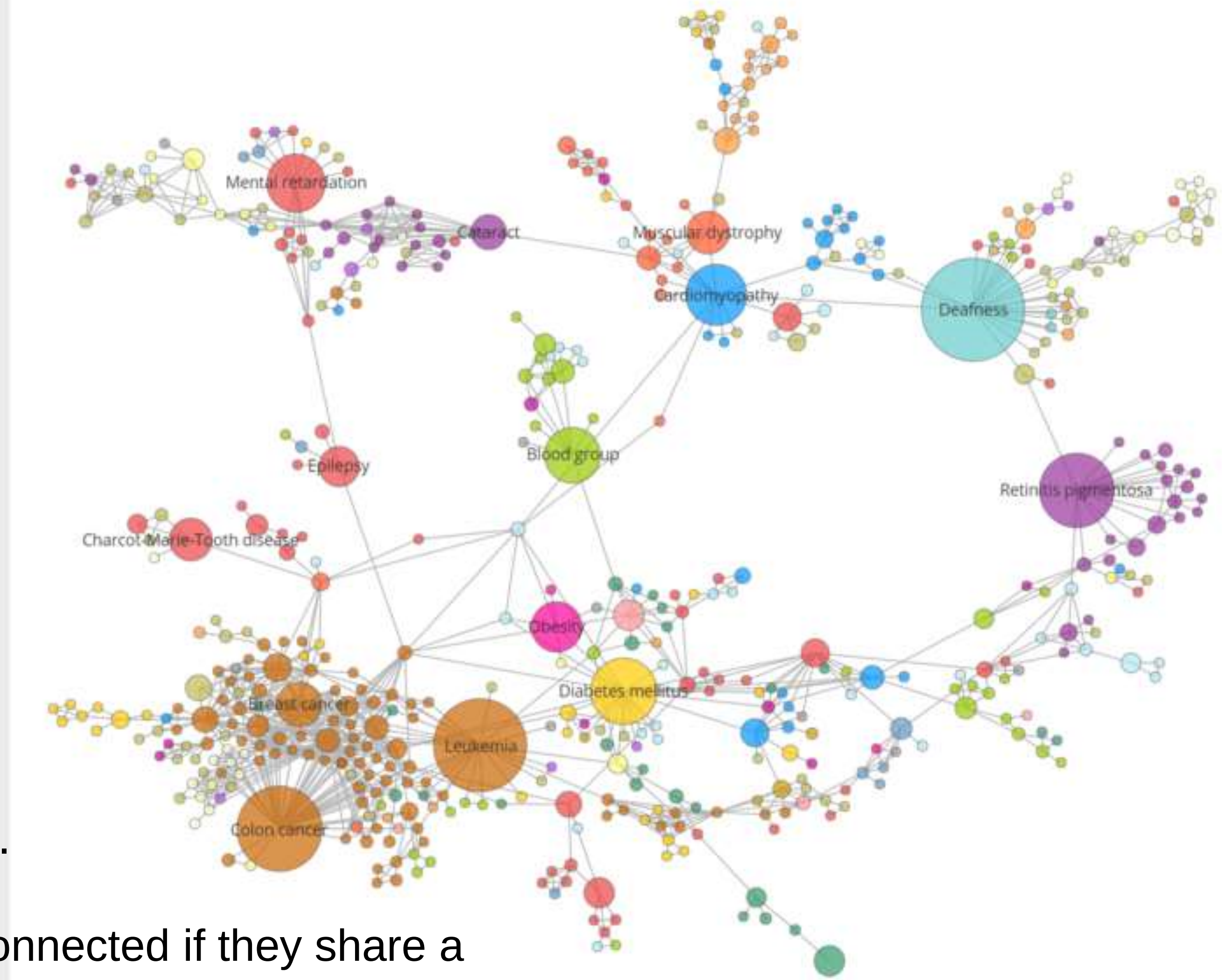
://strangemaps.files.wordpress.com/2007/02/fullinterstatemap-web.jpg

Hyperlink graph

- The best-known example is the link graph of the web,
  - each web page is a vertex, and
  - each hyperlink a directed edge.
- Link graphs help
  - analyze relevance of web pages,
  - the best sources of information,
  - Good link sites.
- *Document link graphs.*



Human disease network

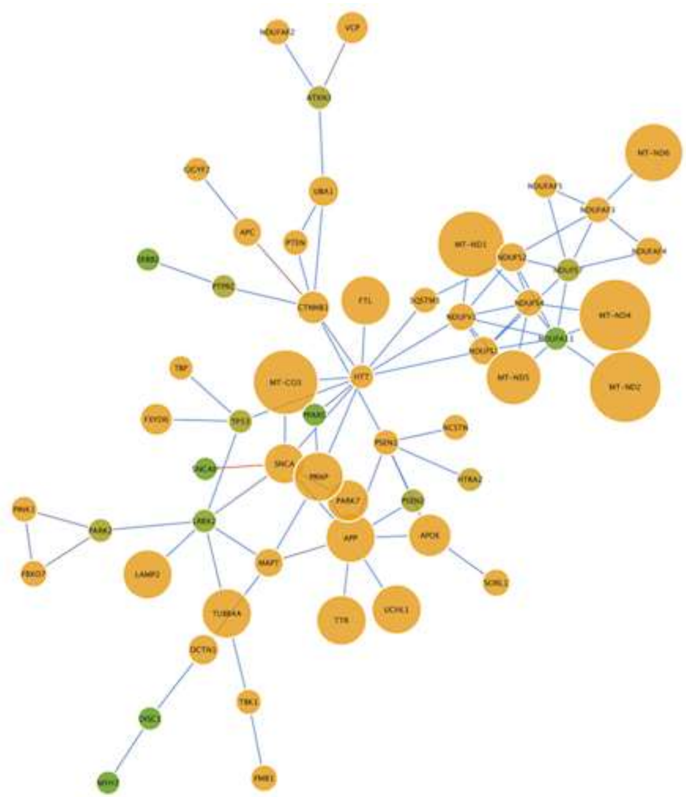


Nodes are diseases.

Two diseases are connected if they share a genetic component.

Protein-protein interactions graphs

- Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell.
  - These graphs can be used, for example, to study molecular pathways
  - Humans have over 120K proteins with millions of interactions among them



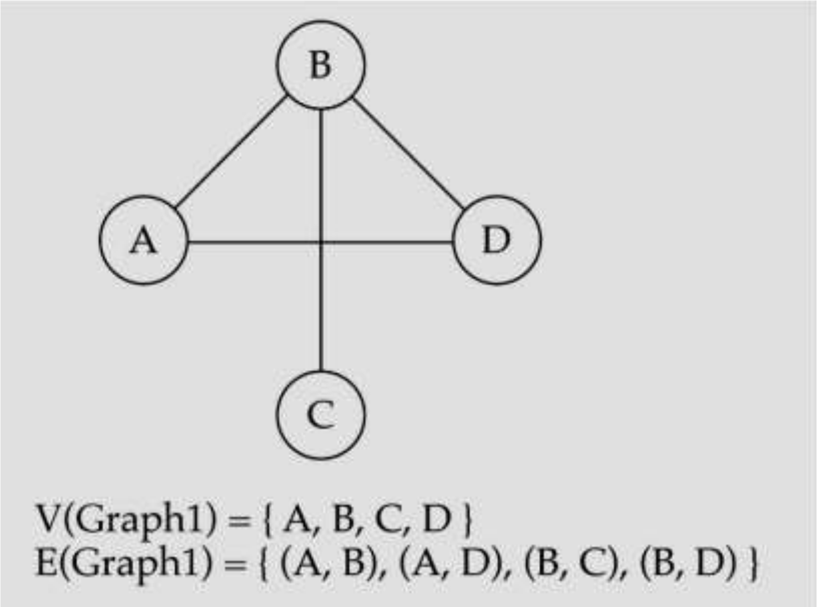
Social Network

Graph Terminology

Undirected Graph

- When the edges in a graph have no direction, the graph is called *undirected*

undirected graph

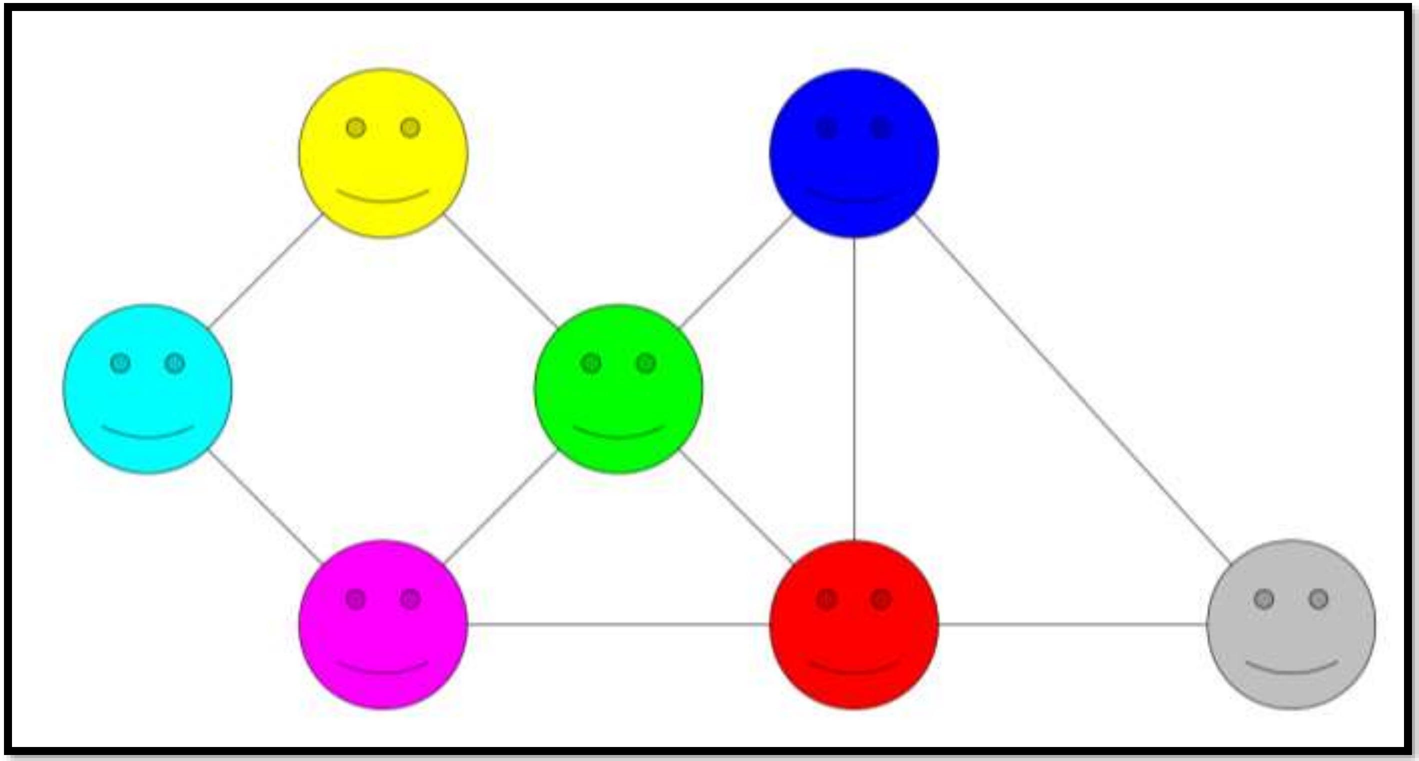


e.g., (v_0, v_1) and (v_1, v_0) represent the same edge

The order of vertices in E **is not** important for undirected graphs!!

Undirected Graph

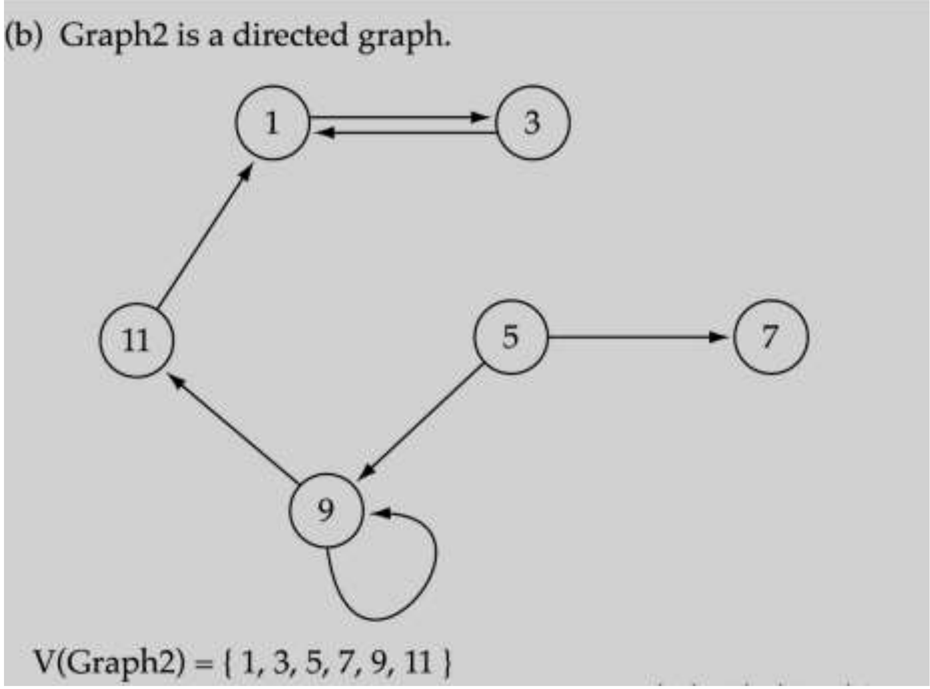
- Example: Friendship



Directed graphs (digraph)

- When the edges in a graph have a direction, the graph is called *directed*.

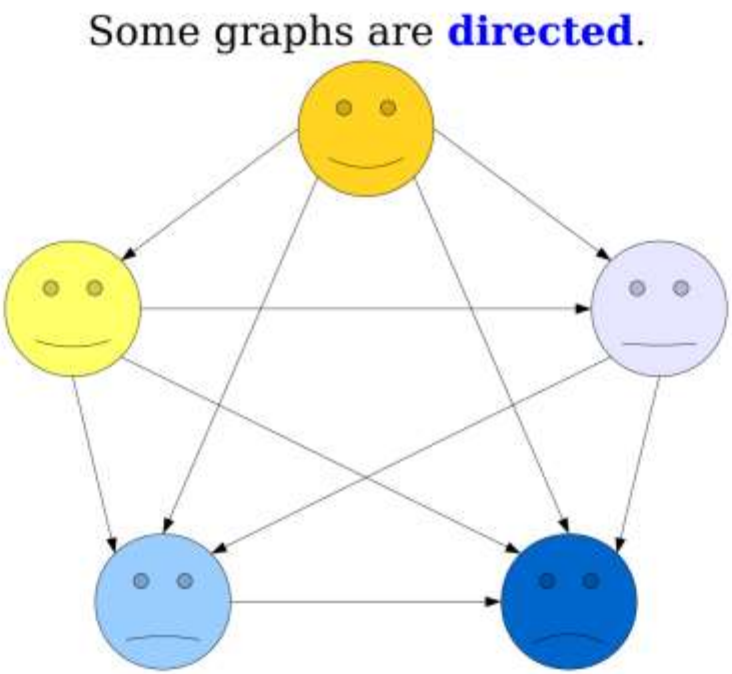
The order of vertices in E **is** important for directed graphs!!



E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7)

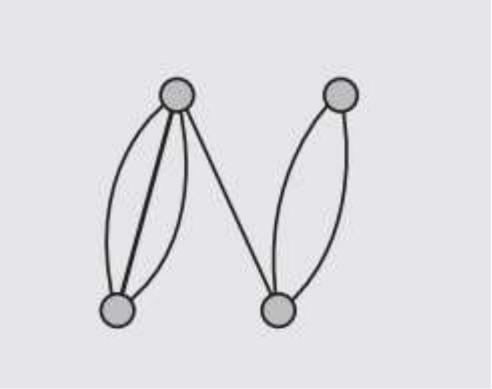
Directed Graph

- Example: Twitter graph of who follows whom.
  - It can be used to determine
    - how information flows,
    - how topics become hot,
    - how communities develop

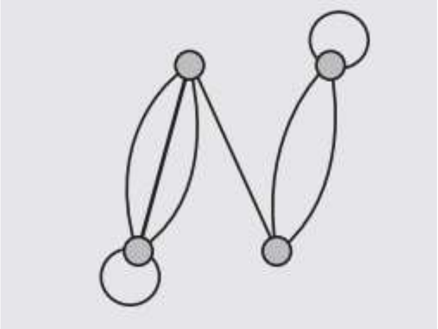


Multi-Graph

A *multigraph* is a graph in which two vertices can be joined by multiple edges but no self-loop.

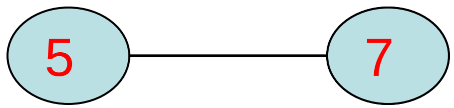


Pseudo graph is a multigraph which allows self-*loops* to occur. A vertex can be joined with itself by an edge.

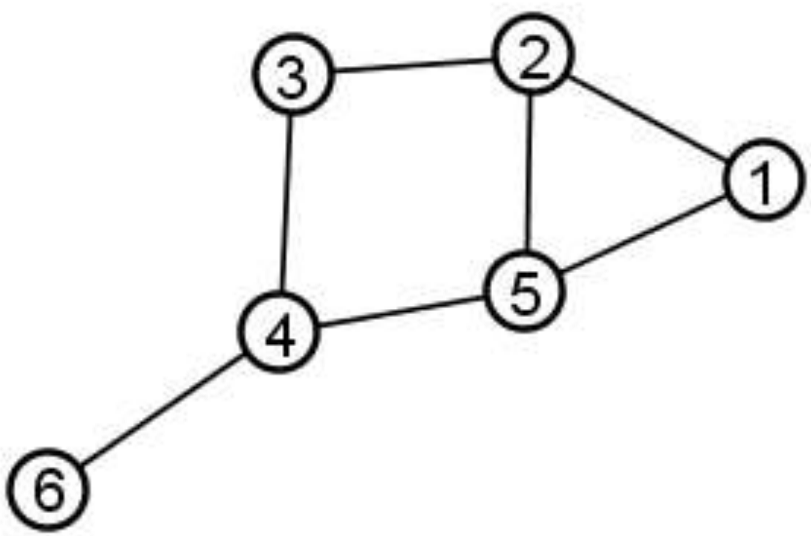


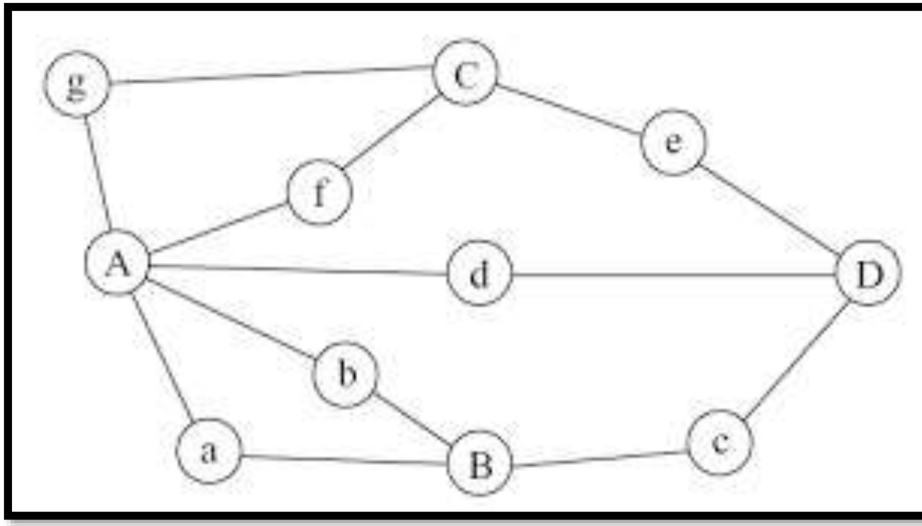
Adjacent nodes

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



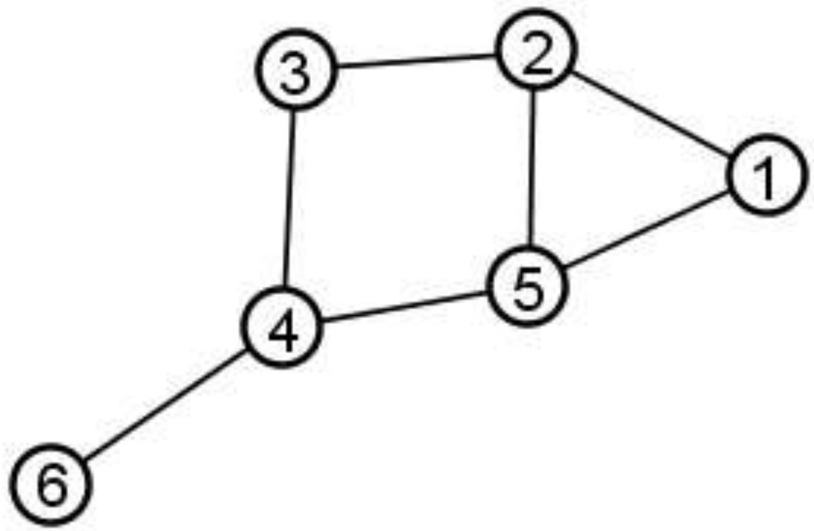
5 is adjacent to 7





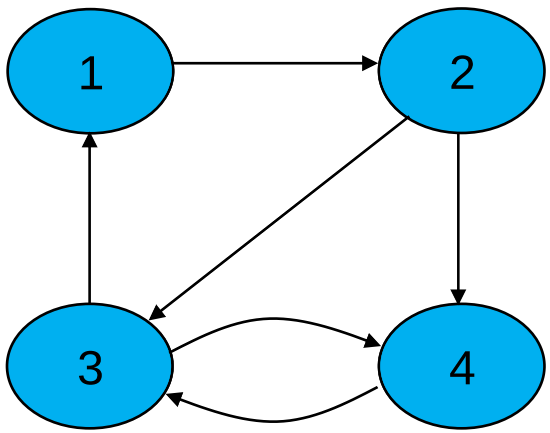
Degree of a node

The ***degree*** of a vertex *v*, *deg*(*v*), is the number of edges incident with *v*.



Degree of a node in Directed graph

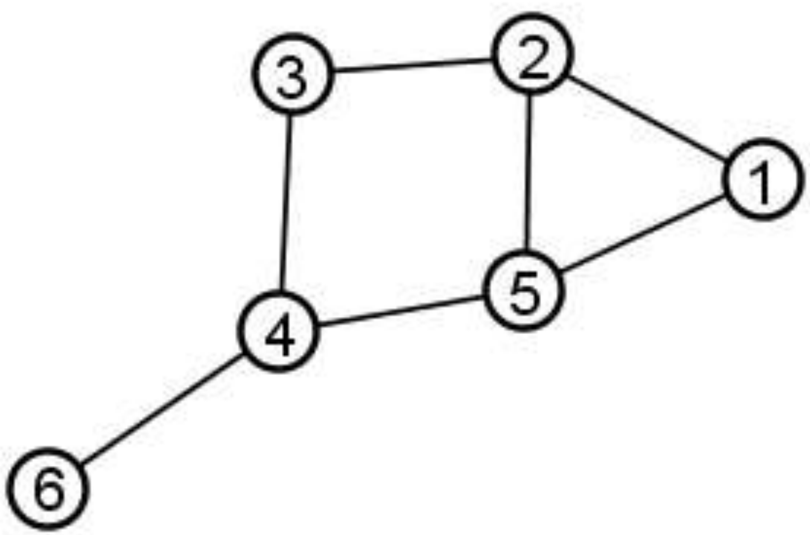
The *indegree* of a vertex *v*, is the number of edges incident to *v*.
The out**degree of vertex v,** is the number of edges incident from v.

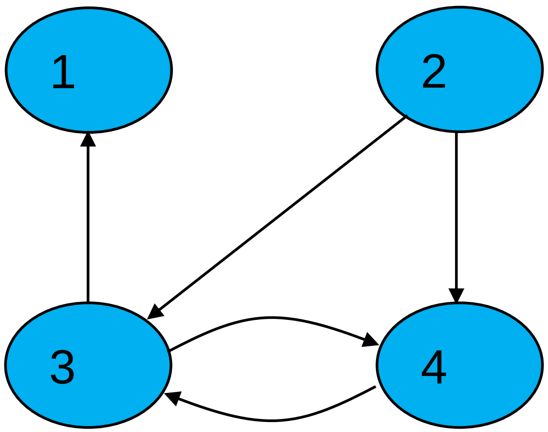


Path

Path: a sequence of vertices that connect two nodes in a graph.

The **length** of a path is the number of edges on the path.

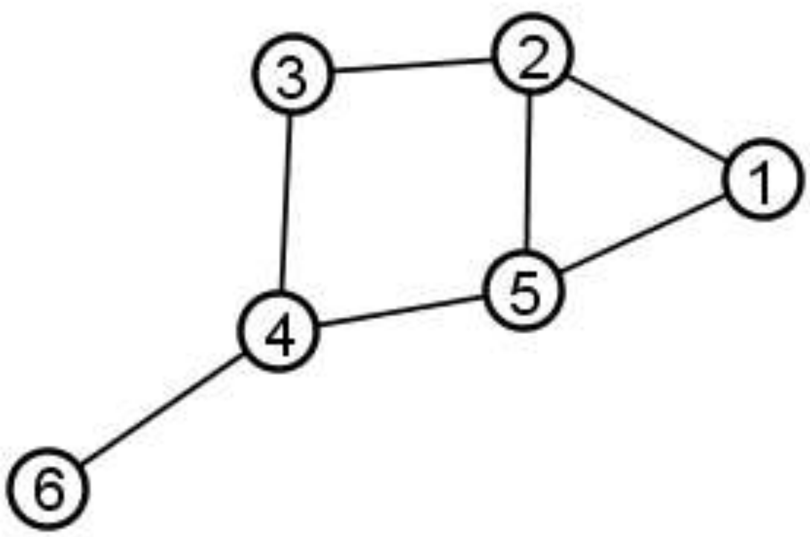


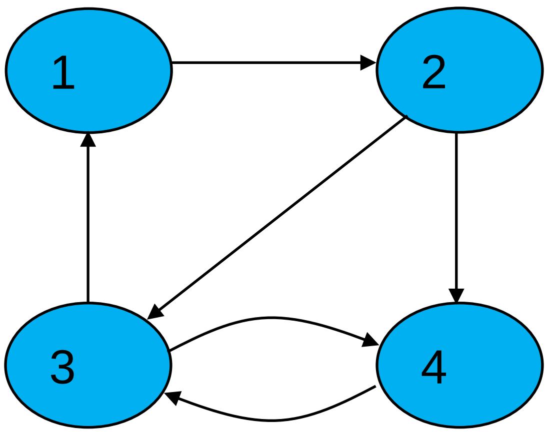


Cycle

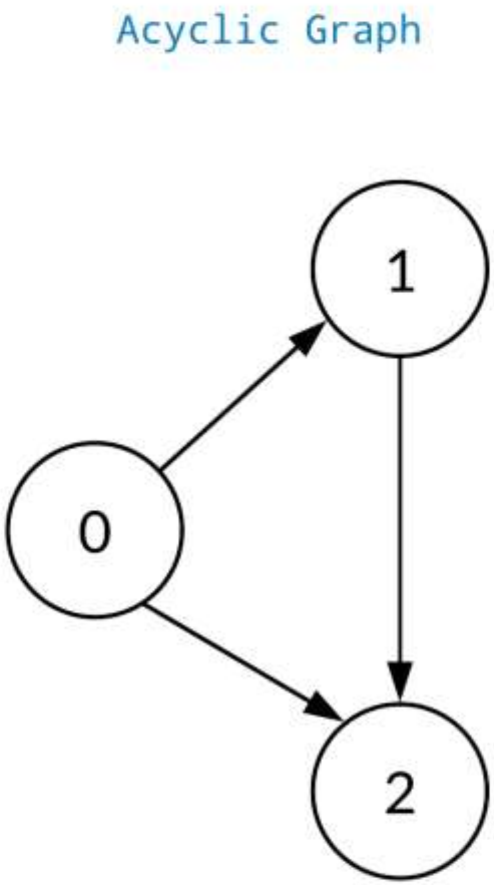
A **simple path** is a path such that all vertices are distinct, except that the first and last could be the same

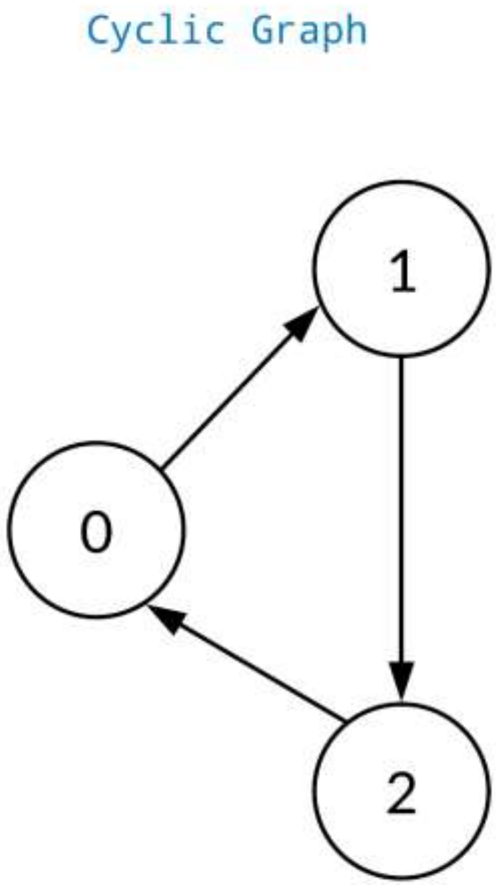
A **cycle** in a graph is a path of length at least 1 such that $v_1 = v_N$





Acyclic Graph





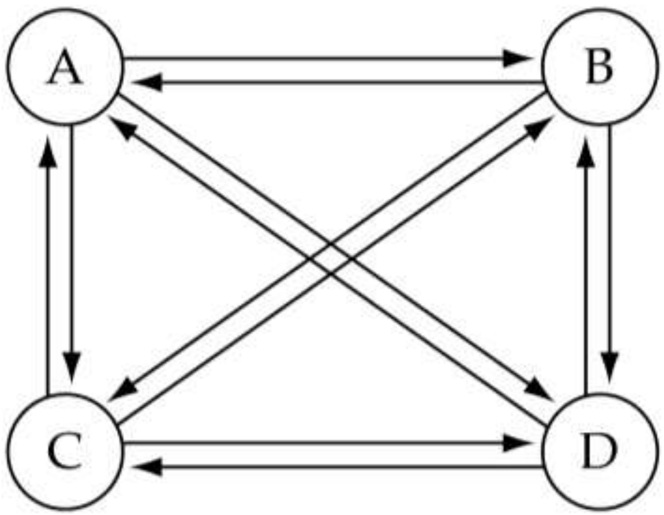
Trees

- Trees are special cases of graphs!!
- A *tree* is an undirected graph *G* that satisfies any of the following equivalent conditions:
  - *G* is connected and has no cycles.
  - *G* is acyclic, and a simple cycle is formed if any edge is added to *G*.
  - *G* is connected, but would become disconnected if any single edge is removed from *G*.

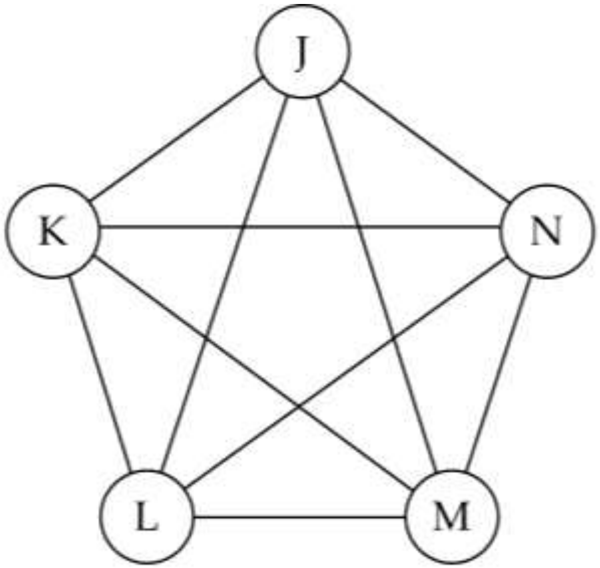
- A directed graph is **acyclic** if it has no cycles. A directed acyclic graph is sometimes referred to by its abbreviation, **DAG.**

Complete graph

Complete graph: a graph in which every vertex is directly connected to every other vertex



(a) Complete directed graph.

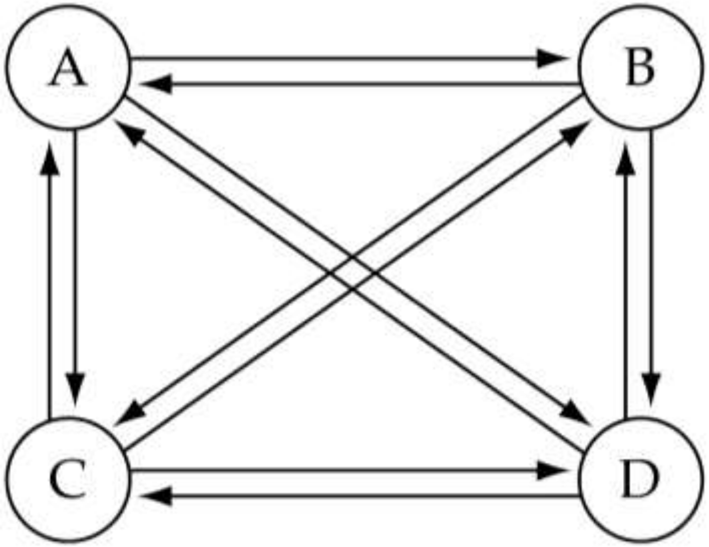


(b) Complete undirected graph.

Complete directed graph

- What is the number of edges E in a complete directed graph with V vertices?

$$E = V * (V-1)$$



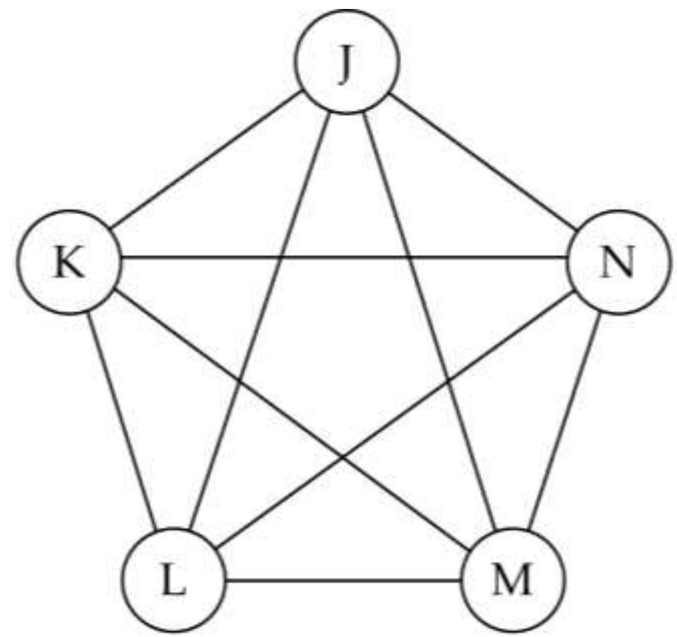
(a) Complete directed graph.

Complete undirected graph

- What is the number of edges E in a complete undirected graph with V vertices?

The number of edges in such a graph $|E| = \binom{|V|}{2} = \frac{|V|!}{2!(|V|-2)!} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$

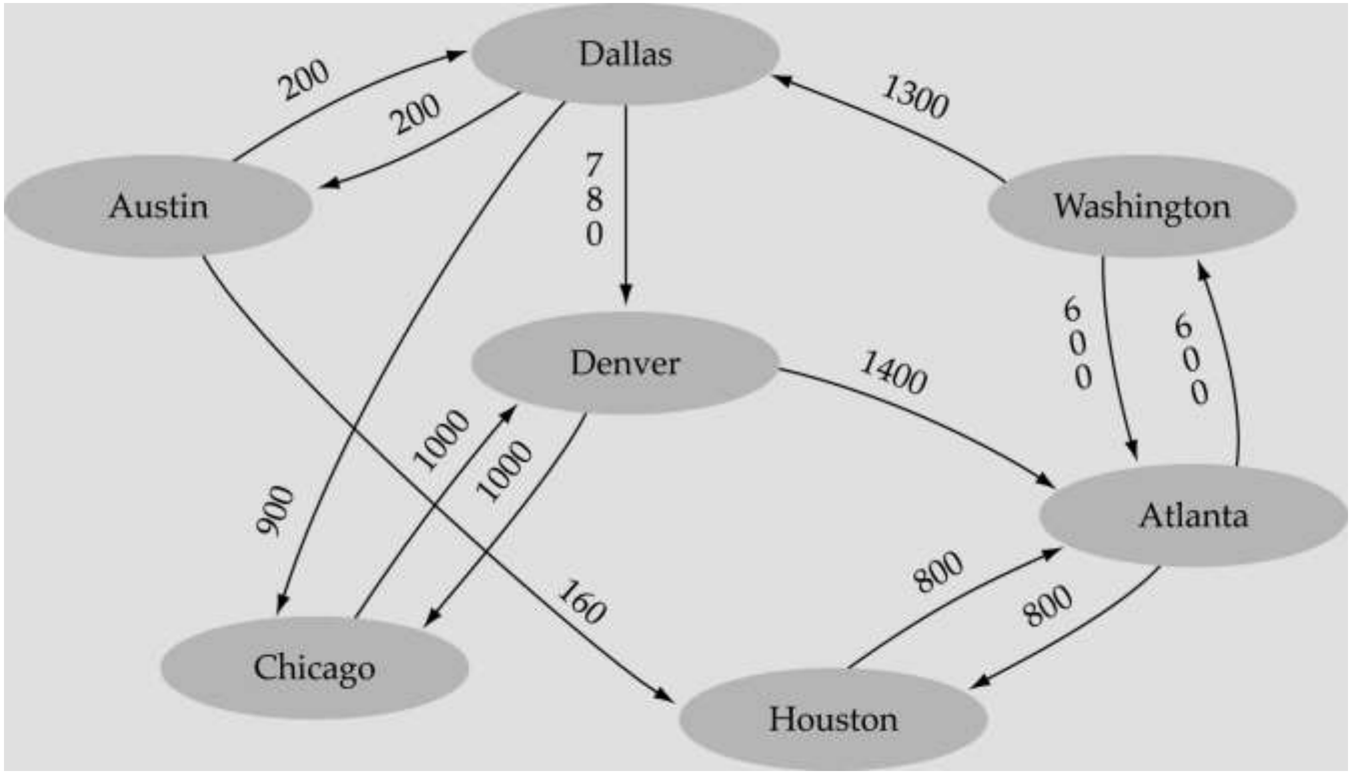
E=V* (V-1) / 2



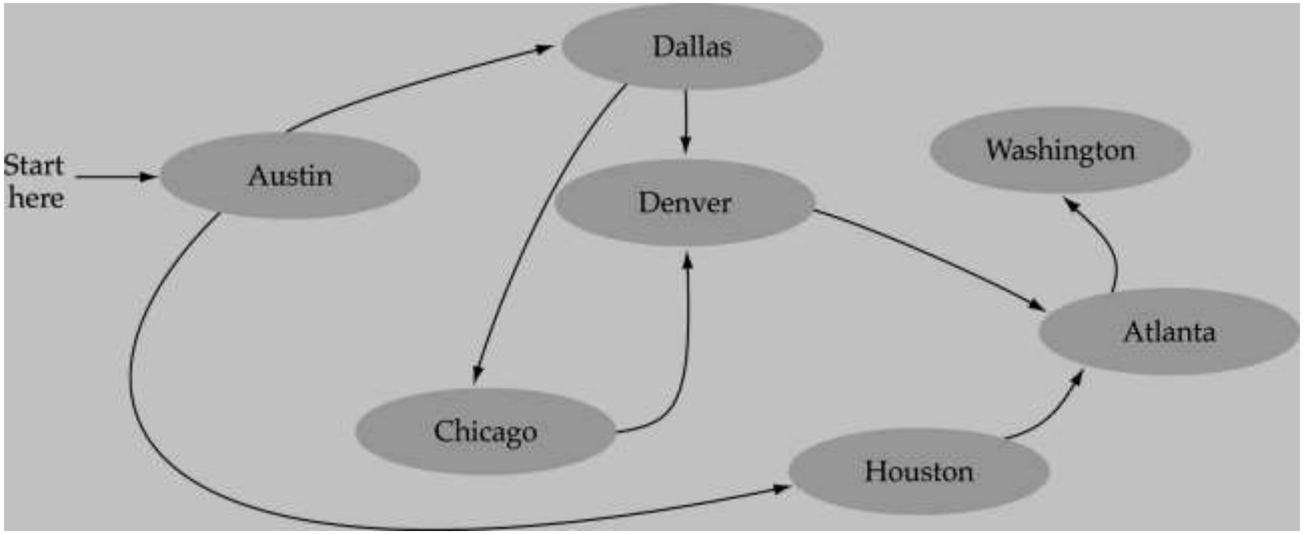
(b) Complete undirected graph.

Weighted graph

- <u>Weighted graph</u>: a graph in which each edge carries a value



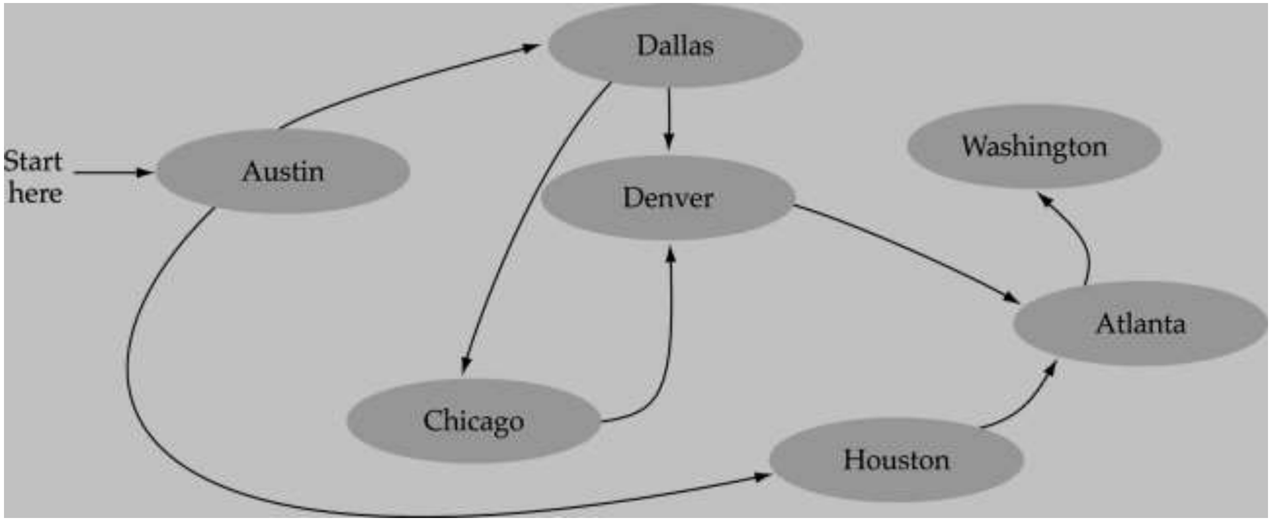
Application – Airport System



- Each airport is a vertex
- Two vertices are connected by an edge if there is a nonstop flight from the airports that are represented by the vertices.
- Each flight have different time, distance and cost.

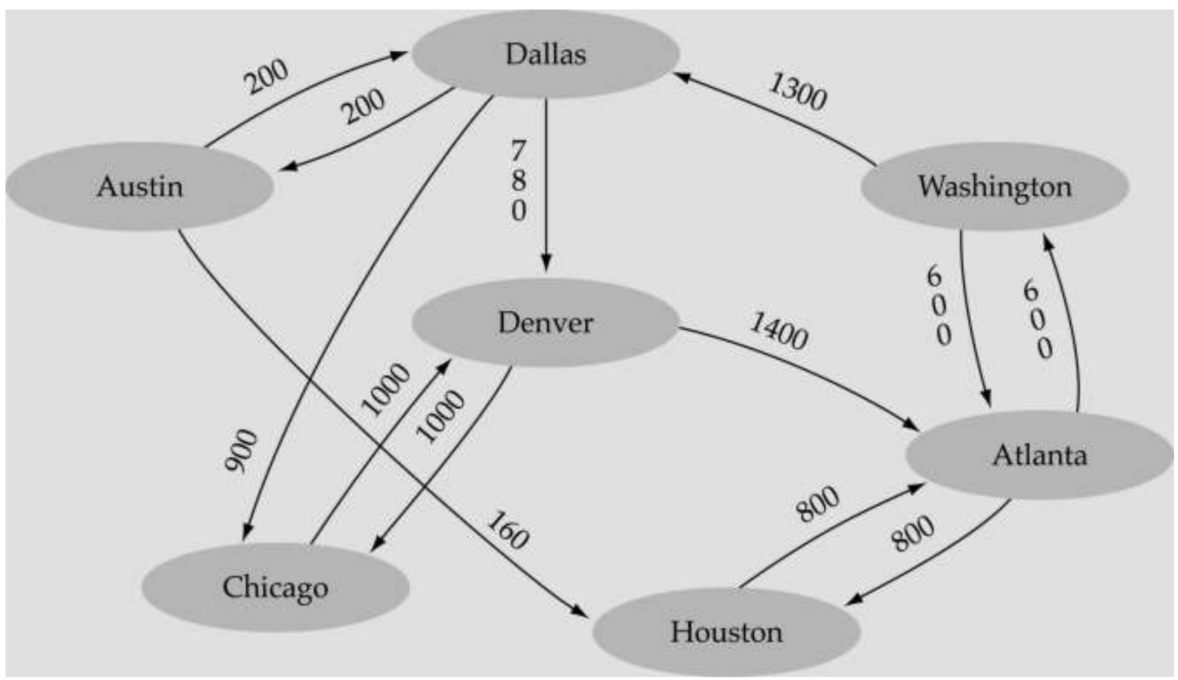
Application – Airport System

- Directed ?
  - It is reasonable to assume that such a graph is directed, since it might take longer or cost more (depending on local taxes, for example) to fly in different directions.
- Weighted ?
  - The edge could have a weight, representing the time, distance, or cost of the flight.

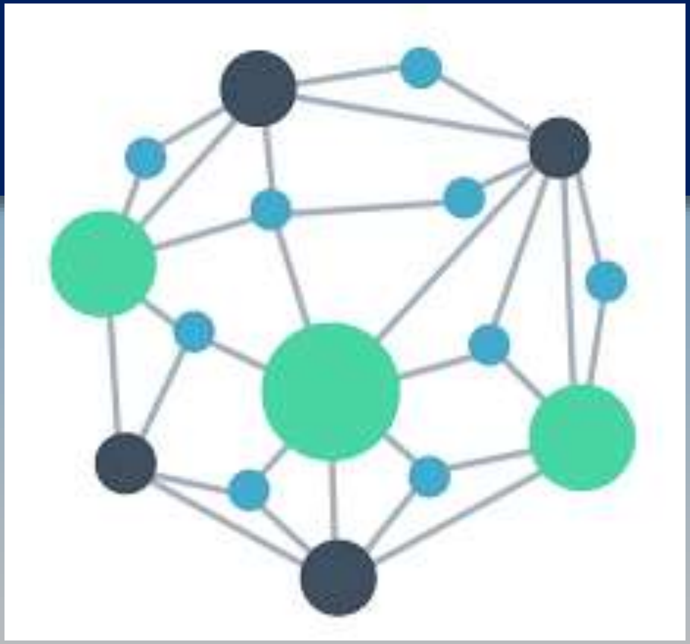


Application – Airport System

- Problems to solve?
  - Find Best flight from Dallas to all other locations
  - One want to find the best flight between any two airports.
  - "Best" could mean the path with the fewest number of edges or could be taken with respect to one, or all, of the weight measures



Lecture 2
Graphs



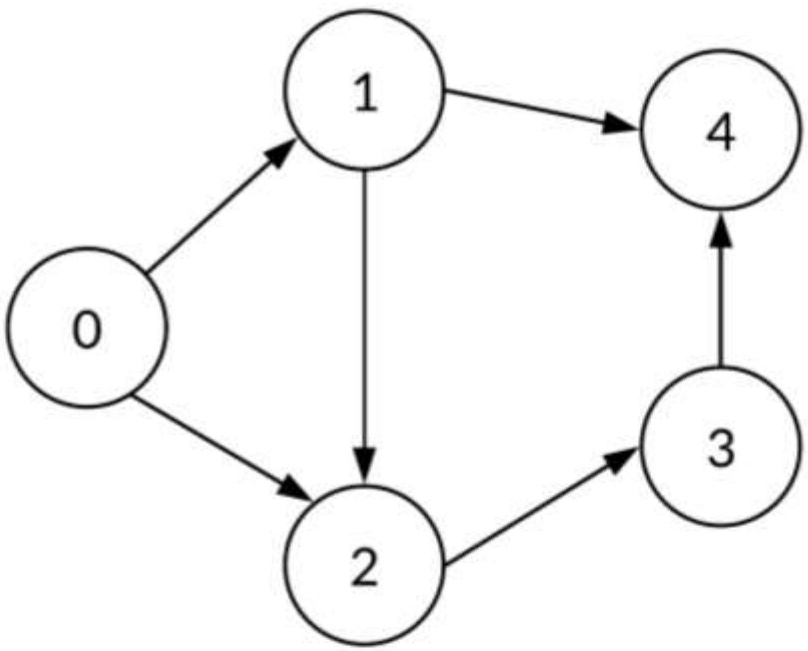
Graph Representation

Adjacency Matrix (Array-based)

Adjacency List (Linked-list-based)

Adjacency Matrix

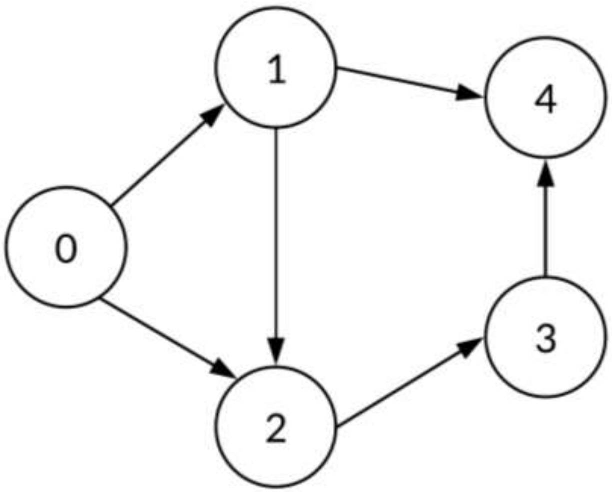
- Use a 2D array (i.e., adjacency matrix) to represent the edges



Adjacency Matrix

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

Adjacency Matrix

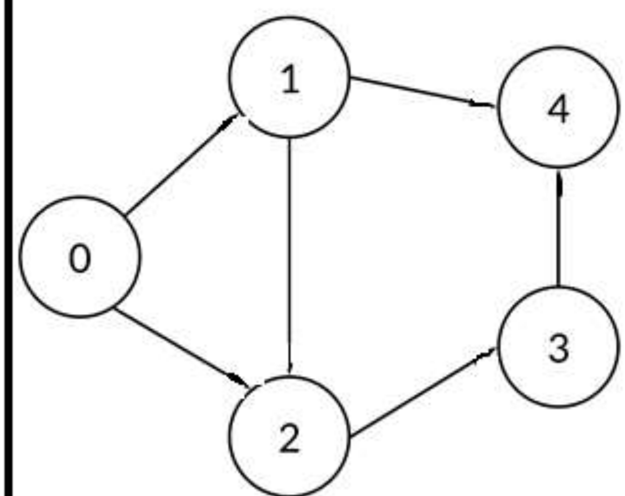


Adjacency Matrix

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

Directed Graph

Undirected Graph

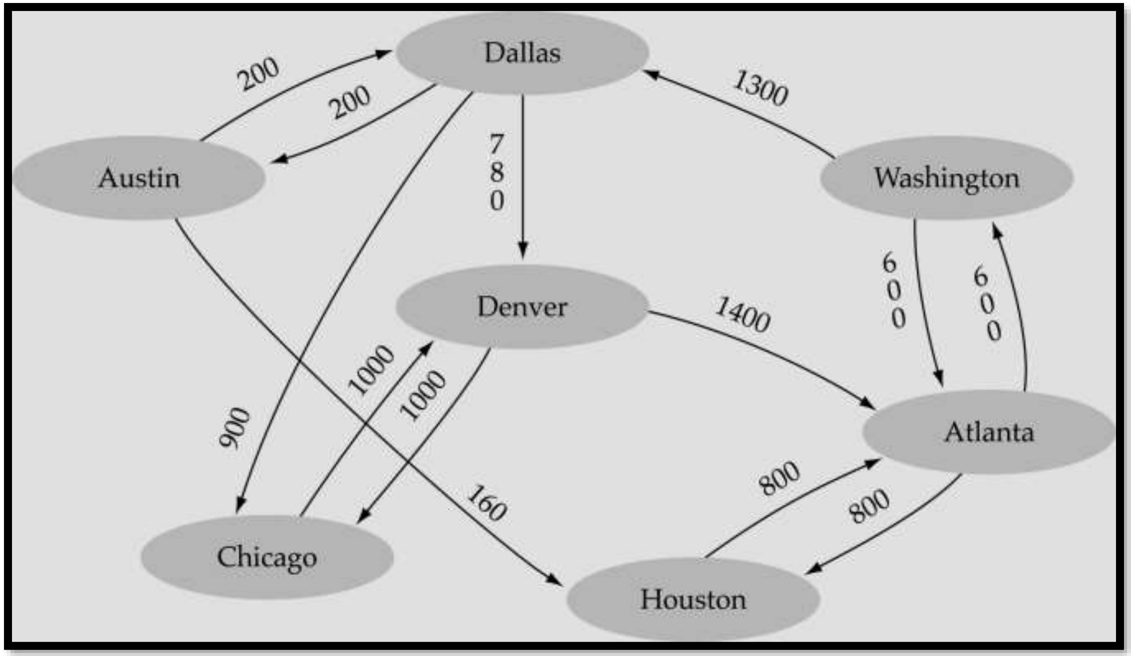


Adjacency Matrix

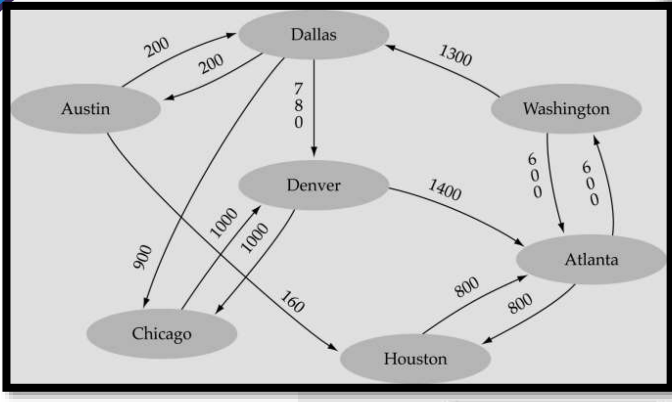
	0	1	2	3	4
0	0	1	1	0	0
1	$\boxed{1}$	0	1	0	1
2	$\boxed{1}$	$\boxed{1}$	0	1	0
3	0	0	$\boxed{1}$	0	1
4	0	$\boxed{1}$	0	$\boxed{1}$	0

Adjacency Matrix–Weighted Graph

- Use a **1D array** to represent the vertices
- Use a **2D array** (i.e., adjacency matrix) to represent the edges



Adjacency Matrix



2D array (adjacency matrix) to represent the edges

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

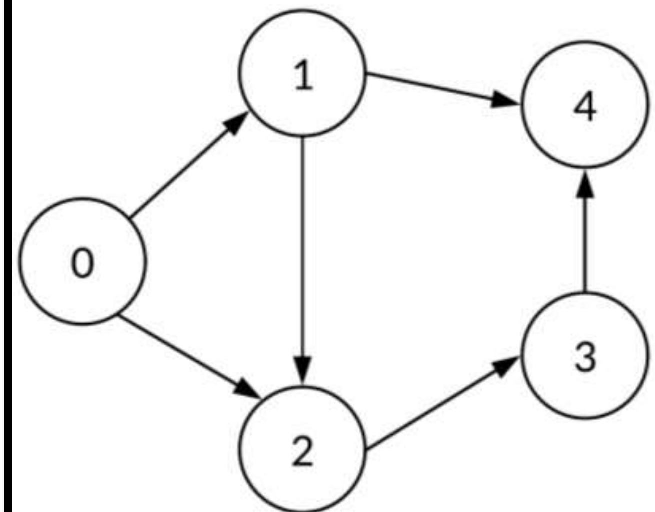
(Array positions marked '•' are undefined)

1D array to represent the vertices

Adjacency Matrix

- Memory required
  - $O(V+V^2)=O(V^2)$
- Preferred when
  - The graph is **dense:**

    $E = O(V^2)$



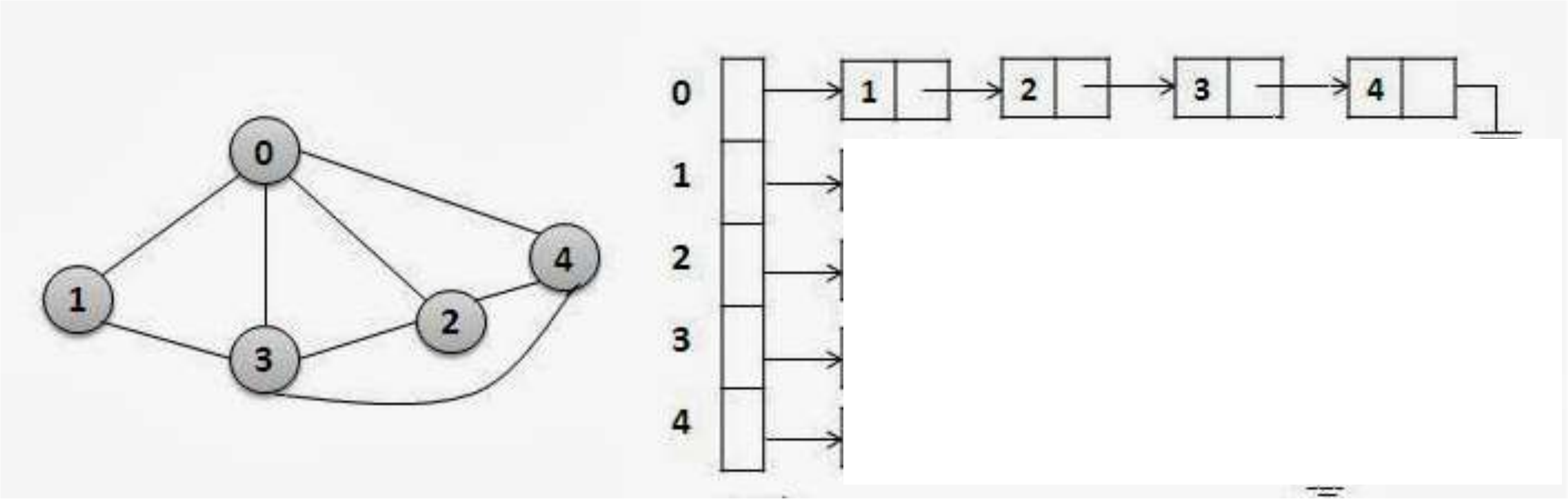
Adjacency Matrix

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

- Advantage
  - Can quickly determine if there is an edge between two vertices
- Disadvantage
  - No quick way to determine the vertices adjacent **from** a vertex

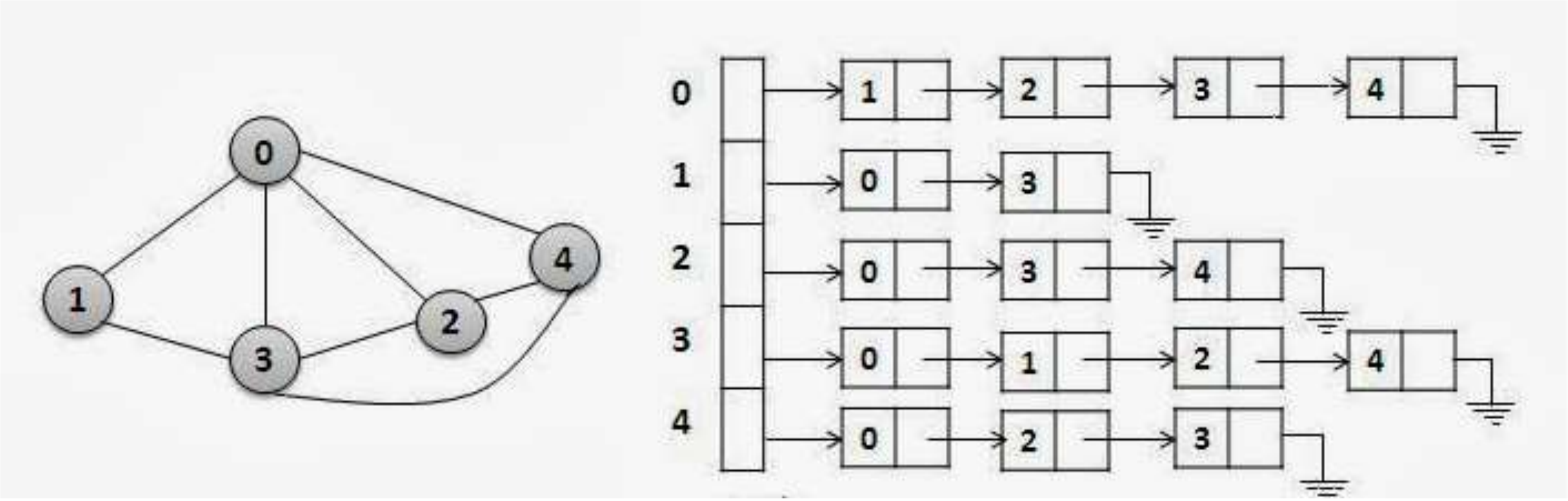
Adjacency List

- Use **a list for each vertex *v* to** contains the adjacent vertices **<u>from</u>** *v* (adjacency list)

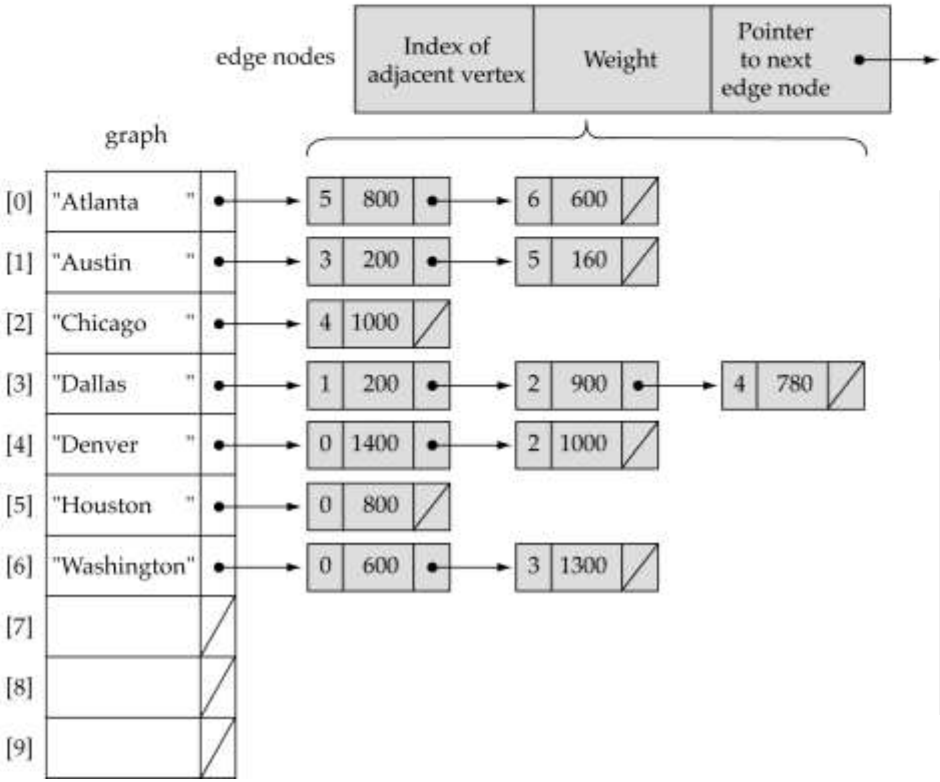


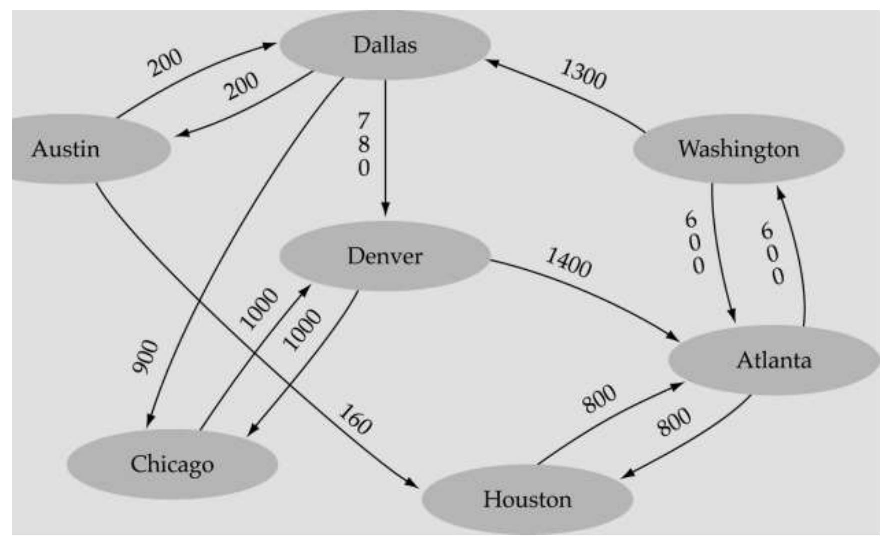
Adjacency List

- Use **a list for each vertex *v* to** contains the adjacent vertices **<u>from</u>** *v* (adjacency list)



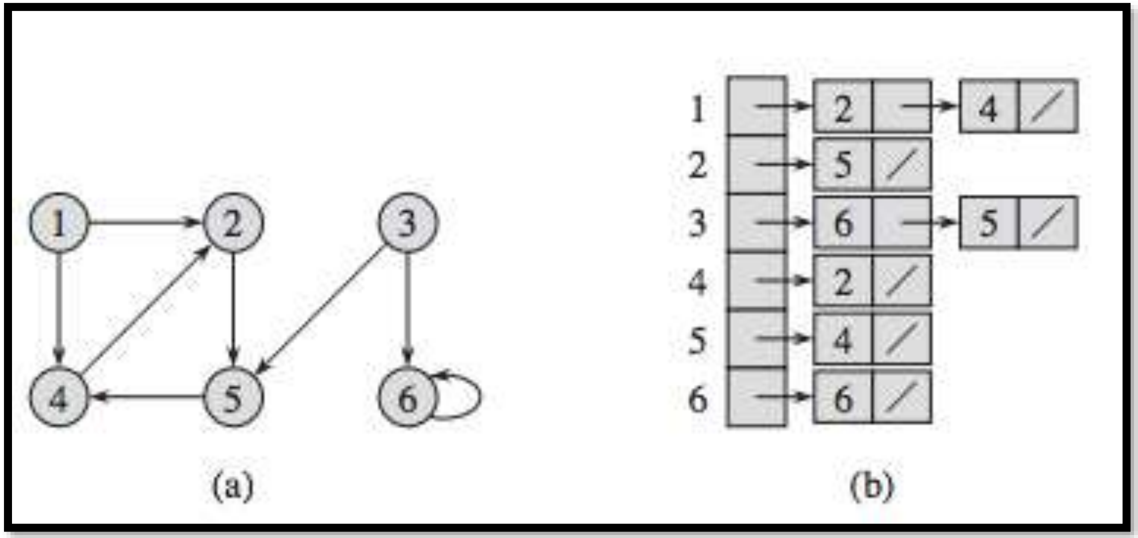
Adjacency List– Weighted DiGraph





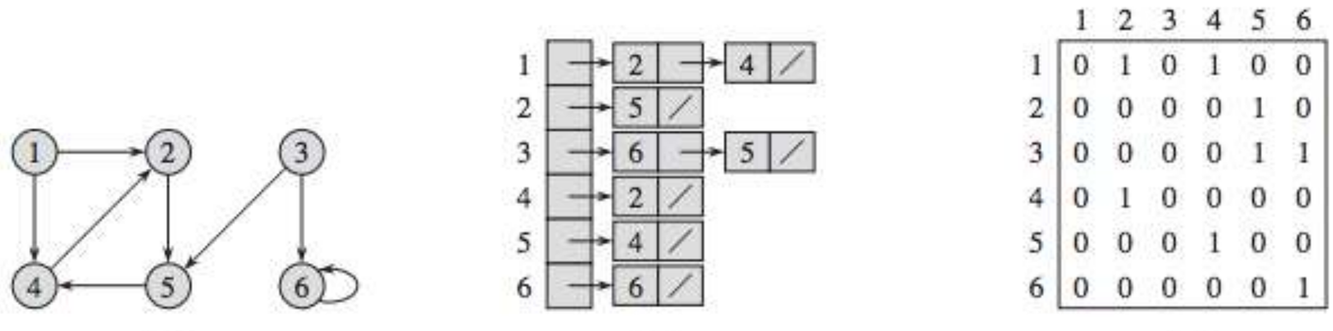
Adjacency List

- **Memory required**
  - O(V + E)
- **Preferred when**
  - graph is Sparse



- **Advantage**
  - Can quickly determine the vertices adjacent **from** a given vertex
- **Disadvantage**
  - No quick way to find if there is an edge between vertices u and v

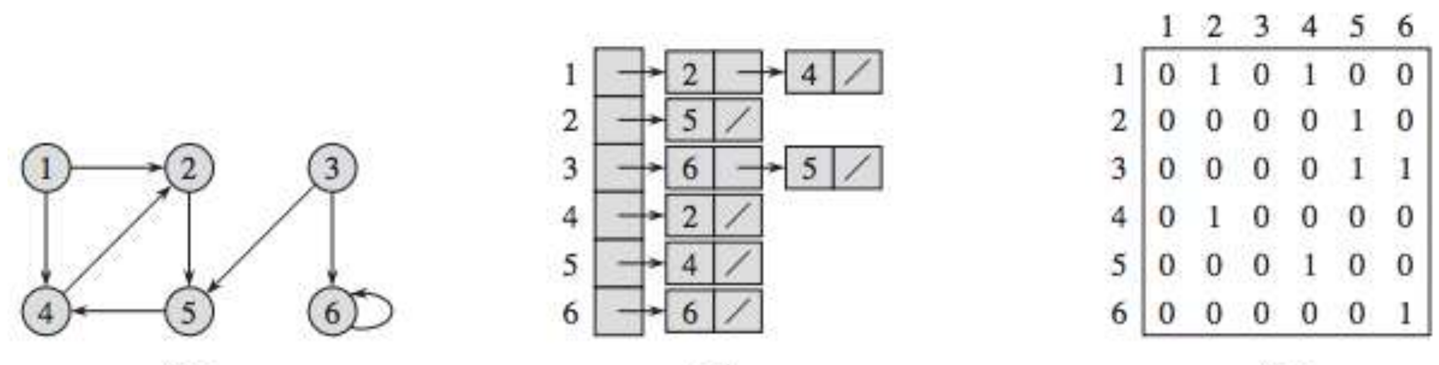
Runtime Analysis



(a) (b) (c)

Representation	Getting all adjacent edges for a vertex
Adjacency matrix	
[redacted]	
Adjacency List	

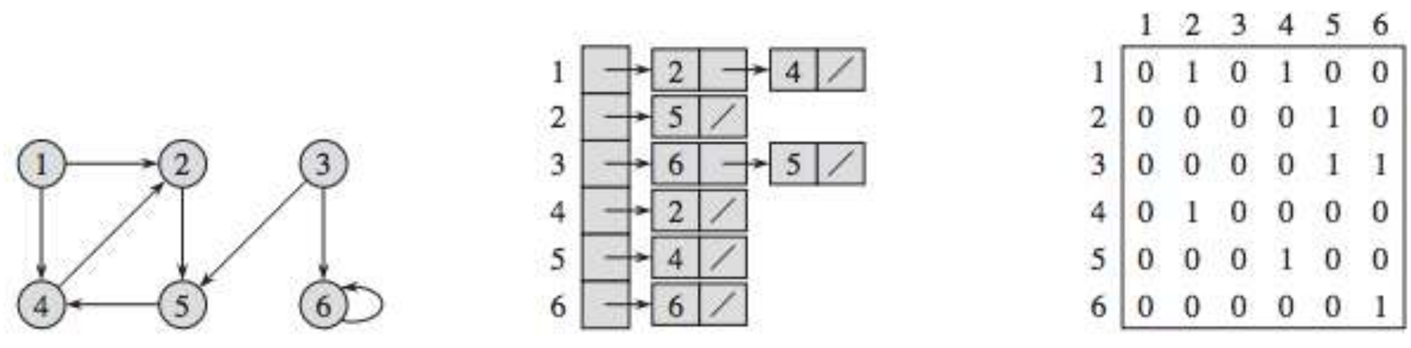
Runtime Analysis



(a) (b) (c)

Representation	Getting all adjacent edges for a vertex	Traversing entire graph
Adjacency matrix	O(V)	
Adjacency List	O(max deg of a vertex)	

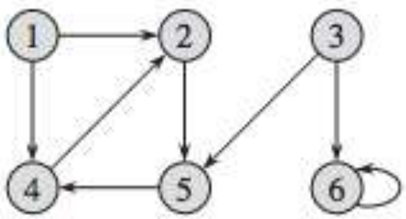
Runtime Analysis



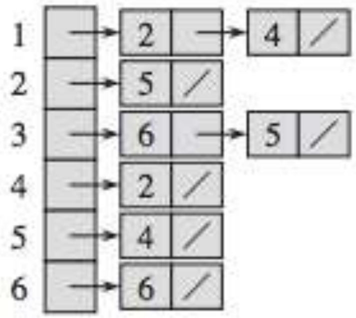
(a) (b) (c)

Representation	Getting all adjacent edges for a vertex	Traversing entire graph	hasEdge(u, v)
Adjacency matrix	O(V)	O(V^2)	
Adjacency List	O(max deg of a vertex)	O(V + E)	

Runtime Analysis



(a)



(b)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(c)

Representation	**Getting all adjacent edges for a vertex**	**Traversing entire graph**	**hasEdge(u, v)**	**Space**
Adjacency matrix	O(V)	$O(V^2)$	O(1)	$O(V^2)$
[redacted]				
Adjacency List	O(max deg of a vertex)	O(V + E)	O(max number of edges a vertex has)	O(E + V)

Which representation is best?

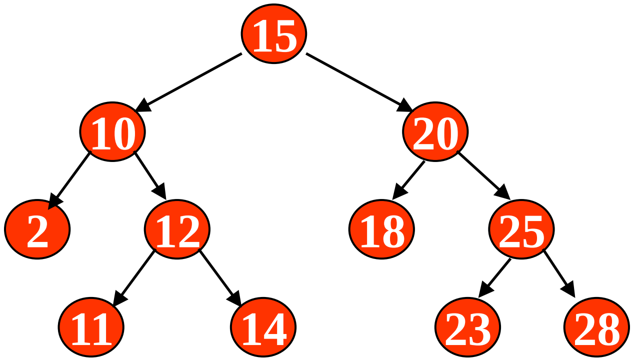
- It depends on the problem at hand.
- If our task is to process vertices adjacent to a vertex *v*,
  - the **adjacency list** requires only **deg(v)** steps,
  - whereas the **adjacency matrix** requires *|V|* steps.
- On the other hand, inserting or deleting a vertex adjacent to *v* requires
  - linked list maintenance for an adjacency list (if such an implementation is used)
  - for a matrix, it requires only changing 0 to 1 for insertion, or 1 to 0 for deletion, in one cell of the matrix.

Graph Traversals

Goal: Systematically explore every vertex and every edge

Iterating over a Graph

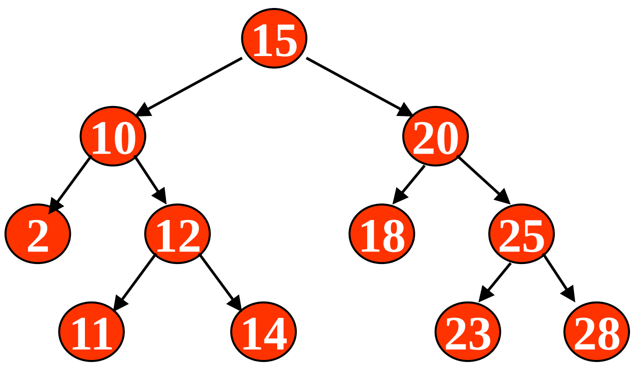
- In a linked list, there is just one way to traverse it.
  - Keep going forward.
- In BST, there are many traversal strategies:
  - An inorder traversal
  - A postorder traversal
  - A preorder traversal

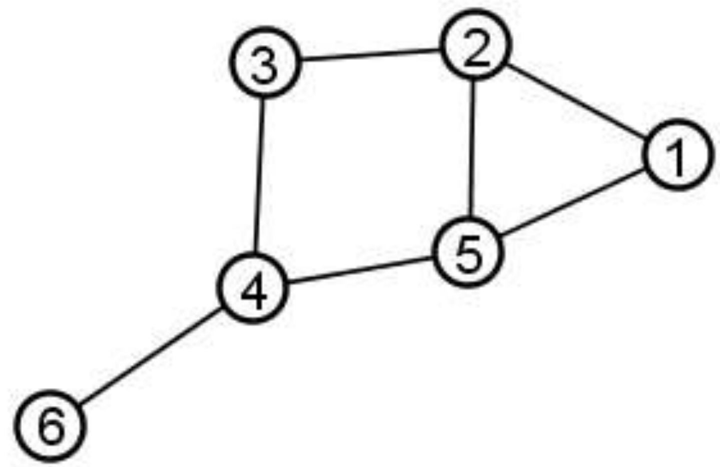


- There are ***many* ways** to iterate over a graph

Tree vs Graph Traversals

- As in trees, traversing a graph consists of visiting each vertex only one time.

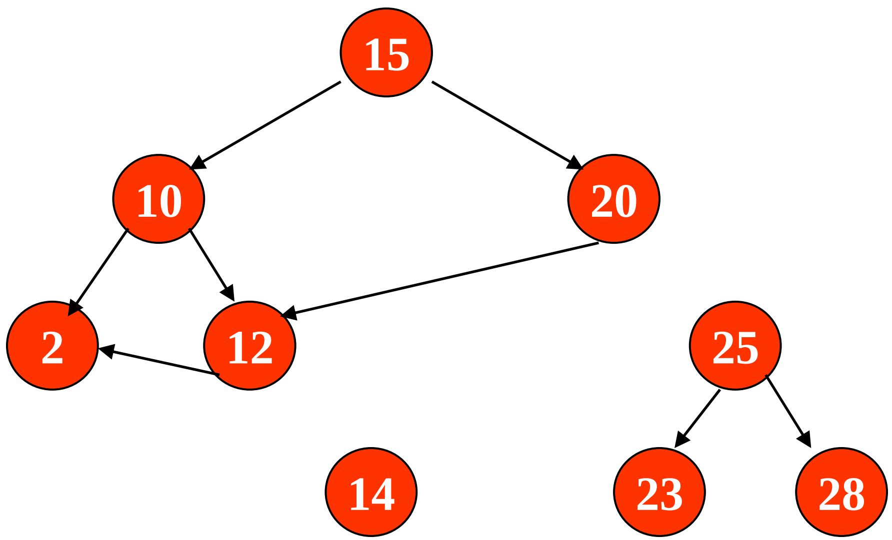




- Graph may have a cycle
  - So, tree traversal algorithms cannot be applied to graphs;
    - the tree traversal algorithms would result in infinite loops
    - To prevent that from happening, each visited vertex can be **marked** to avoid revisiting it.

Tree vs Graph Traversals

- Graphs can have isolated vertices
  - which means that some parts of the graph are left out if unmodified tree traversal methods are applied



Graph Traversals

- All methods of iterating over a graph involve keeping track of 3 sets of nodes:
  - Set of Nodes already visited
  - Set of Nodes to look at next
  - Everything else

Methods of iterating over nodes differ in how they choose which node to look at next

Depth-First-Search (DFS)

Graph Traversal

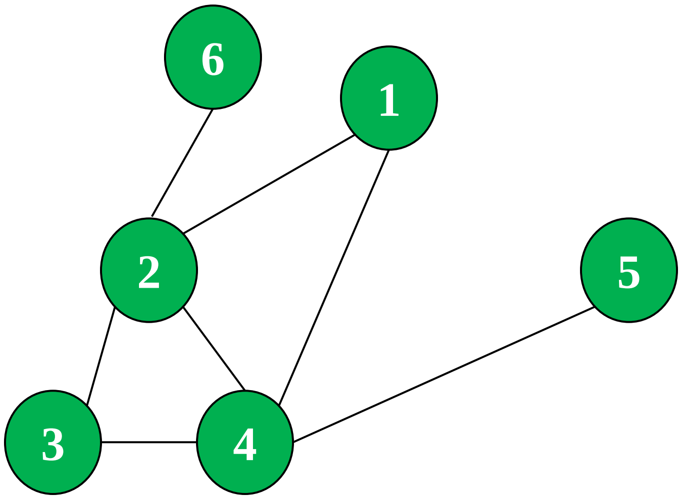
Depth-First-Search (DFS)

- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up ***as little as possible*** when you reach a **"dead end"** *(i.e., next vertex has been "marked" or there is no next vertex)*

- Goal: Systematically explore every vertex and every edge
- Idea: search deeper whenever possible
  - Using a Stack

- DFS was developed by John Hopcroft & Robert Tarjan

Example DFS

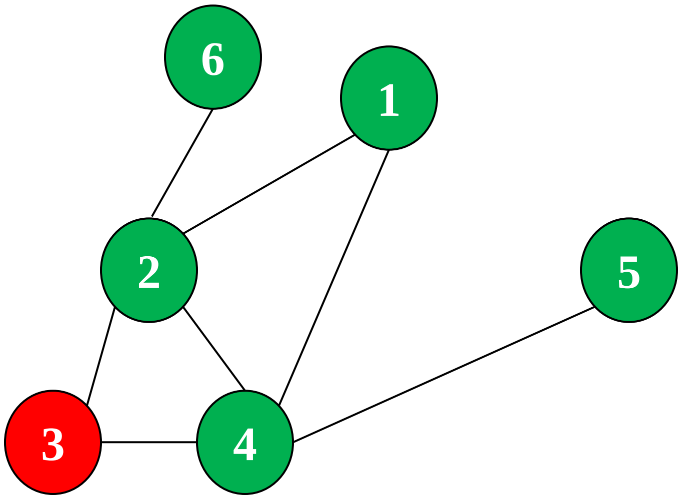


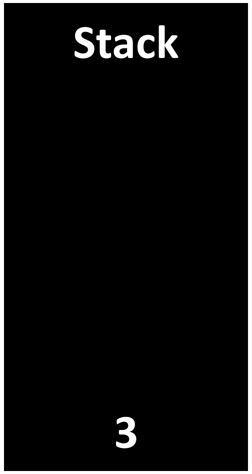
Stack

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

Output

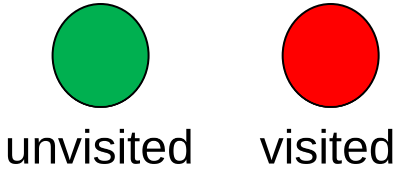
Example DFS



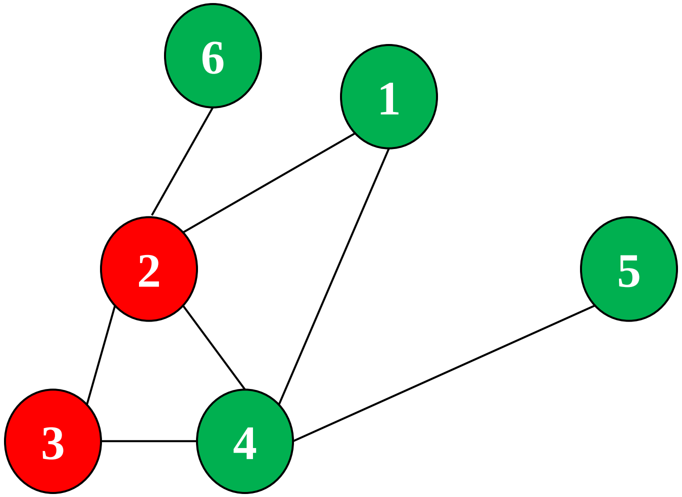


1	2	3	4	5	6
0	0	1	0	0	0

Output: 3



Example DFS



Stack

2

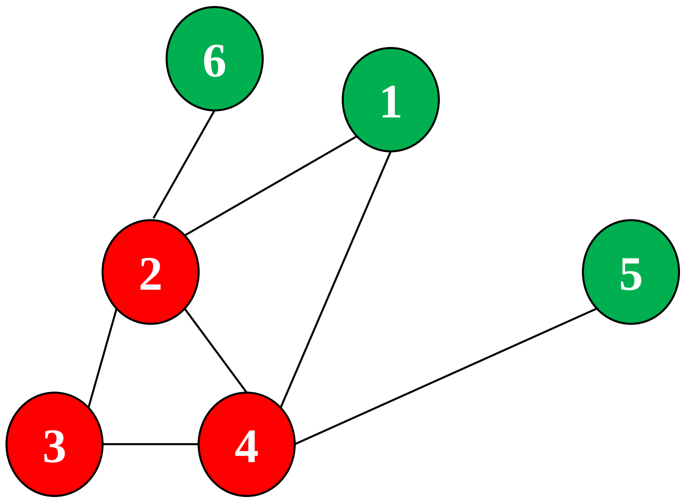
3

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

Output: 3 2

unvisited visited

Example DFS



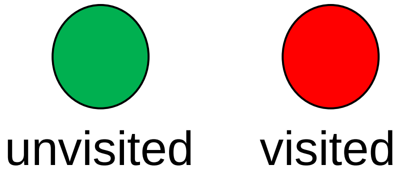
Stack

4

2

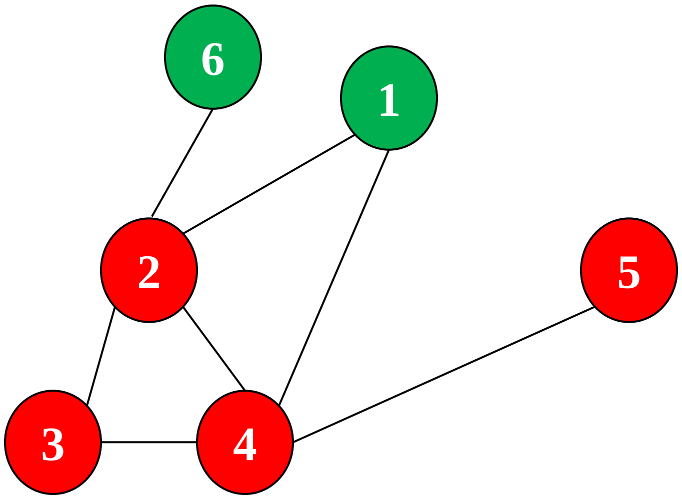
3

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



Output: 3 2 4

Example DFS



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

unvisited visited

Stack

5

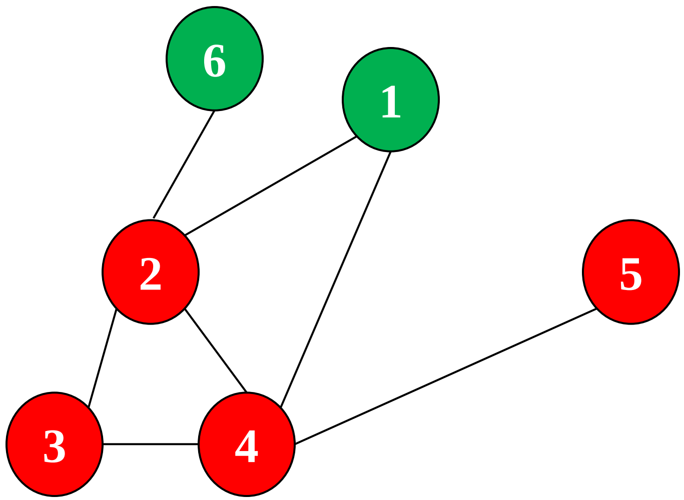
4

2

3

Output: 3 2 4 5

Example DFS



Stack

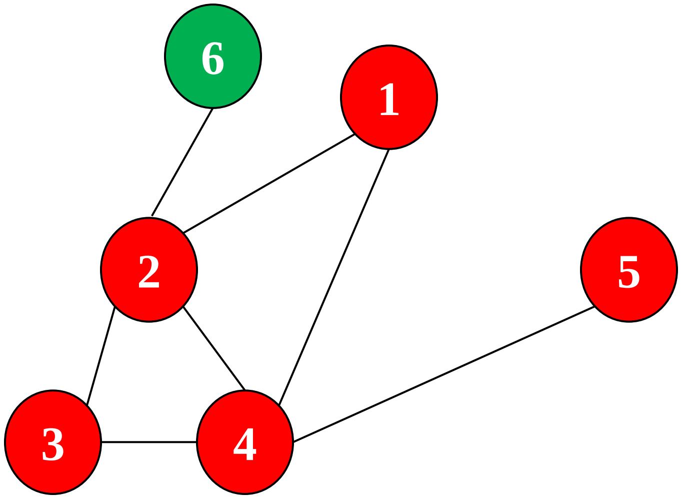
4
2
3

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

unvisited visited

Output: 3 2 4 5

Example DFS



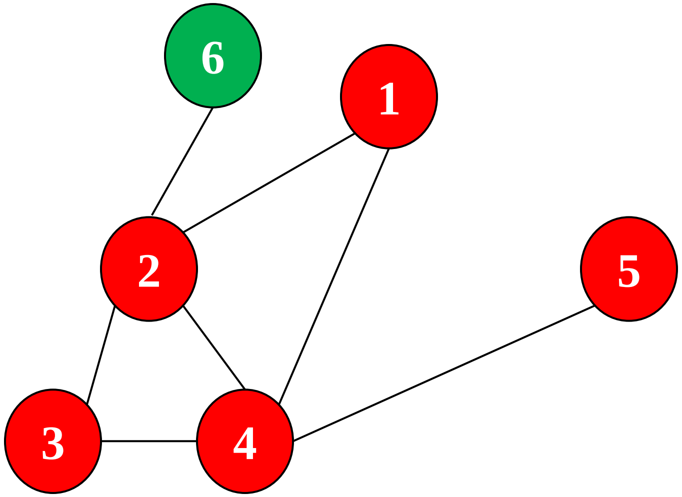
Stack

1
4
2
3

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

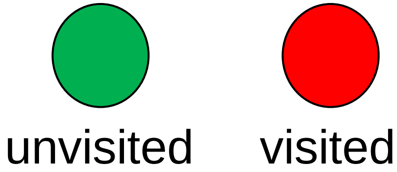
Output: 3 2 4 5 1

Example DFS



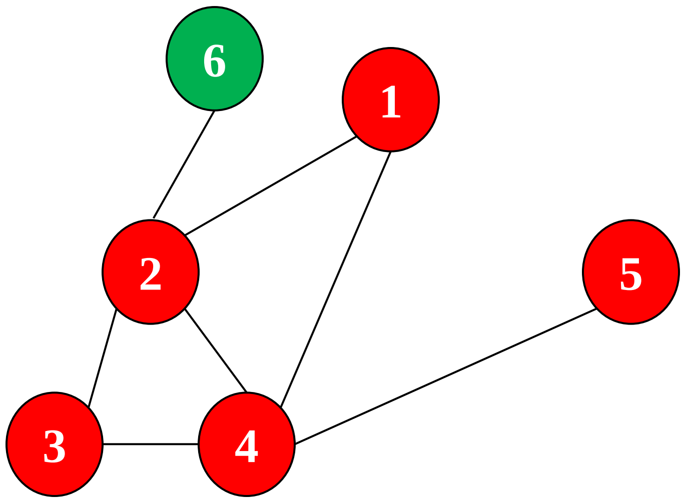
Stack
4
2
3

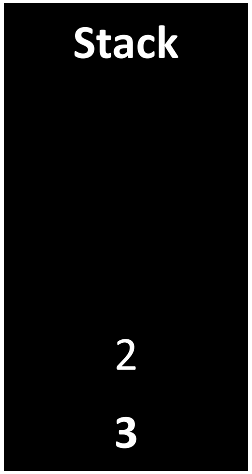
1	2	3	4	5	6
1	1	1	1	1	0



Output: 3 2 4 5 1

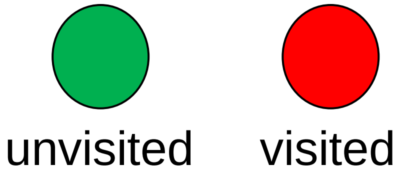
Example DFS



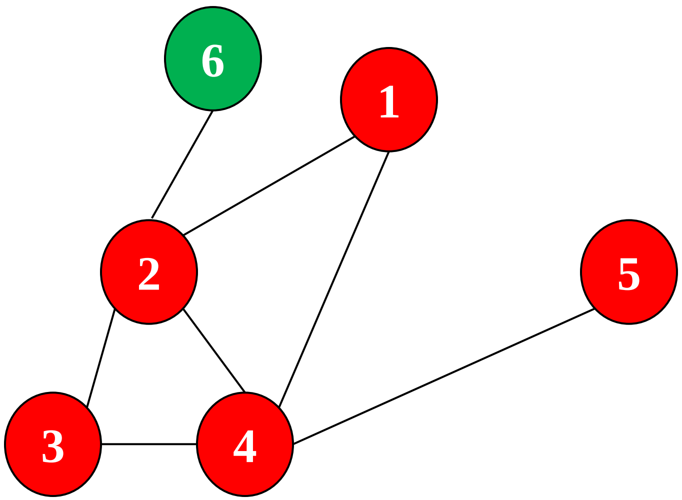


1	2	3	4	5	6
1	1	1	1	1	0

Output: 3 2 4 5 1



Example DFS



Stack

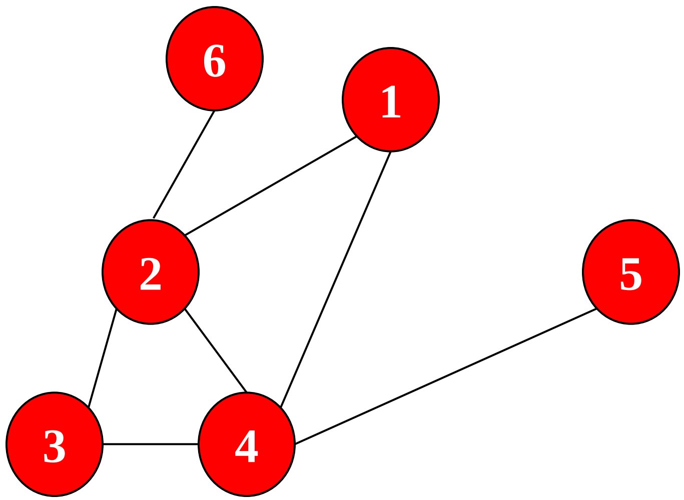
2

3

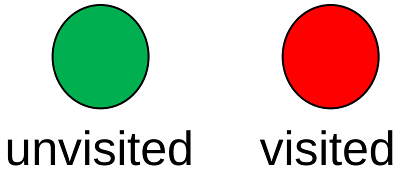
1	2	3	4	5	6
1	1	1	1	1	0

Output: 3 2 4 5 1

Example DFS



1	2	3	4	5	6
1	1	1	1	1	1



Stack

6

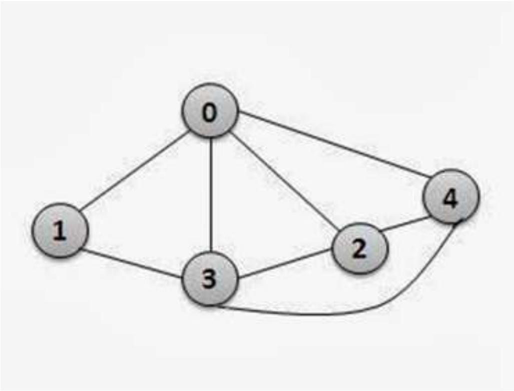
2

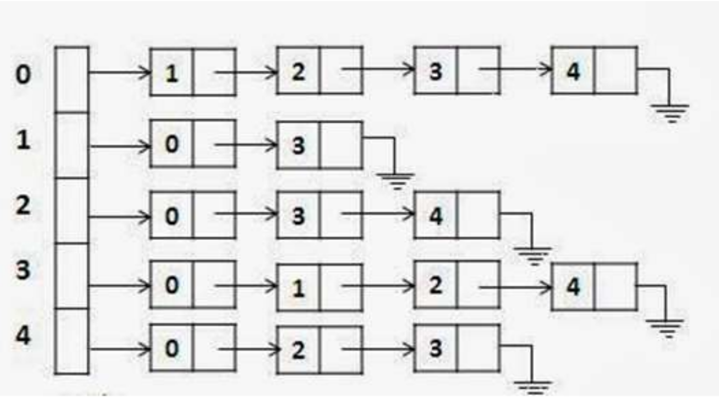
3

Output: 3 2 4 5 1 6

GRAPH – Adjacency list

```
#include<iostream>
#include <list>
class Graph{
    int V;    // No. of vertices

    list<int> *adj; //adjacency lists of
    neighbours (std list)

    // A recursive function used by DFS
    void DFSHelp(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    reachable from v
    void DFS(int v);
};
```





GRAPH – Adjacency list

```
#include<iostream>
#include <list>
class Graph{
    int V;    // No. of vertices

    list<int> *adj; //adjacency lists of
    neighbours (std list)

    // A recursive function used by DFS
    void DFSHelp(int v, bool visited[]);
public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    reachable from v
    void DFS(int v);
};
```

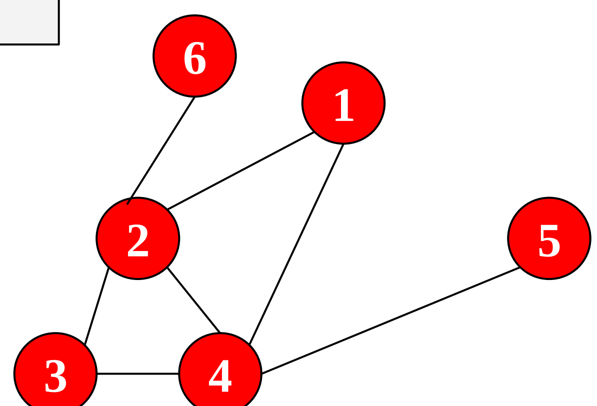
```
Graph::Graph(int V){
    this->V = V;
    adj = new list<int>[V];
}
void Graph::addEdge(int v,int w){
        adj[v].push_back(w);
        adj[w].push_back(v);
// undirected.
}
```

```
void main(){
// Create a graph
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
}
```

Recursive DFS

```
// DFS traversal of the vertices
reachable from v.
// It uses recursive DFSHelper()

void Graph::DFS(int v)
{
// Mark all vertices as not
visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

// Call the recursive helper
function to print DFS traversal
    DFSHelp(v, visited);
}
```

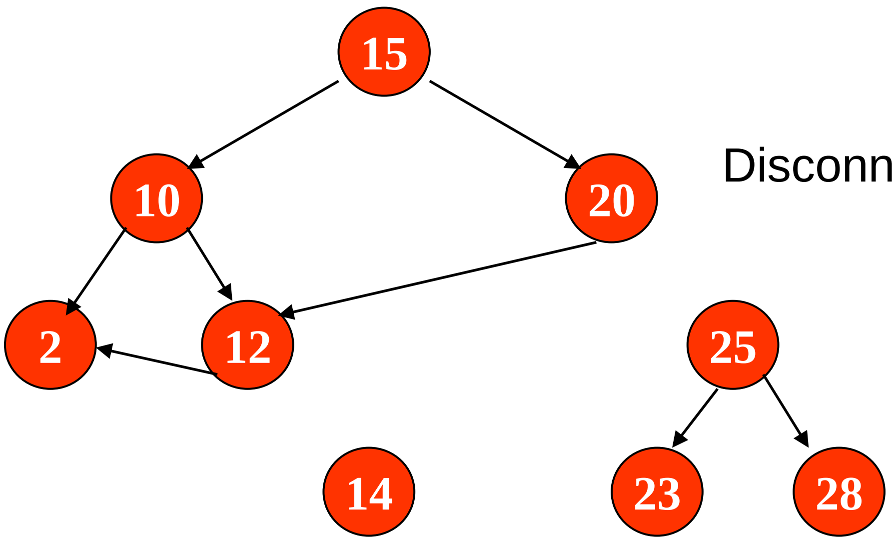
```
void Graph::DFSHelp(int v, bool visited[])
{
// Mark the current node as visited and
// print it
    visited[v] = true;
    cout << v << " ";

// Recur for all adjacent vertices
list<int>::iterator i;
  for(i=adj[v].begin(); i!=adj[v].end();++i){
        if (!visited[*i])
            DFSHelp(*i, visited);
  }
}
```



https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

How to handle disconnected graph?

- The code traverses only the vertices reachable from a given source vertex.
- All the vertices may not be reachable from a given vertex
- To do complete DFS traversal of such graphs, we must call DFSHelper() for every vertex.



Disconnected graph

Recursive DFS For Disconnected graph

```
// DFS traversal of the vertices
reachable from v.

void Graph::DFS()
{
// Mark all vertices as not
visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
    visited[i] = false;
// Call the recursive helper
function to print DFS traversal
starting from all vertices one
by one

for (int i = 0; i < V; i++)
     if (visited[i] == false)
         DFSHelp(i, visited);

 delete []visited;
}
```

```
void Graph::DFSHelp(int v,bool visited[])
{
// Mark the current node as visited and
// print it
    visited[v] = true;
    cout << v << " ";

// Recur for all the vertices adjacent
// to this vertex
    list<int>::iterator i;
  for(i=adj[v].begin(); i!=adj[v].end();++i){
        if (!visited[*i])
            DFSHelp(*i, visited);
  }
}
```

TIME COMPLEXITY

- The complexity of DFS() is $O(|V| + |E|)$ because
  - initializing *visited array* for each vertex *v* requires $|V|$ steps;
  - DFSHelp is called *deg*(*v*) times for each *v*—that is, once for each edge of *v*
  - For a graph with no isolated parts, the loop makes only one iteration, and an initial vertex can be found in one step,
  - Although it may take $|V|$ steps. For a graph with all isolated vertices, the loop iterates $|V|$ times,
    and each time a vertex can also be chosen in one step

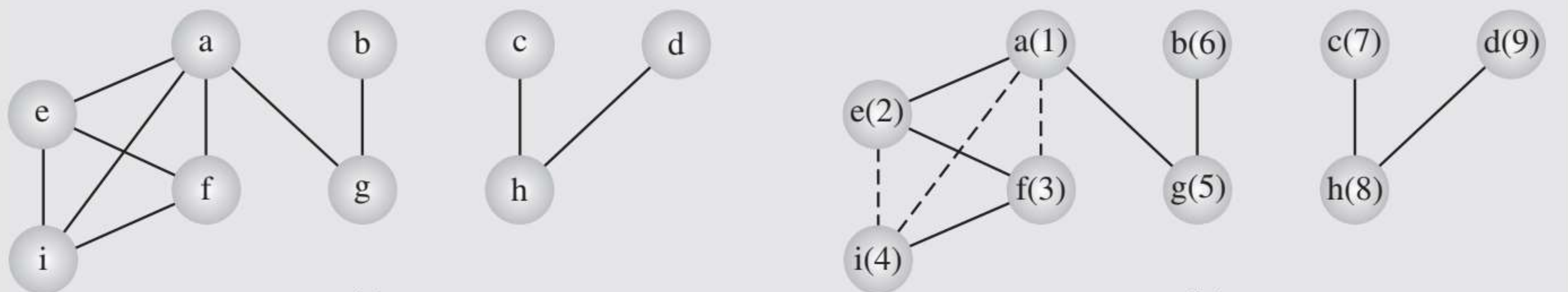
TIME COMPLEXITY

- **The complexity of DFS() when adjacency Matrix is used.**

- For example, if an adjacency list is used, then for each *v*, the condition in the loop, for *all vertices* u *adjacent to* v is checked *deg*(v) times.
- However, if an adjacency matrix is used, then the same condition is used $|V|$ times, whereby the algorithm's complexity becomes $O(|V|^2)$

DFS - can save the order of nodes visited

- Instead of printing we can safe the order in which vertices are visited …
  - in **visited** array by making it of type int



(a) (b)

PseudoCode for order of nodes and edge list

- `DFSHelp(v, visited[], edges[]){`
  - `visited[v] = i++;`
  - `for all vertices u adjacent to v`
    - `if (visited[u] = 0 ){`
      - `Attach edge(uv) to edges;`
      - `DFSHelper(u);`
    - `}`
- `}`

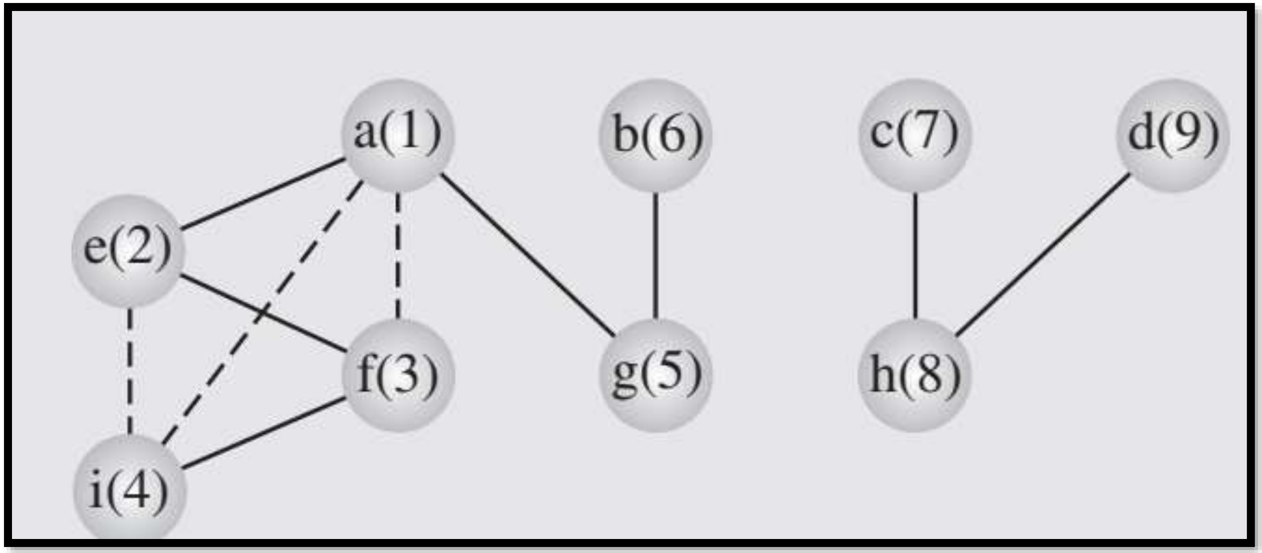
This algo (Adam Drozdek)

- Save the order of vertices
- Save the edges visited (forward edges)

Spanning tree from DFS

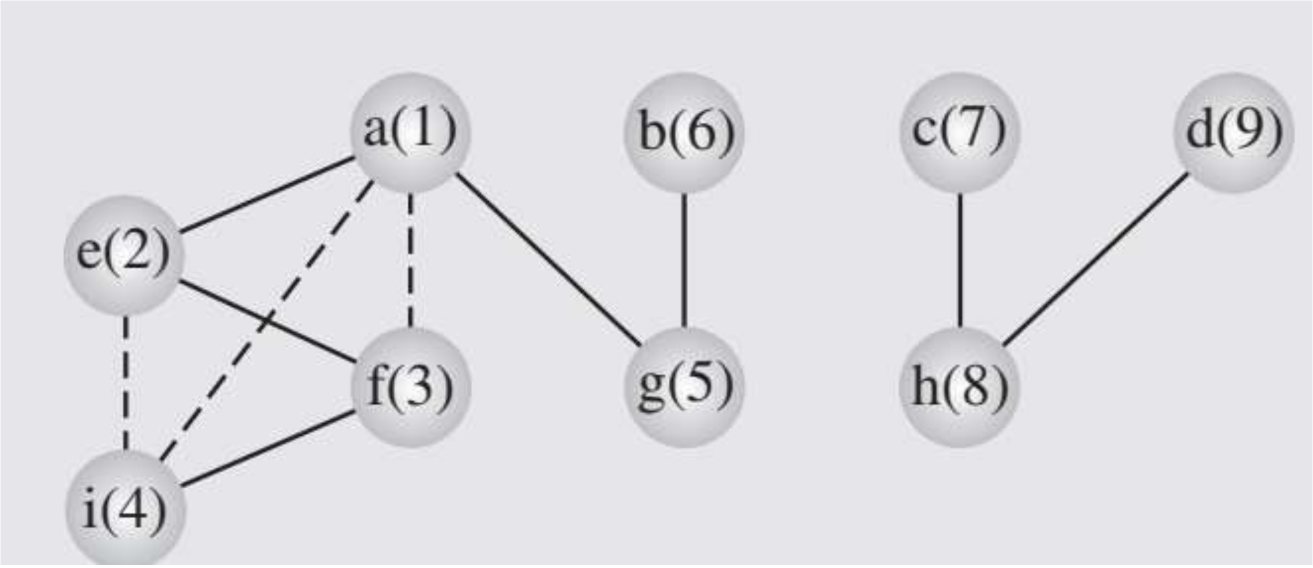
Note that this algorithm guarantees generating a tree (or a forest, a set of trees) that includes or spans over all vertices of the original graph.

A tree that meets this condition is called a *spanning tree.*



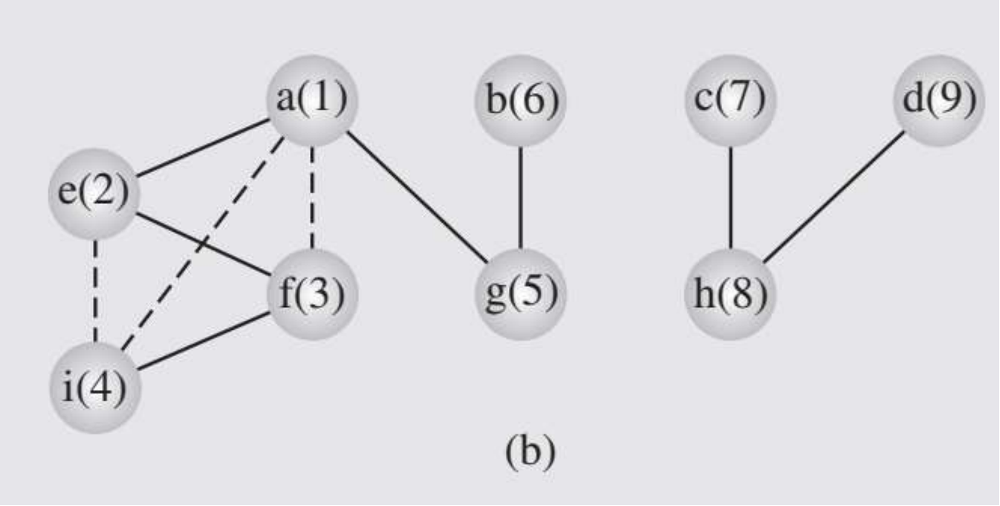
Spanning tree from DFS

- The fact that a tree is generated is ascertained by the fact
  - that the algorithm does not include in the resulting tree any edge that leads from the currently analyzed vertex to a vertex already analyzed.
- An edge is added to edges only if the condition in "if visited[u]=0" is true;
  - that is, if vertex *u* reachable from vertex *v* has not been processed.

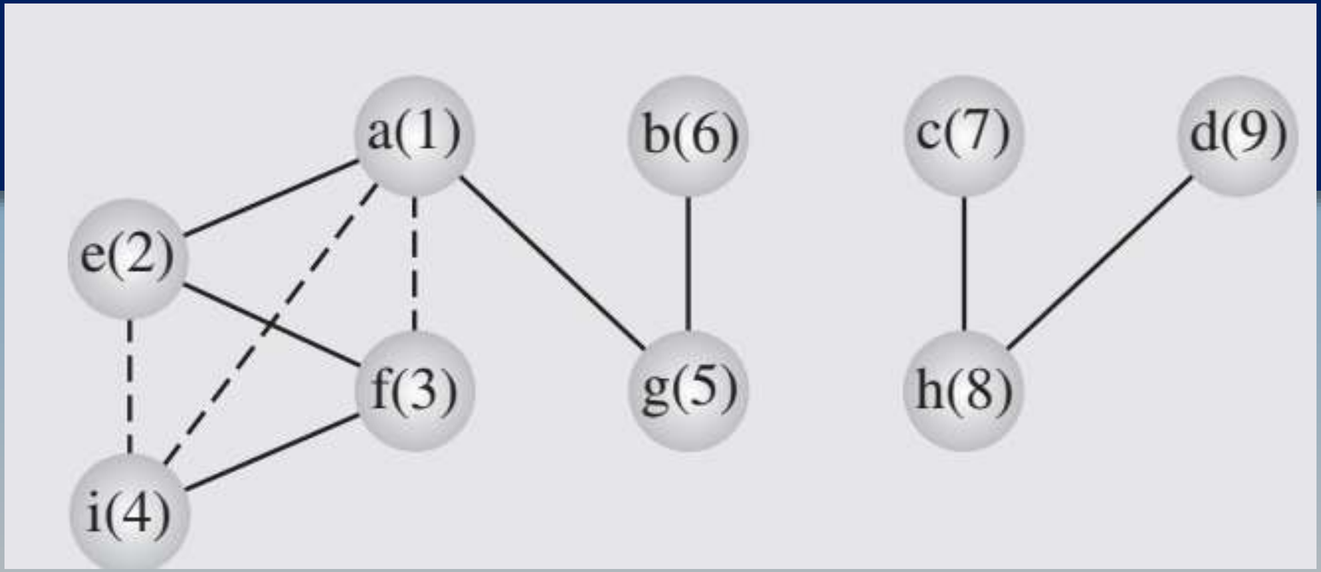


Spanning tree from DFS

- As a result, certain edges in the original graph do not appear in the resulting tree.
- The edges included in this tree are called **forward edges (or tree edges), and**
- the edges not included in this tree are called ***back edges*** **and are shown as dashed lines**



Cycle Detection



Detect cycle code

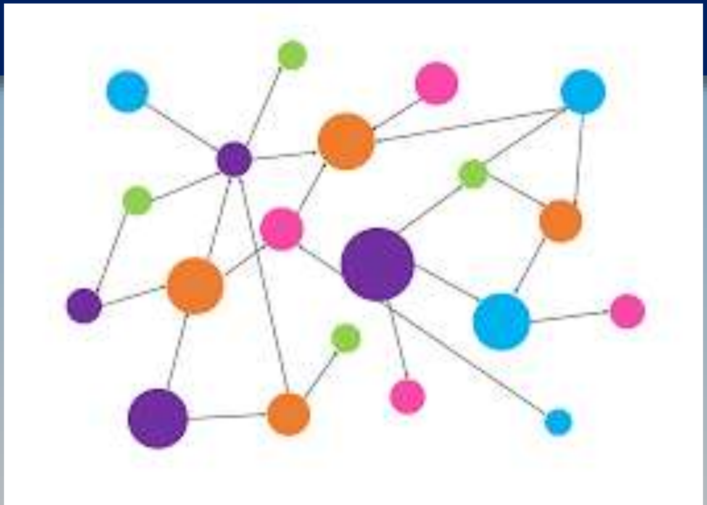
```
bool Graph::isCyclic()
{
    // initialize visited array
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the helper func to detect cycle in different DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if it is already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

     return false;
}
```

Detect cycle code

```
// A recursive funct that uses visited[] and parent to detect cycle in subgraph
reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent){
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i){
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i]){
            if (isCyclicUtil(*i, visited, v))
            return true;
        }

        // If an adjacent vertex is visited and not parent of current
        vertex, then there is a cycle.
        else if (*i != parent)
                return true;
    }
    return false;
}
```

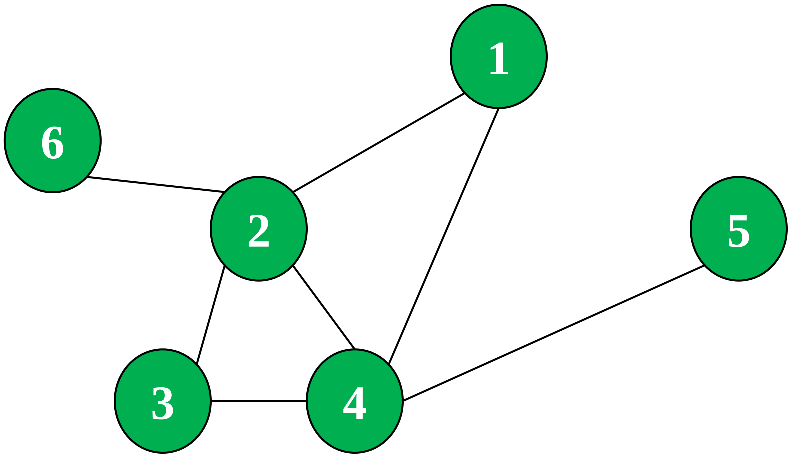
Time Complexity O(V+E)

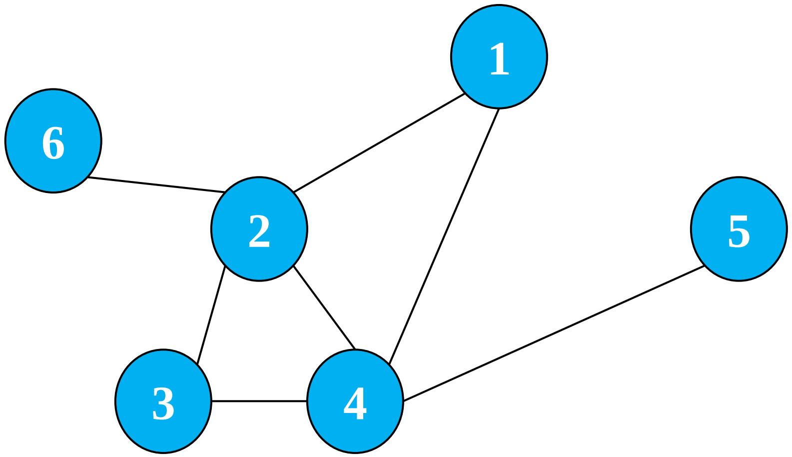
Breadth First Search
BFS



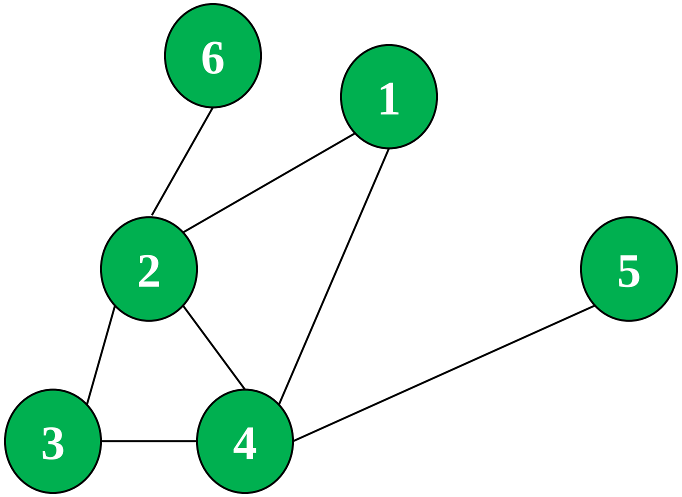
BFS

- BreadthFirstSearch() first tries to mark all neighbors of a vertex *v* before proceeding to other vertices,
- DFS() picks one neighbor of a *v* and then proceeds to a neighbor of this neighbor before processing any other neighbors of *v*.





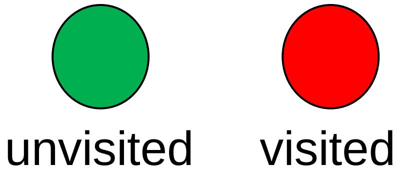
Example BFS



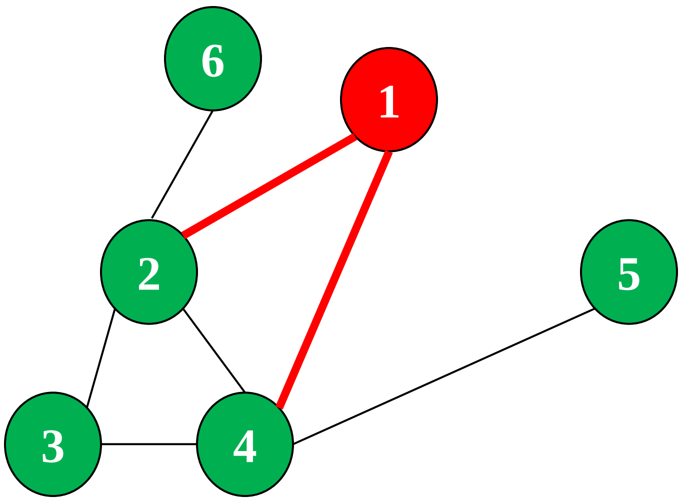
queue

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

Output



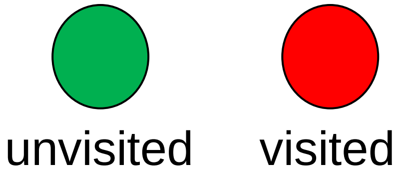
Example BFS



queue

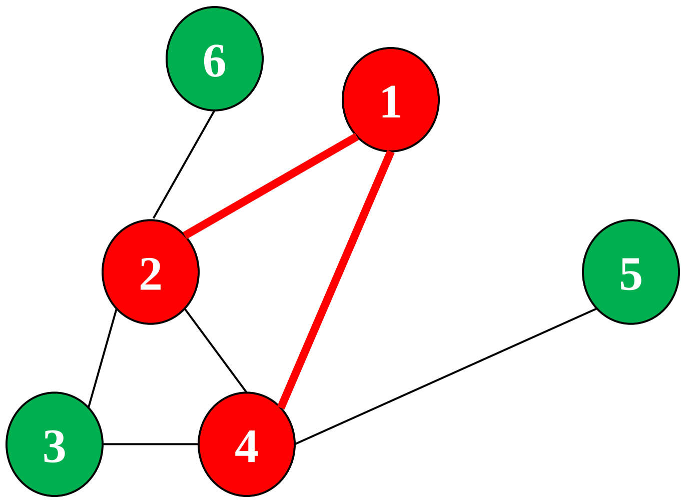
1					

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

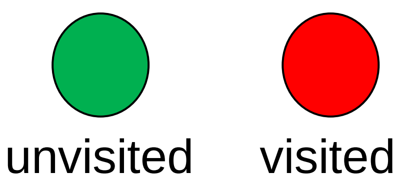


Output

Example BFS



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



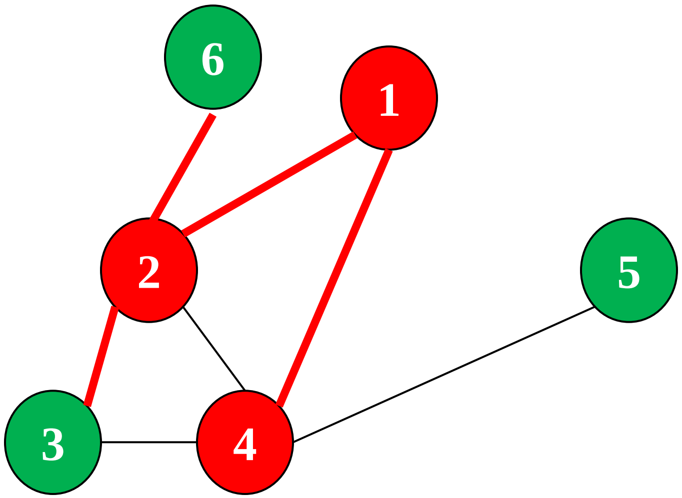
queue

2	**4**				

Output

1

Example BFS



queue

4					

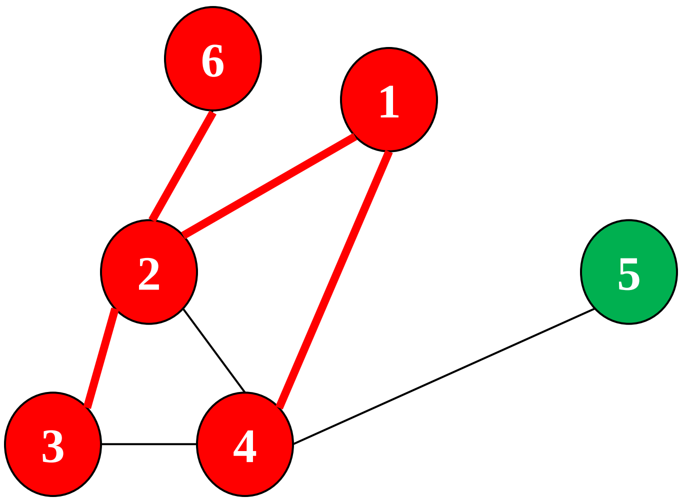
1	2	3	4	5	6
1	1	0	1	0	0

unvisited visited

Output

1 2

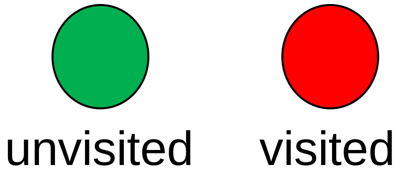
Example BFS



queue

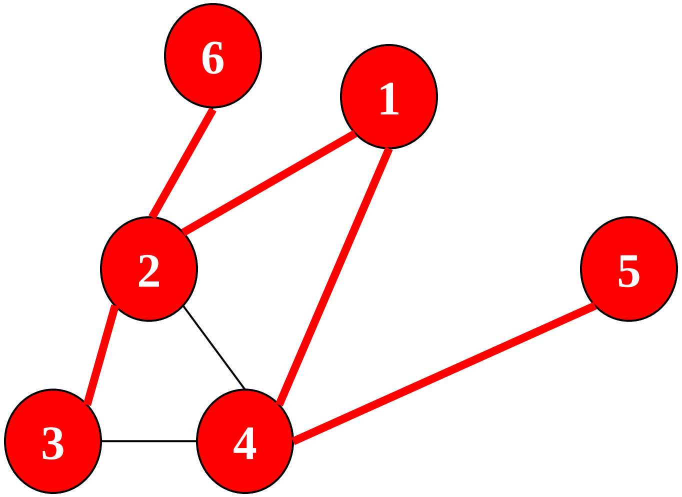
4	3	6			

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



Output
1 2

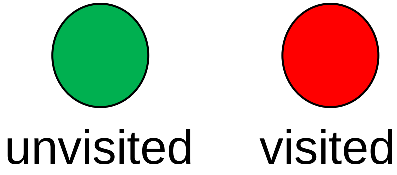
Example BFS



queue

3	6	5			

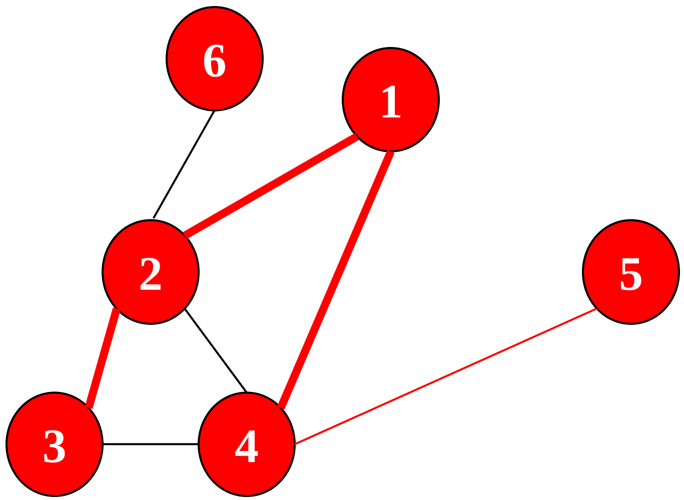
1	2	3	4	5	6
1	1	1	1	1	1



Output

1 2 4

Example BFS



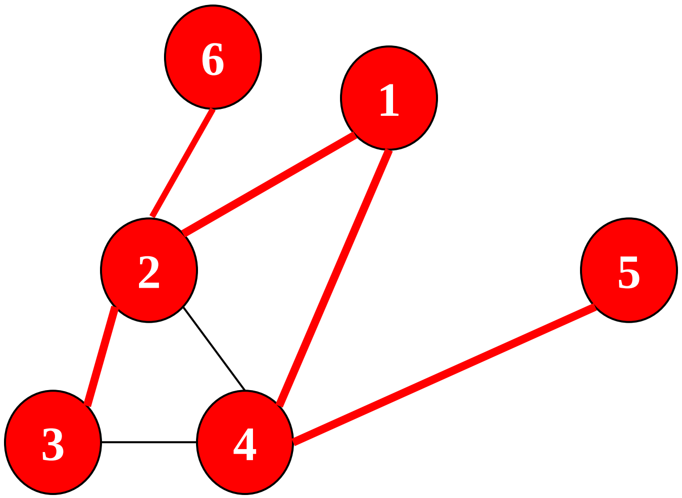
queue

6	5				

1	2	3	4	5	6
1	1	1	1	1	1

Output
1 2 4 3

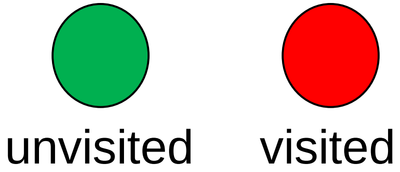
Example BFS



queue

5					

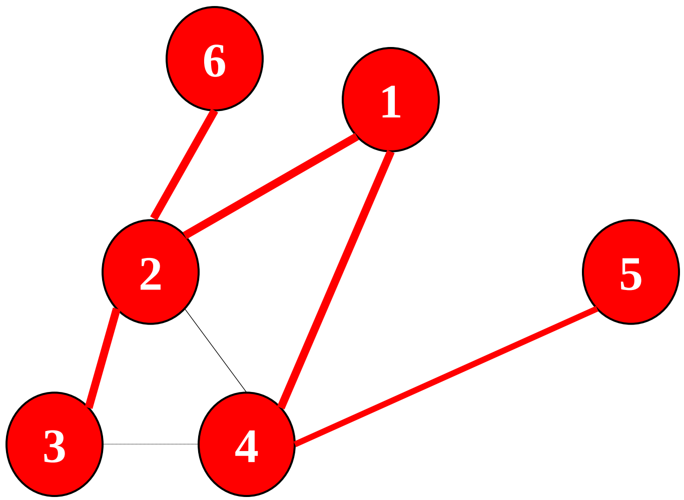
1	2	3	4	5	6
1	1	1	1	1	1



Output

1 2 4 3 6

Example BFS



1	2	3	4	5	6
1	1	1	1	1	1

queue

Output

1 2 4 3 6 5

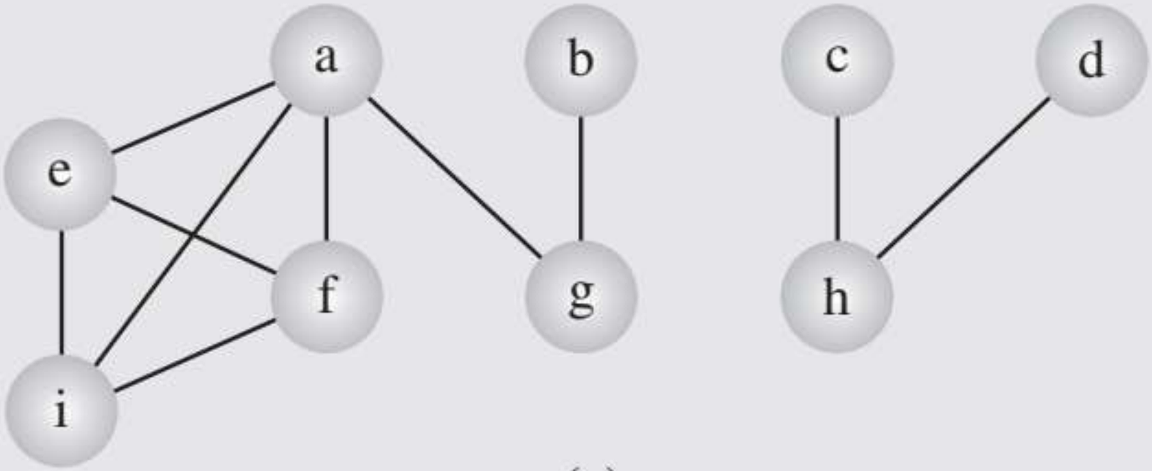
GRAPH – Adjacency list

```
#include<iostream>
#include <list>
class Graph{
    int V;    // No. of vertices

    list<int> *adj; //adjacency lists of
    neighbours (std list)

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // BFS traversal of the vertices
    reachable from v
    void BFS(int v);
};
```

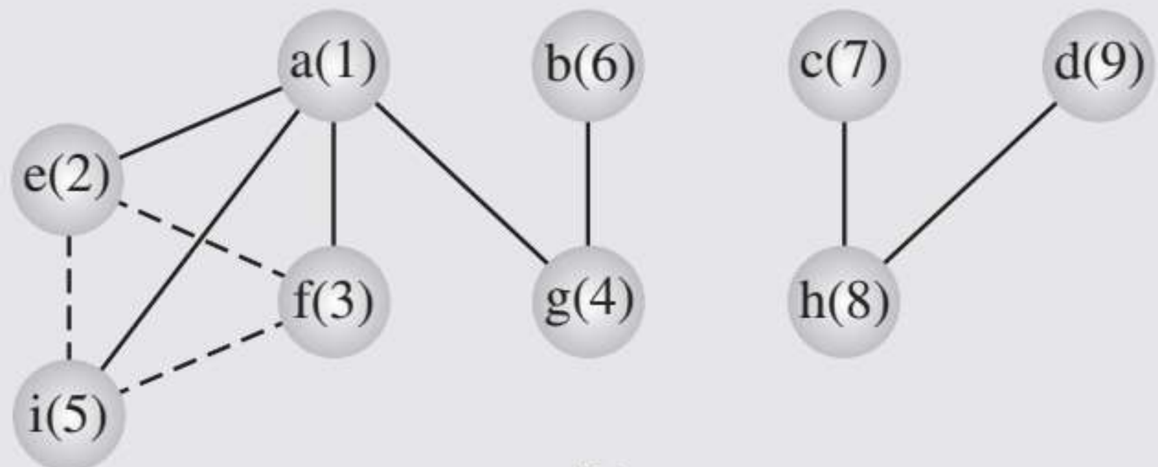
```
Graph::Graph(int V){
    this->V = V;
    adj = new list<int>[V];
}
void Graph::addEdge(int v,int w){
        adj[v].push_back(w);
        adj[w].push_back(v);
// undirected.
}
```

```
void main(){
// Create a graph
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
}
```

```
void Graph::BFS(int s){
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++) // Mark all the vertices as not visited
           visited[i] = false;

    list<int> queue; // Create a queue for BFS
    visited[s] = true; // Mark the current node as visited and enqueue it
    queue.push_back(s);

    list<int>::iterator i;
    while (!queue.empty()){
        s = queue.front(); // Dequeue a vertex from queue and print it
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s. If a adjacent
        has not been visited, then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i){
            if (!visited[*i]){
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
    delete [] visited;
}
```



(a)



(b)

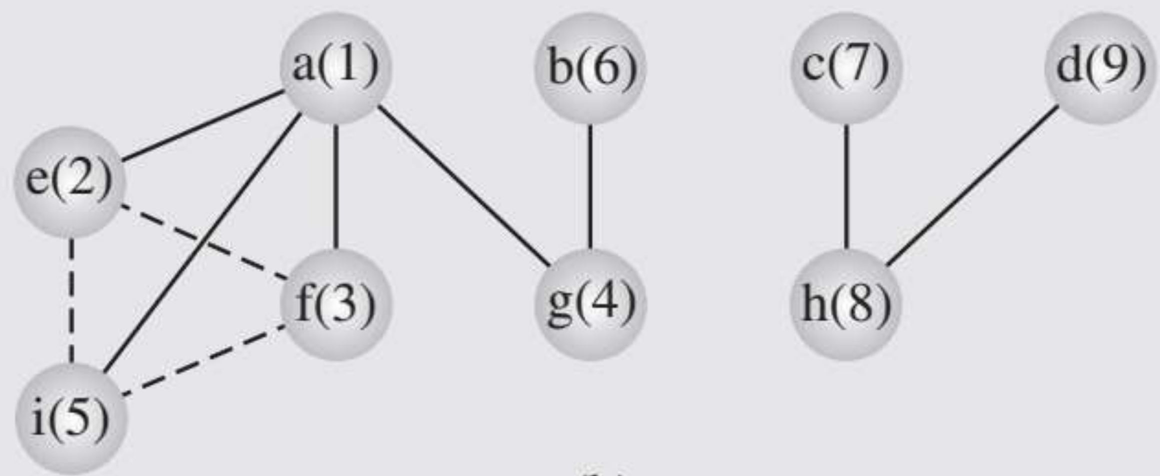
```
void Graph::BFS-Disconnected(){
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++) visited[i] = false;
    list<int> queue; // Create a queue for BFS

    for(int i=0; i<V; i++){
        if (!visited[i]){
            visited[i] = true; // Mark the node as visited and enqueue it
            queue.push_back(i);

            list<int>::iterator ite;
            while (!queue.empty()){
                i = queue.front(); // Dequeue a vertex from queue & print it
                cout << i << " ";
                queue.pop_front();
        // Get all adjacent vertices of the dequeued vertex i. If an adjacent
        vertex is not visited,then mark it visited and enqueue it
                for (ite = adj[i].begin(); ite != adj[i].end(); ++ite)
                    if (!visited[*ite]){
                        visited[*ite] = true;
                        queue.push_back(*ite);
                    }
            }
        }
    }
    delete [] visited;
}
```

```
void Graph::BFS-VertexORDER(){
    int *visited = new int [V]; int order=1;
    for (int i = 0; i < V; i++) visited[i] = 0;
    list<int> queue; // Create a queue for BFS

    for(int i=0; i<V; i++){
        if (visited[i] == 0){
            visited[i] = order++;; //Mark the node as visited & enqueue it
            queue.push_back(i);

            list<int>::iterator ite;
            while (!queue.empty()){
                i = queue.front(); // Dequeue a vertex from queue & print it
                cout << i << " ";
                queue.pop_front();
        // Get all adjacent vertices of the dequeued vertex i. If an adjacent
        vertex is not visited,then mark it visited and enqueue it
                for (ite = adj[i].begin(); ite != adj[i].end(); ++ite)
                    if (!visited[*ite]){
                        visited[*ite] = order++;
                        queue.push_back(*ite);
                    }
            }
        }
    }
    delete [] visited;
}
```

Application BFS

Shortest Path for Unweighted Graph

- In an unweighted graph, the shortest path is the path with least number of edges.
- With BFS, we always reach a vertex from given source using the minimum number of edges.



(b)

Shortest Path for Unweighted Graph

- With BFS, we always reach a vertex from given source using the minimum number of edges.
- Example:
  - Let A is a starting point (or train station)
  - We wish to find route to all other stations from A with minimum number of hops (or stations in between)
  - DFS will not give us shortest path from A to all other nodes
  - However BFS will give us shortest path from A to all other nodes

BFS Applications

- **Social Networking Websites:**
  - In social networks, we can find people within a given distance ‘k’ from a person using BFS till ‘k’ levels.

BFS Applications

- **GPS Navigation systems**: BFS is used to find all neighboring locations.

- **Broadcasting in Network**:

In networks, a broadcasted packet follows BFS to reach all nodes with min number of hops.

- **In internet**, we want to download from where there is minimum number of hops.

Path Finding

- **Path Finding**
  - We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

- **Cycle detection in undirected graph:**
  - In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

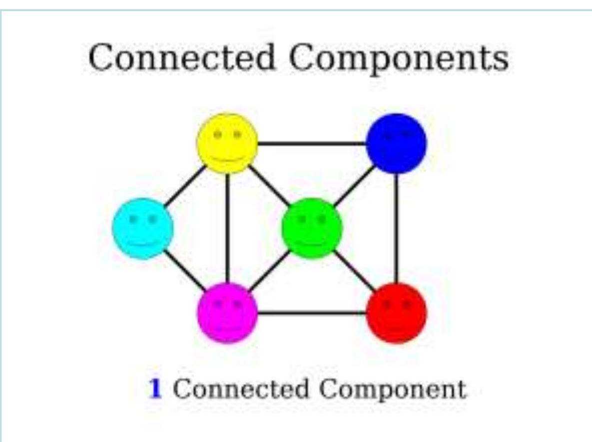
Connected Components

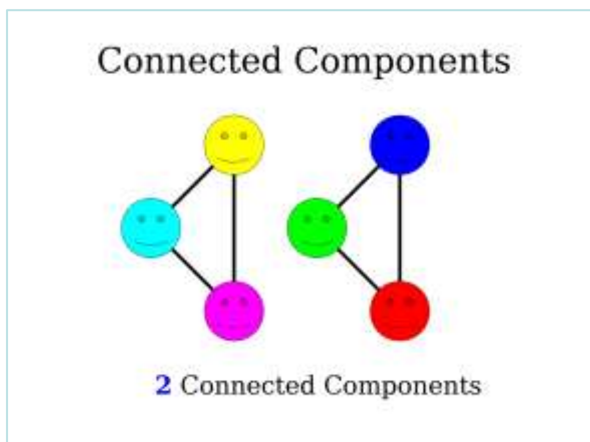
What interesting things can we do with graphs

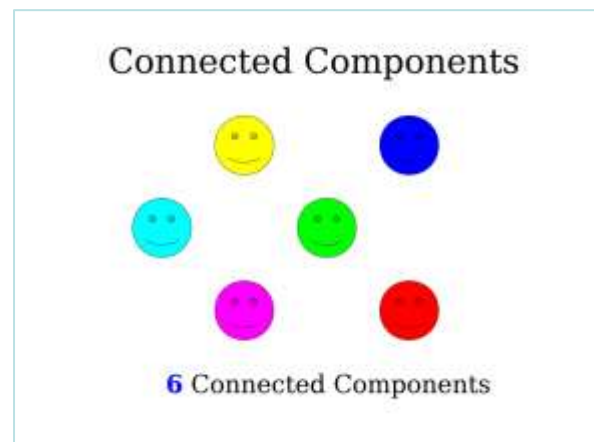
Connected components

Connected Components

- A **connected component** is a subset of the nodes in a graph such that:
  - For every pair of nodes in the subset there exists a path between them
  - No node in the subset is not connected any node not in the subset







Connected Components

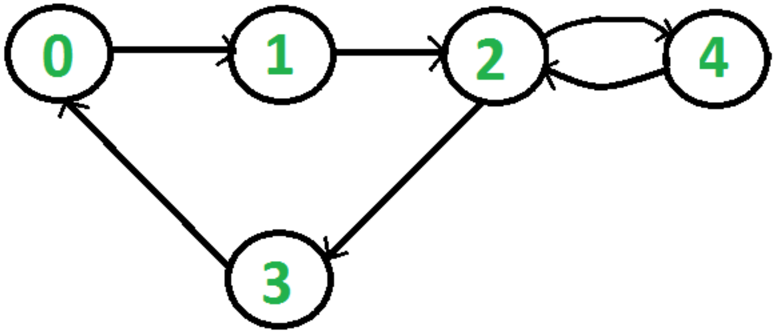
- Detecting connected components in a graph is important because it can provide useful insights into the structure of graph
- How do people in a community separate themselves into separate groups.
- In order to detect connected components, we first need to be able to iterate over nodes in a graph.

Connected Graph

- An undirected graph is **connected** if there is a path from every vertex to every other vertex.
- A directed graph with this property is called **strongly connected.**
- If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be **weakly connected.**

Directed graph

- A directed graph is strongly connected if there is a path between any two pair of vertices. For example, following is a strongly connected graph.



Strongly Connected

Detect Connected Components

Recursive DFS For Disconnected graph

```
// DFS traversal of the vertices
reachable from v.

void Graph::DFS()
{
// Mark all vertices as not
visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
    visited[i] = false;

// Call the recursive helper
function to print DFS traversal
starting from all vertices one
by one
 for (int i = 0; i < V; i++)
     if (visited[i] == false)
         DFSHelp(i, visited);

 delete []visited;
}
```

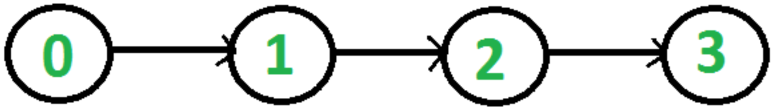
```
void Graph::DFSHelp(int v,bool visited[])
{
// Mark the current node as visited and
// print it
    visited[v] = true;
    cout << v << " ";

// Recur for all the vertices adjacent
// to this vertex
    list<int>::iterator i;
  for(i=adj[v].begin(); i!=adj[v].end();++i){
        if (!visited[*i])
            DFSHelp(*i, visited);
  }
}
```

UnDirected graph

- **It is easy for undirected graph**, we can just do a BFS and DFS starting from any vertex.
- If BFS or DFS visits all vertices, then the given undirected graph is connected.
- This approach won't work for a directed graph.

Directed graph

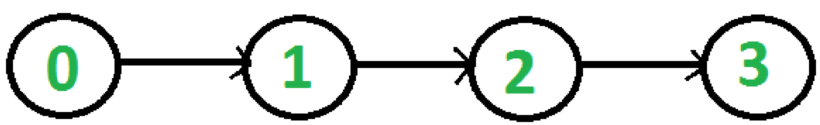
- Consider the following graph which is not strongly connected.
- If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.



Not Strongly Connected

Directed graph

- 1) Initialize all vertices as not visited.
- 2) Run DFS on the graph starting from any vertex v.
  - If DFS traversal doesn't visit all vertices, then return false.
- 3) Reverse all arcs (or find transpose or reverse of graph)
- 4) Mark all vertices as not-visited in reversed graph.
- 5) Run DFS on reversed graph starting from same vertex v (Same as step 2).
  - If DFS traversal doesn't visit all vertices, then return false. Otherwise return true.



Not Strongly Connected