# Graphs I

- Applications
- Terminology
- Graph API
- Implementations

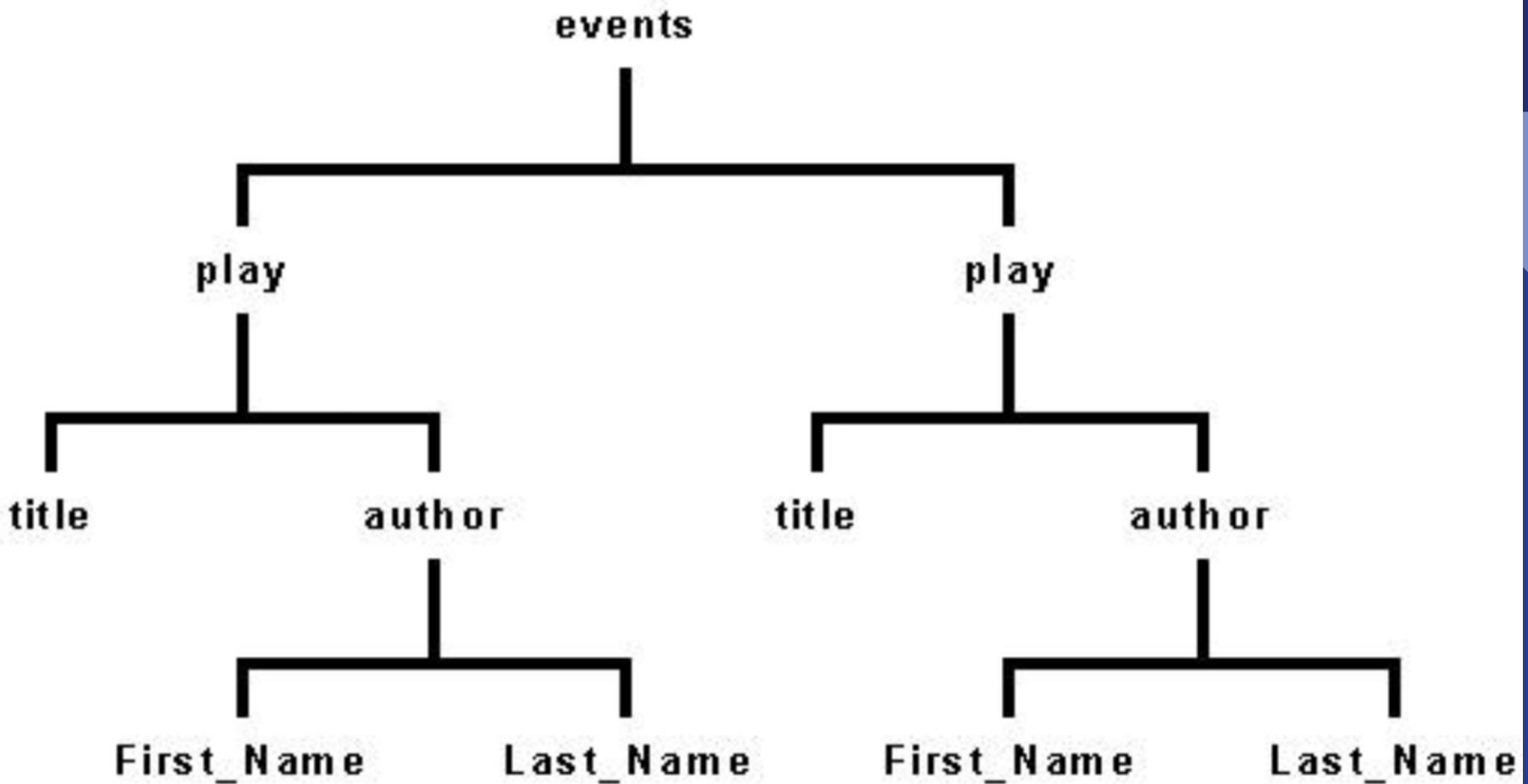
# What is a graph?

set of nodes (vertices) with connections (edges)

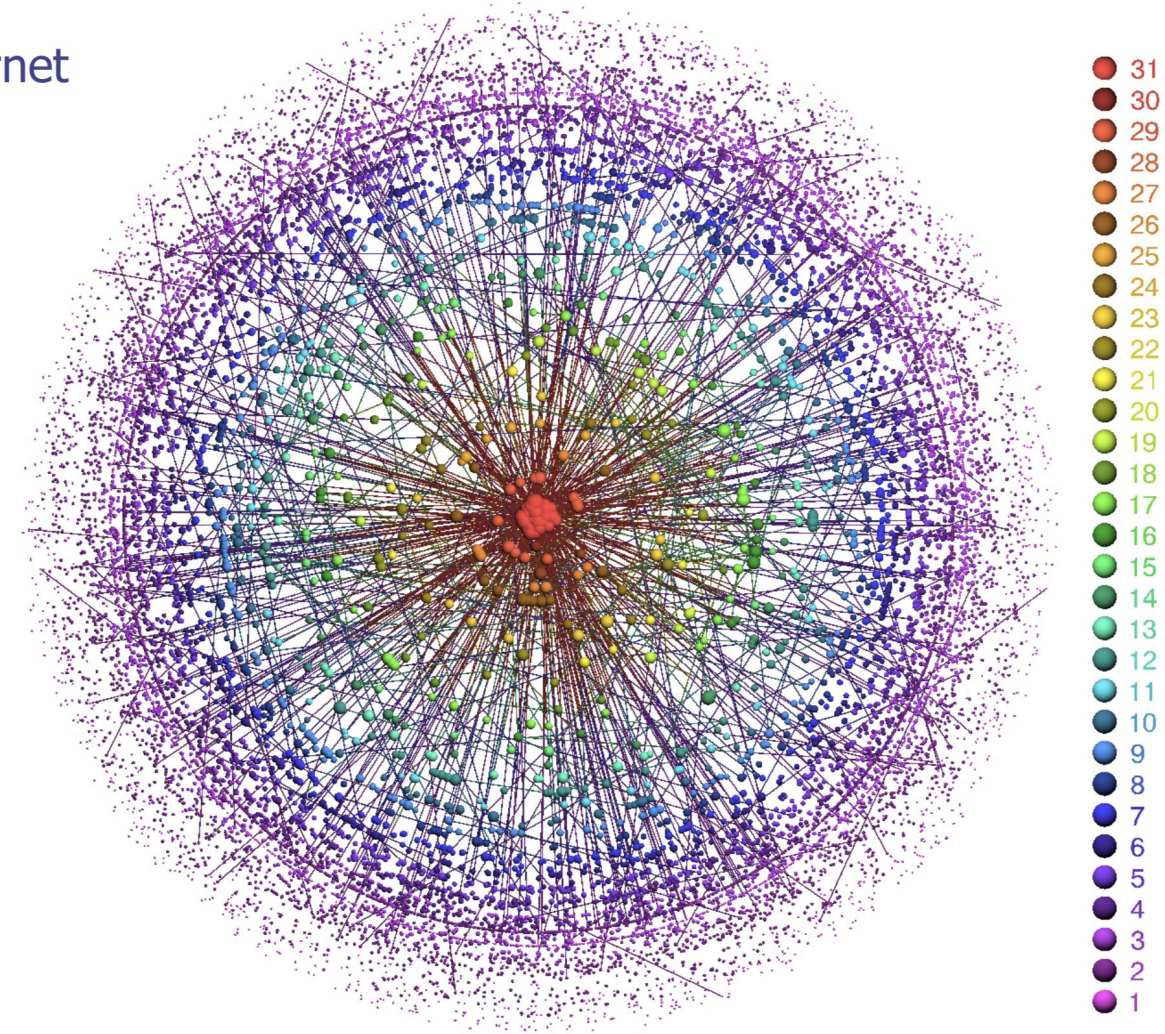
can be undirected OR directed

can be weighted

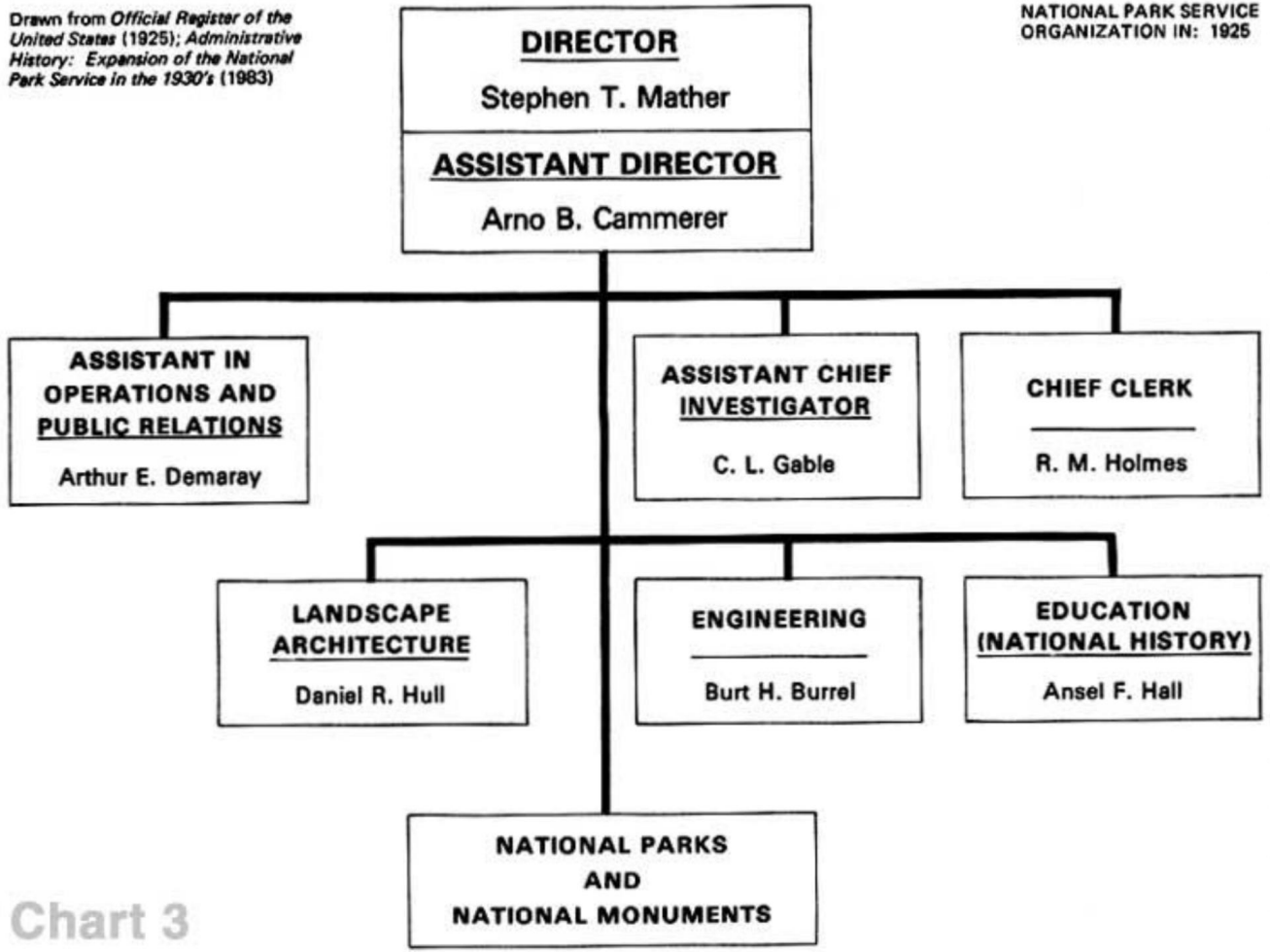
TSP → NP - complete


events
play
play
title
author
title
author
First_Name
Last_Name
First_Name
Last_Name

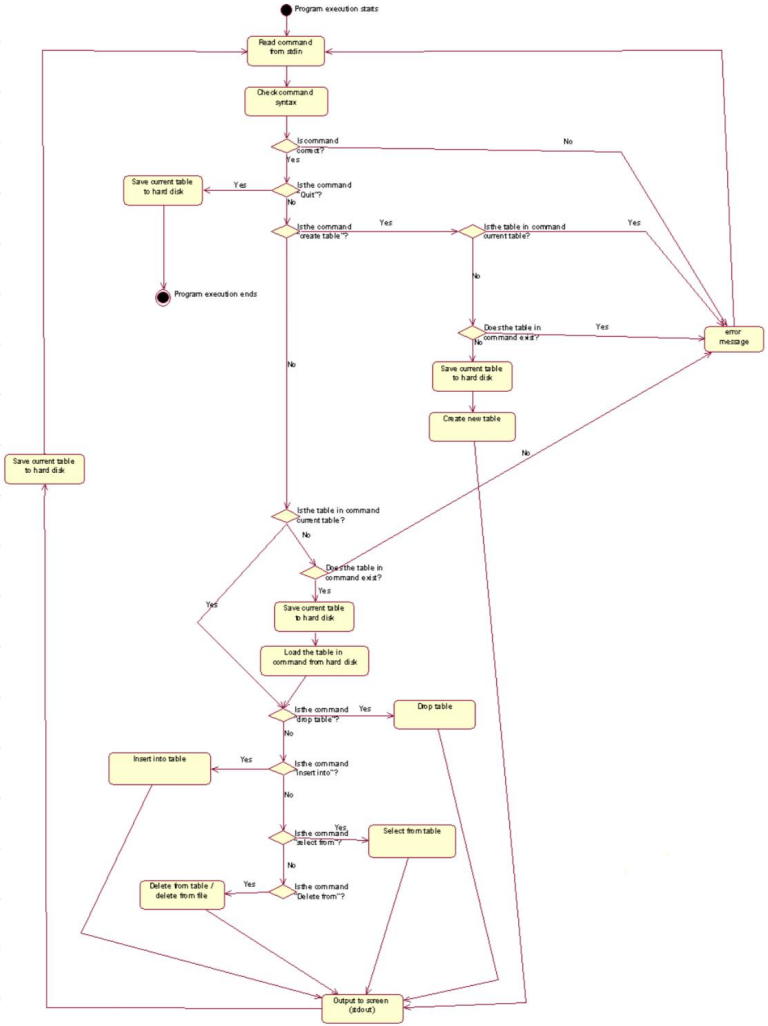
# Internet


31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1

# Organizational chart

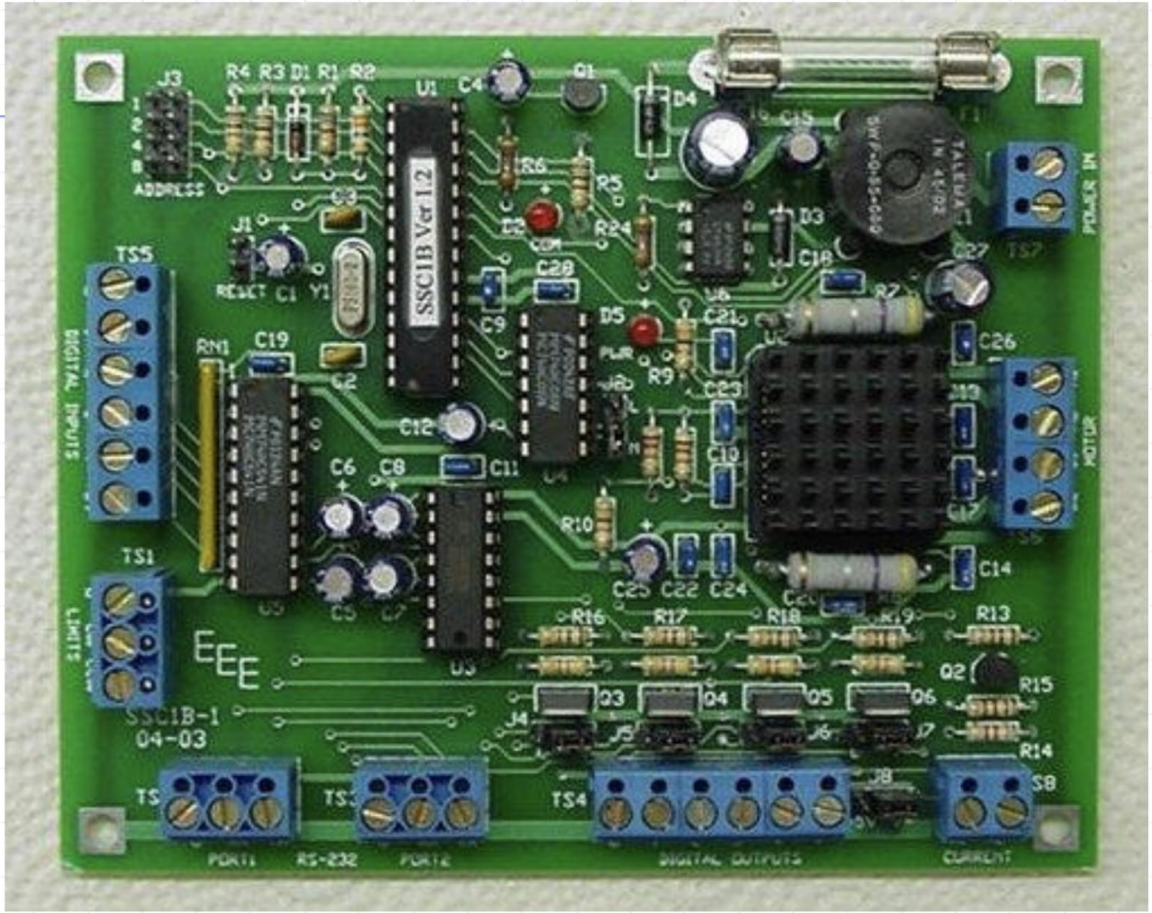


# Program Flow Chart

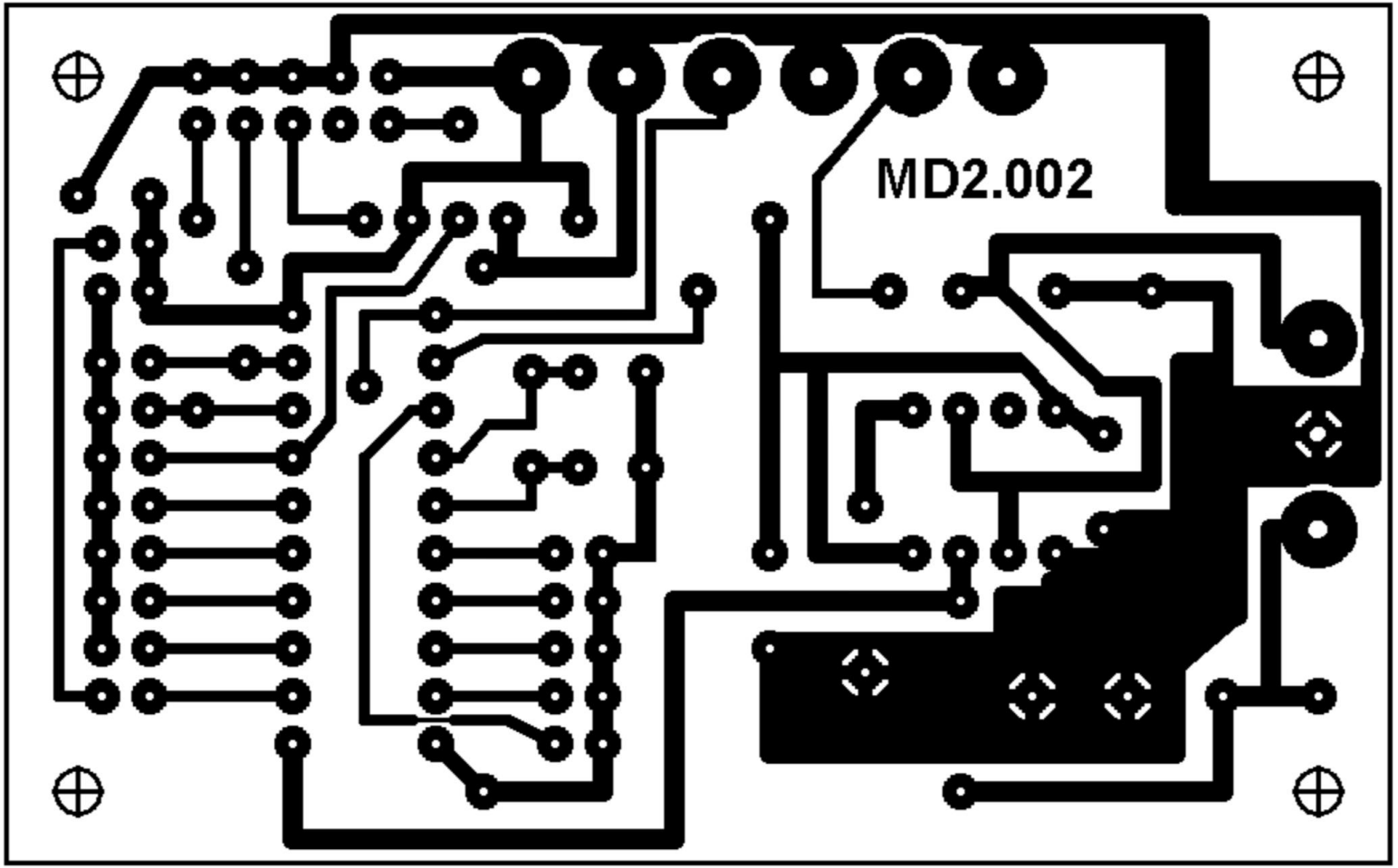


# Flight Routes

# Electronic Circuit Board



# Electronic Circuit Board



# What are some applications of graphs?

social network

shipping

dependencies

path finding

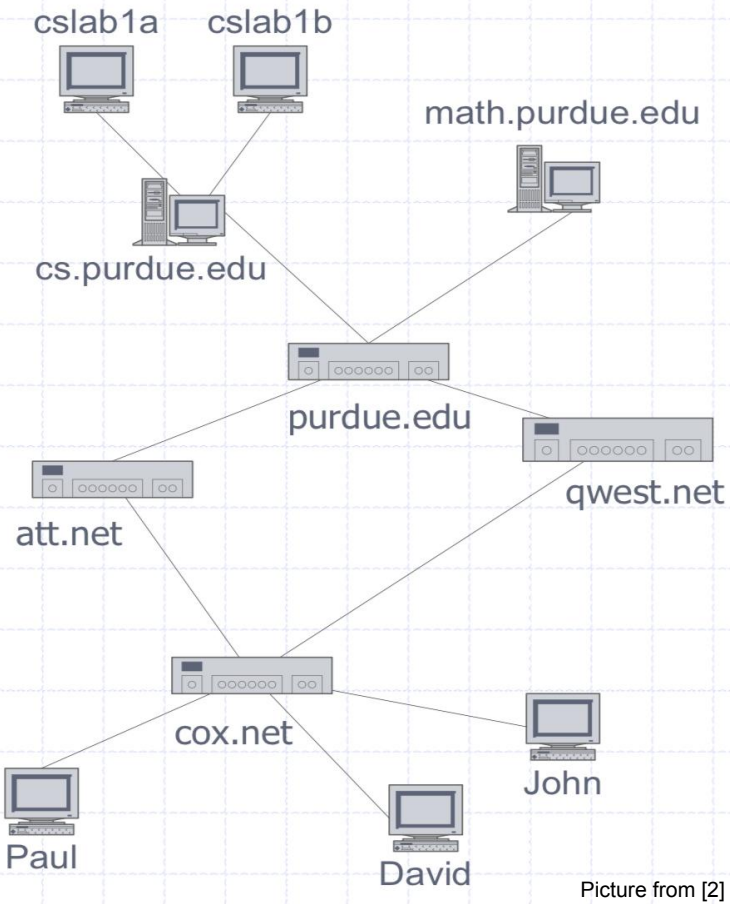
Recommendations

connected universes → connected

Union-find

# Applications

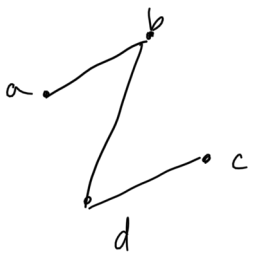
- Computer networks
  - Local area network
  - Internet
  - Web
- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Databases
  - Entity-relationship diagram

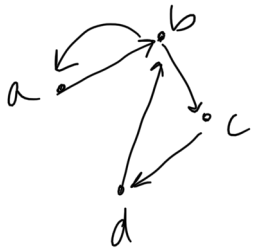



Picture from [2]

# Graph Types & Terminology

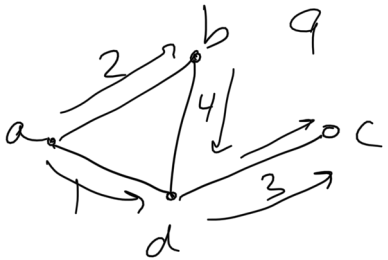
undirected vs directed




$(a,b) \neq (b,a)$

$(a,b) = (b,a)$




9

4

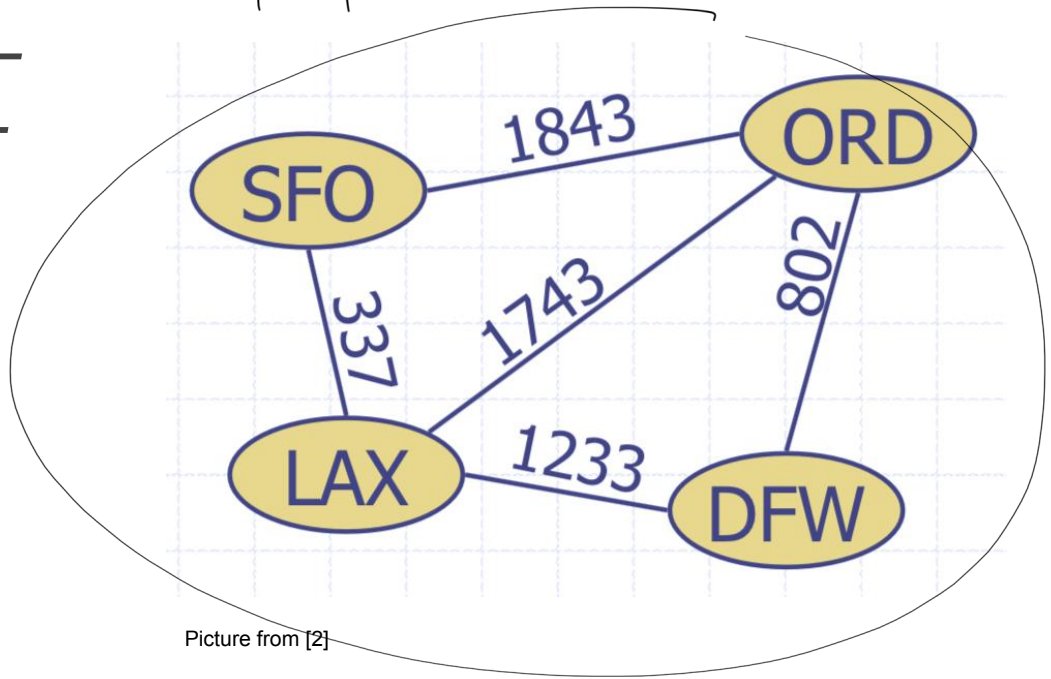
# What is a graph?

A ***graph*** is a set of vertices and a collection of edges that connect pairs of vertices.

$|V| = 4$

$|E| = 5$

undirected




Picture from [2]

# Edges

- Edges can be directed or undirected.
- Edges can be weighted or unweighted.
- The choice between these is dependent on the context.
- An directed graph is made up of directed edges, and a undirected graph is made up of undirected edges.
- A weighted graph can be either directed or undirected and is made up of weighted edges.
- Edges are often represented as ordered pairs. Does the order matter?

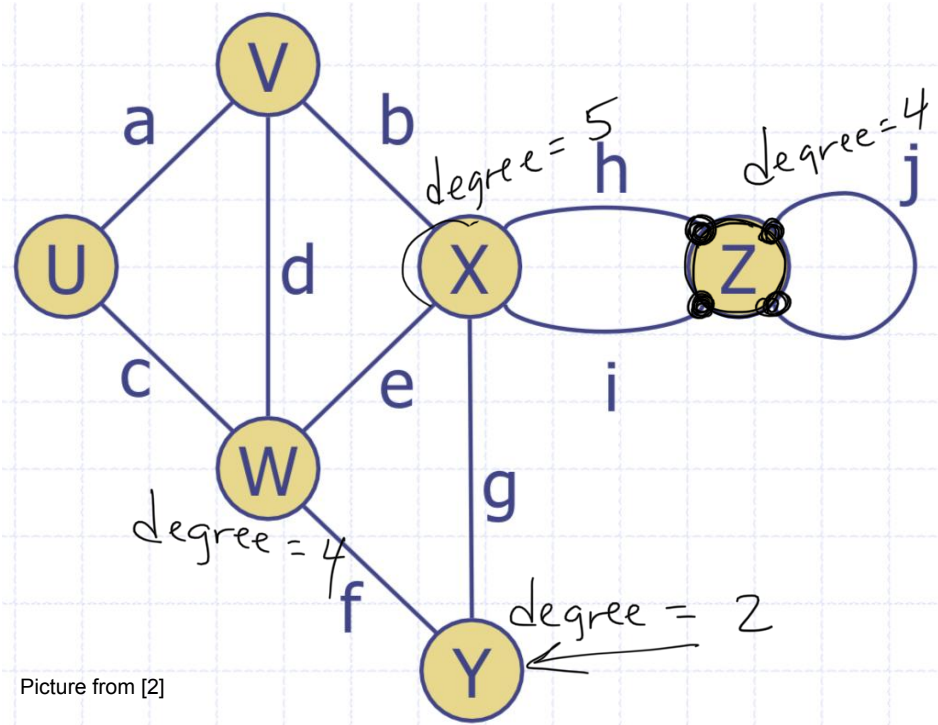
What are some applications of each of these?

# Terminology

The endpoints of $a$ : $u, v$

$a = (u, v) = (v, u)$

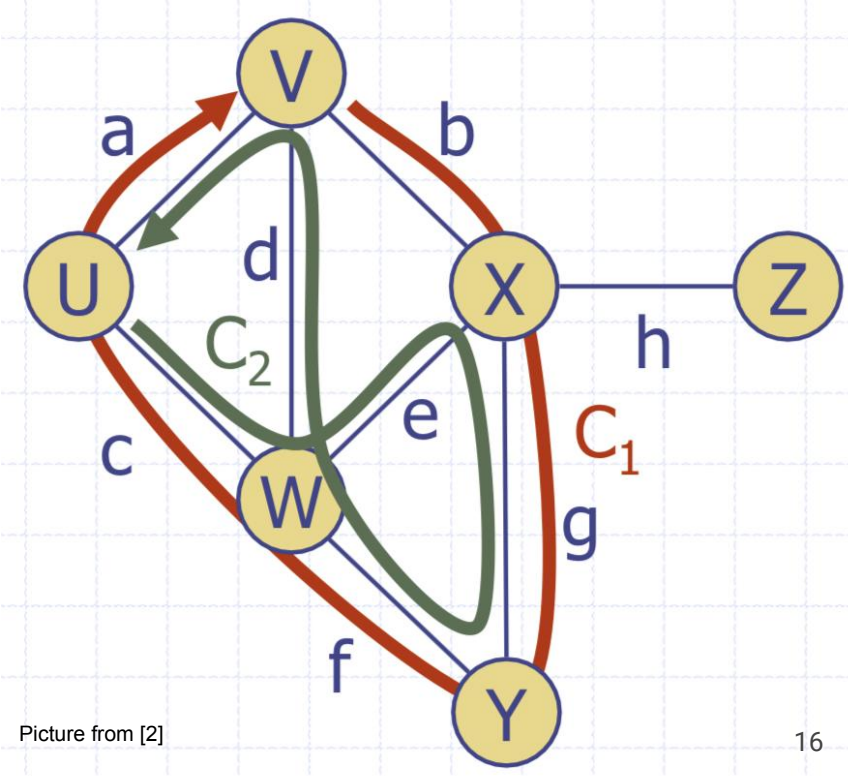
- The vertices that are connected by an edge are called the endpoints of the edge.
- When an edge touches a vertex, we say the edge is incident on on that vertex.
- Two vertices that are connected with an edge are said to be adjacent (to each other).
- The number of edges incident on a vertex is called the degree of the vertex.
- Two edges with the same endpoints are called parallel edges.
- An edge whose endpoints are the same vertex is called a self loop.
- A graph with no parallel edges and no self loops is called simple.




Picture from [2]

# More Terminology

- A sequence of vertices connected by edges is called a path.
- A path with no repeated edges or vertices is called simple.
- A circular sequence of vertices connected by edges is called a cycle.
- A cycle with no repeated edges or vertices (except the first and last) is called simple.
- The length of a path or cycle is the number of edges included in the path or cycle.
- A subset of a graph's edges and vertices is called a subgraph.
- A connected graph is one where there is a path connecting any two pairs of vertices.
- An acyclic connected graph is called a tree.
  → no cycles



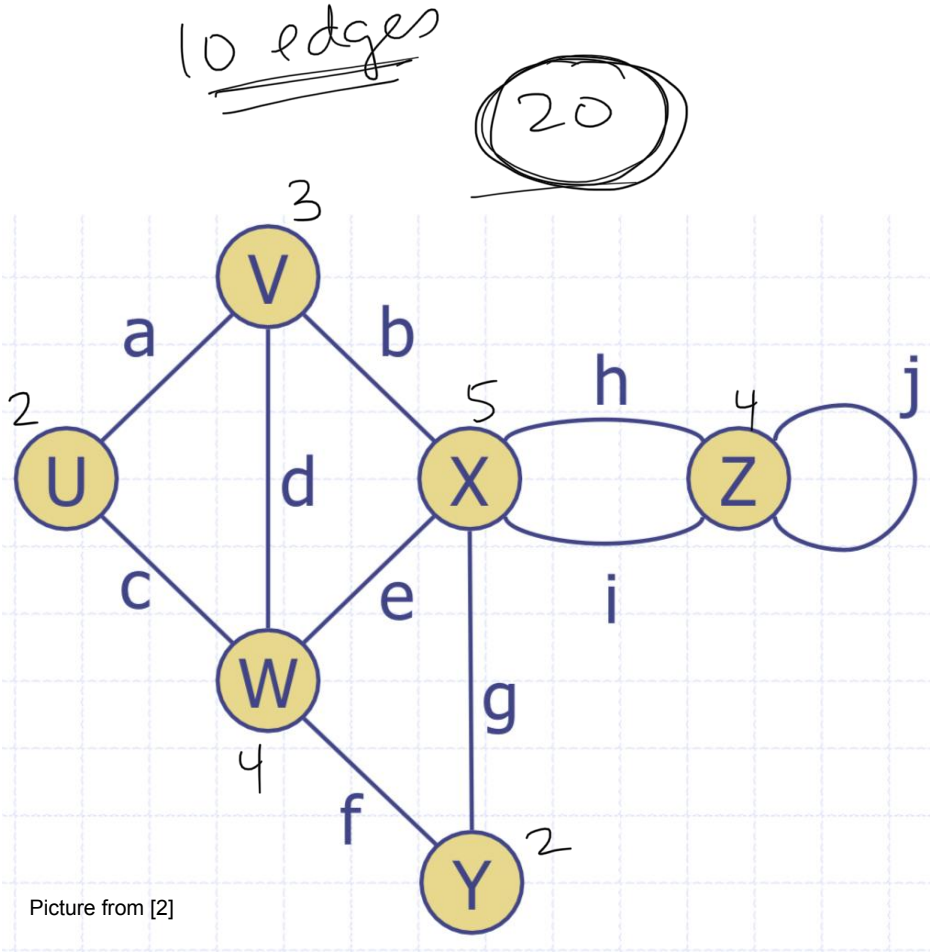

Picture from [2]

# Properties of Graphs

**Property 1: Degree Sum**

Given a graph with **V** vertices and **E** edges, what is the relationship between the sum of the degrees of all the vertices and the number of edges?

$$\sum_{v \in V} \deg(v) = 2|E|$$



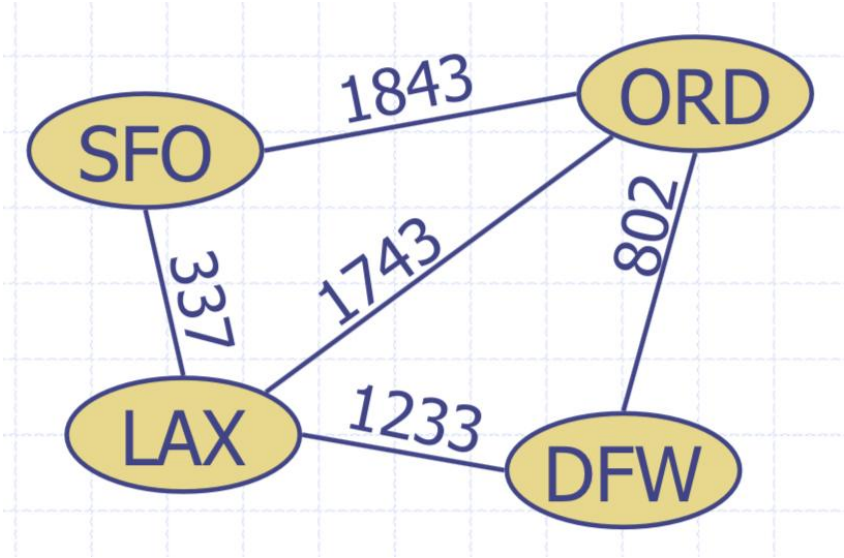

Picture from [2]

# Properties of Graphs

**Property 2: Maximum Degree**

Given a simple, undirected graph with **V** vertices and **E** edges, what is the maximum possible degree of any vertex?

$|V| - 1$




Pictures from [2]

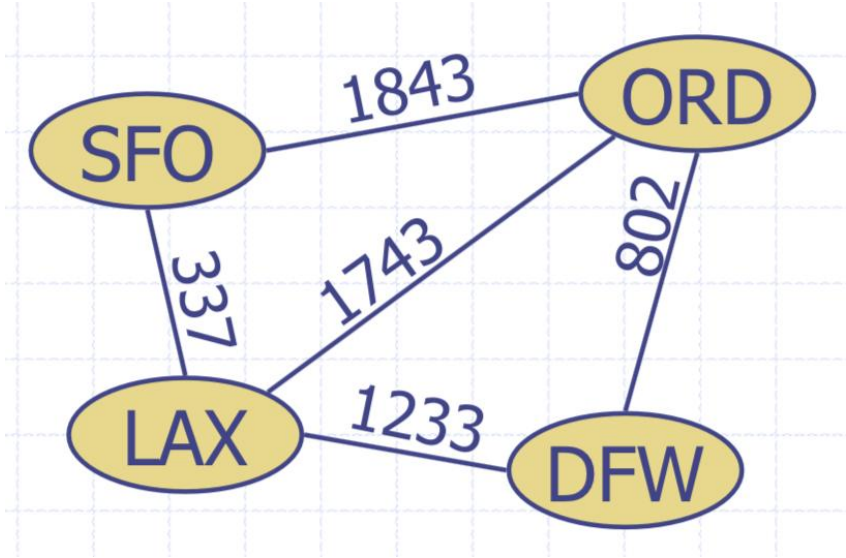
# Properties of Graphs

$$\binom{V}{2} = \frac{V(V-1)}{2}$$

**Property 3: Edge Count**

Given a simple, undirected graph with **V** vertices and **E** edges, what is the maximum possible number of edges in terms of **V**?

$$|E| \leq \frac{|V|(|V|-1)}{2}$$




Pictures from [2]

# Undirected Graph API

| public class Graph | |
|---|---|
| Graph(int V) | *create a V-vertex graph with no edges* |
| Graph(In in) | *read a graph from input stream* in |
| int V() | *number of vertices* |
| int E() | *number of edges* |
| void addEdge(int v, int w) | *add edge* v-w *to this graph* |
| Iterable<Integer> adj(int v) | *vertices adjacent to* v |
| String toString() | *string representation* |

**API for an undirected graph**



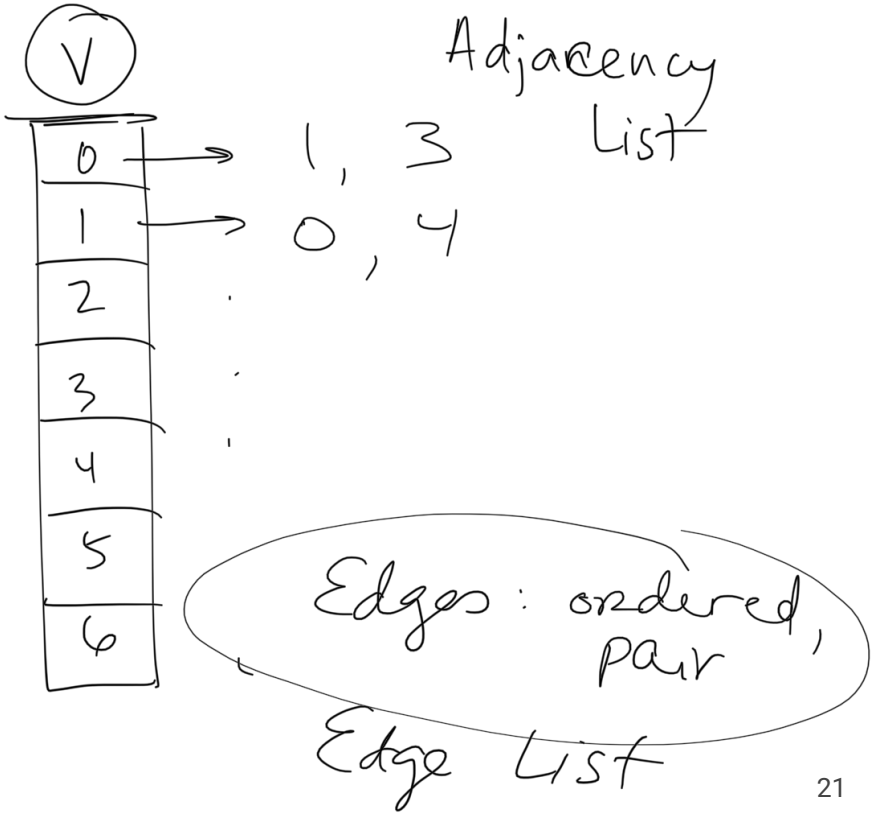
Picture from [3]

# Graph Representation & Implementation

adjacency matrix

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 | 0 |

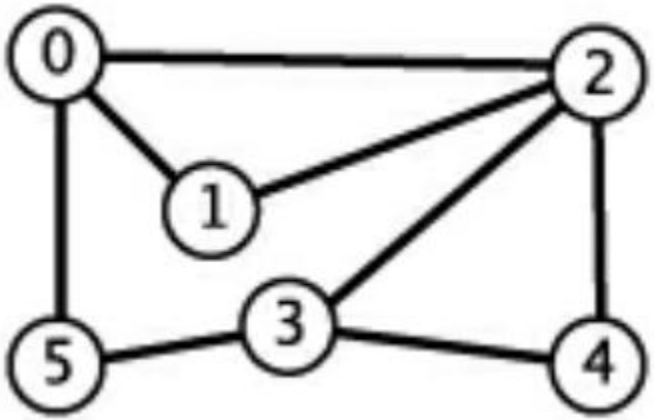
Nodes & pointers



# Edge Lists: *m* edges, *n* vertices

- Represent the graph as a list of edges denoted by their endpoints

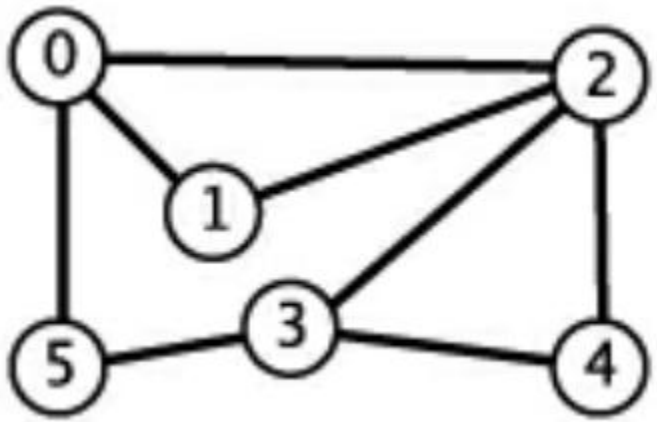
- EXAMPLE: {(0,1), (0,2), (0,5), (1,2), (2,3), (2,4), (3,4), (3,5)}




- Space: $O(m) = O(E)$
- Add edge: $O(1)$
- Check if two vertices are adjacent: $O(m) = O(E)$
- Iterate through the vertices adjacent to a vertex: $O(m) = O(E)$

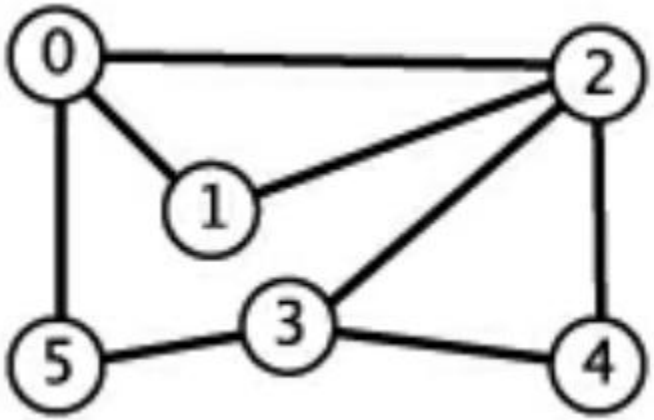
# Adjacency Matrix: *m* edges, *n* vertices

Represent the graph as a matrix where the columns are vertices and the rows are vertices, and the elements are 0's or 1's depending on whether or not there is an edge between the vertices.



# Adjacency Matrix: *m* edges, *n* vertices

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 0 | 0 |
| 5 | 1 | 0 | 0 | 1 | 0 | 0 |



# Adjacency Matrix: *m* edges, *n* vertices

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 0 | 0 |
| 5 | 1 | 0 | 0 | 1 | 0 | 0 |

- Space: $O(n^2) = O(V^2)$
- Add edge: $O(1)$
- Check if two vertices are adjacent: $O(1)$
- Iterate through the vertices adjacent to a vertex: $O(n) = O(V)$

dense → lots of edges

sparse → few edges

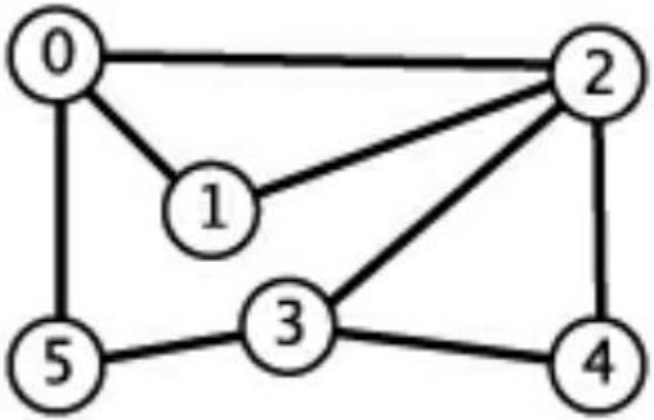
# Adjacency Lists: *m* edges, *n* vertices

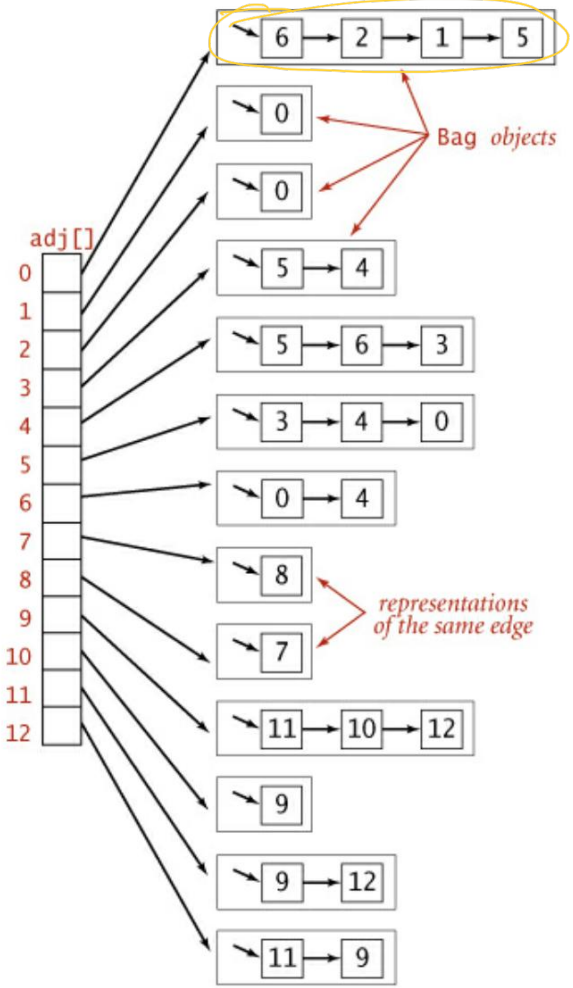
Represent the graph as an array of adjacency lists, where each list indicates the vertices adjacent to one of the **V** vertices.

$n + 2m$

| | |
|---|---|
| 0 | 1, 2, 5 |
| 1 | 0, 2 |
| 2 | 0, 1, 3, 4 |
| 3 | 2, 4, 5 |
| 4 | 2, 3 |
| 5 | 0, 3 |

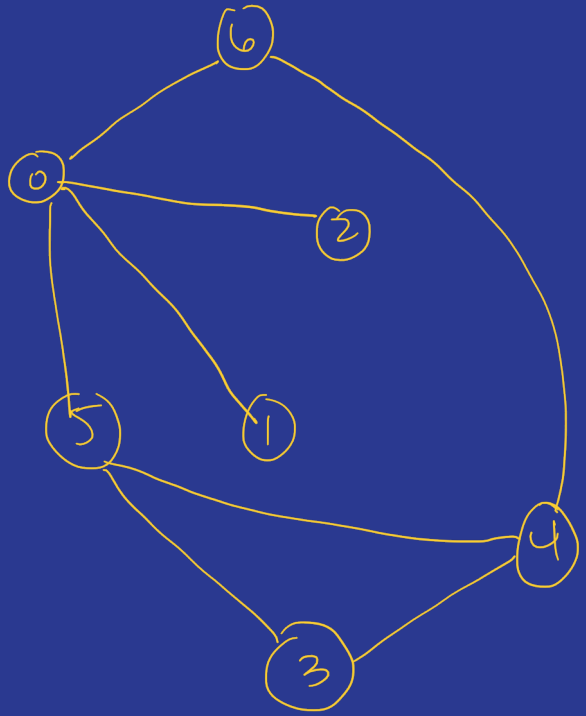
- Space: $O(n+m) = O(V+E)$
- Add edge: $O(1)$
- Check if a vertex is adjacent to another vertex: $O(deg(v))$
- Iterate through the vertices adjacent to a vertex: $O(deg(v))$





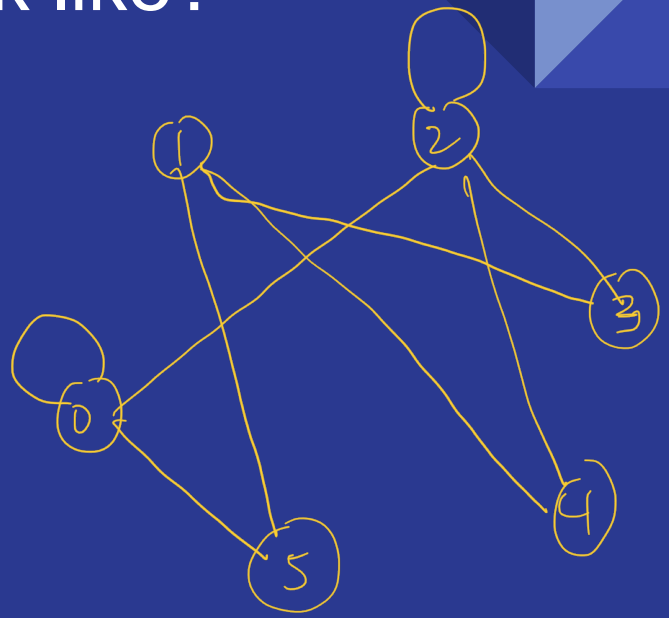

Adjacency-lists representation (undirected graph)

# What does this graph look like?



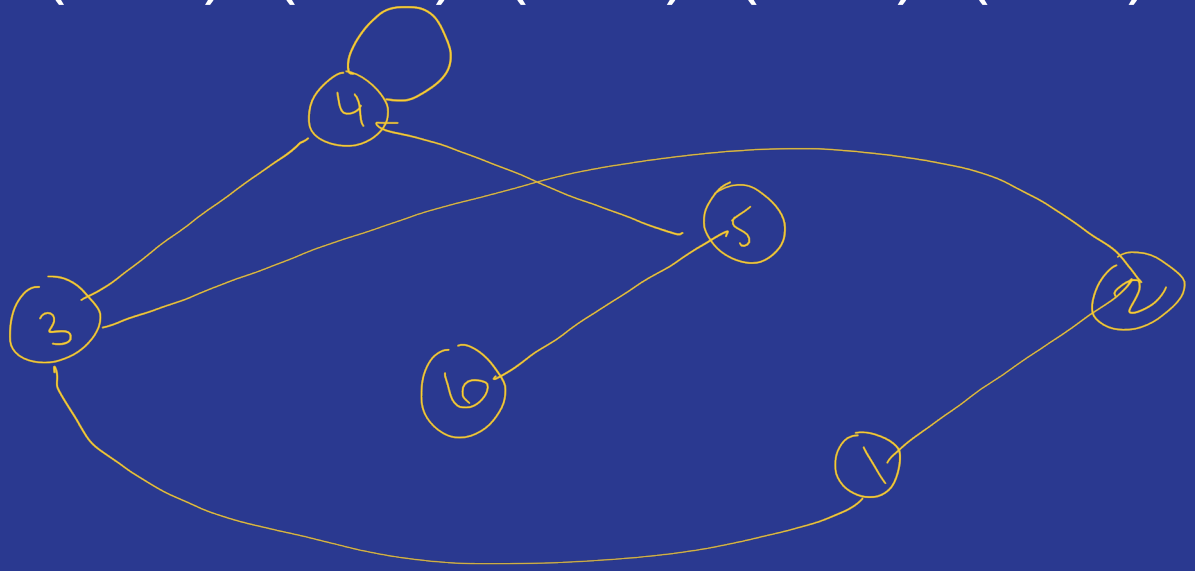
# What does this graph look like?

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 | 0 |
| 5 | 1 | 1 | 0 | 1 | 0 | 0 |



# What does this graph look like?

{(3, 4), (6, 5), (1, 2), (4, 4), (3, 2), (1, 3), (4, 5)}

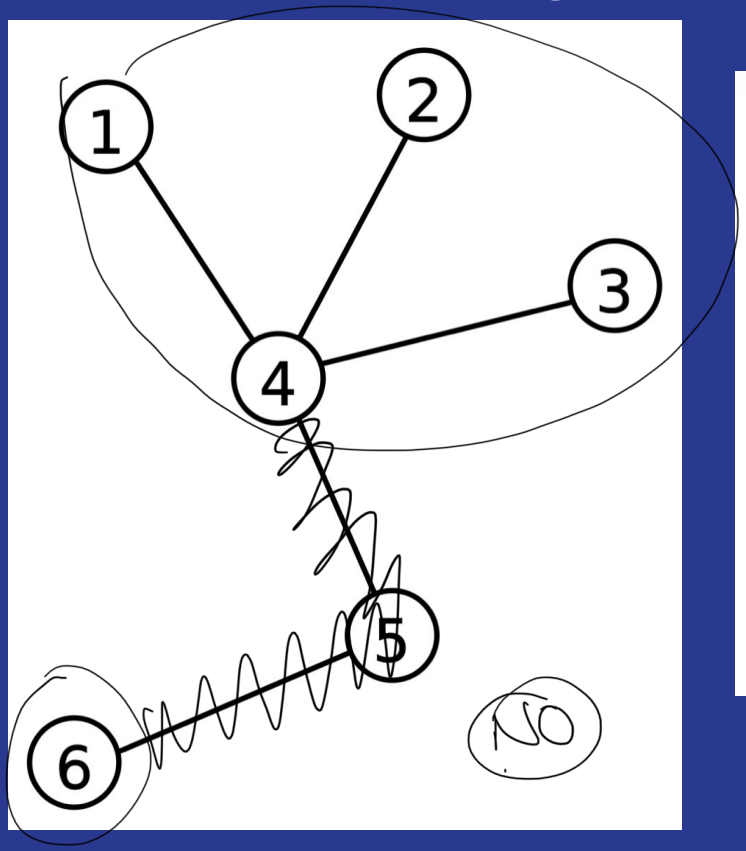


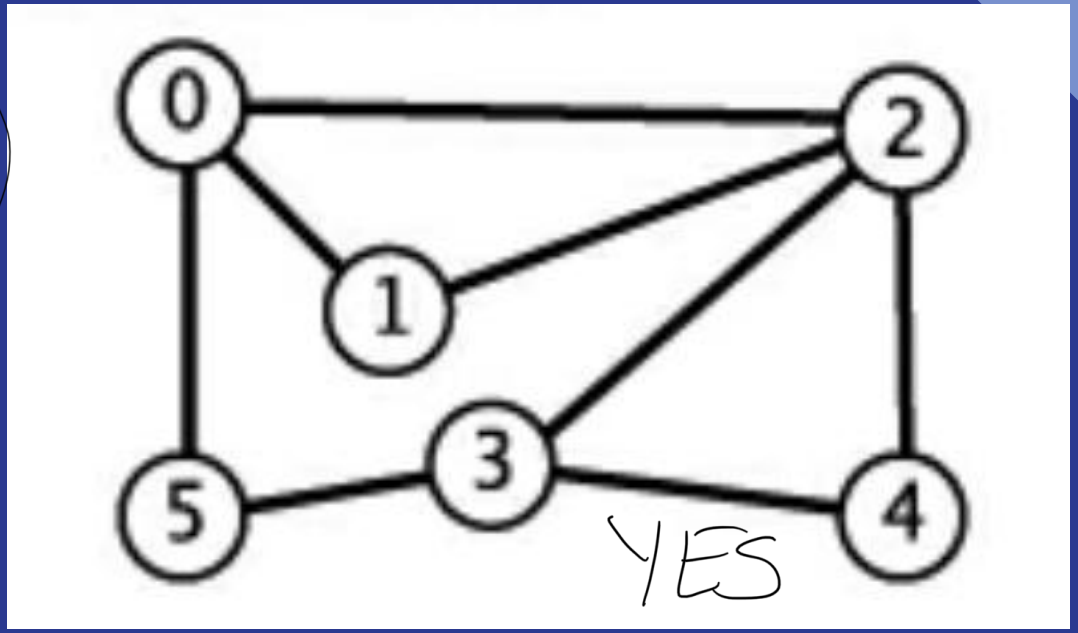
# Biconnectivity

- A *biconnected* graph is a graph that can lose any one vertex and still be connected.

- Another way of putting this is that there are no articulation vertices.

# Are these graphs biconnected?



# References

[1] https://optimization.mccormick.northwestern.edu/index.php/Traveling_salesman_problems
[2] Tamassia and Goodrich
[3] Sedgewick and Wayne