Data Structures

GRAPHS (CHAPTER6)

Google Map

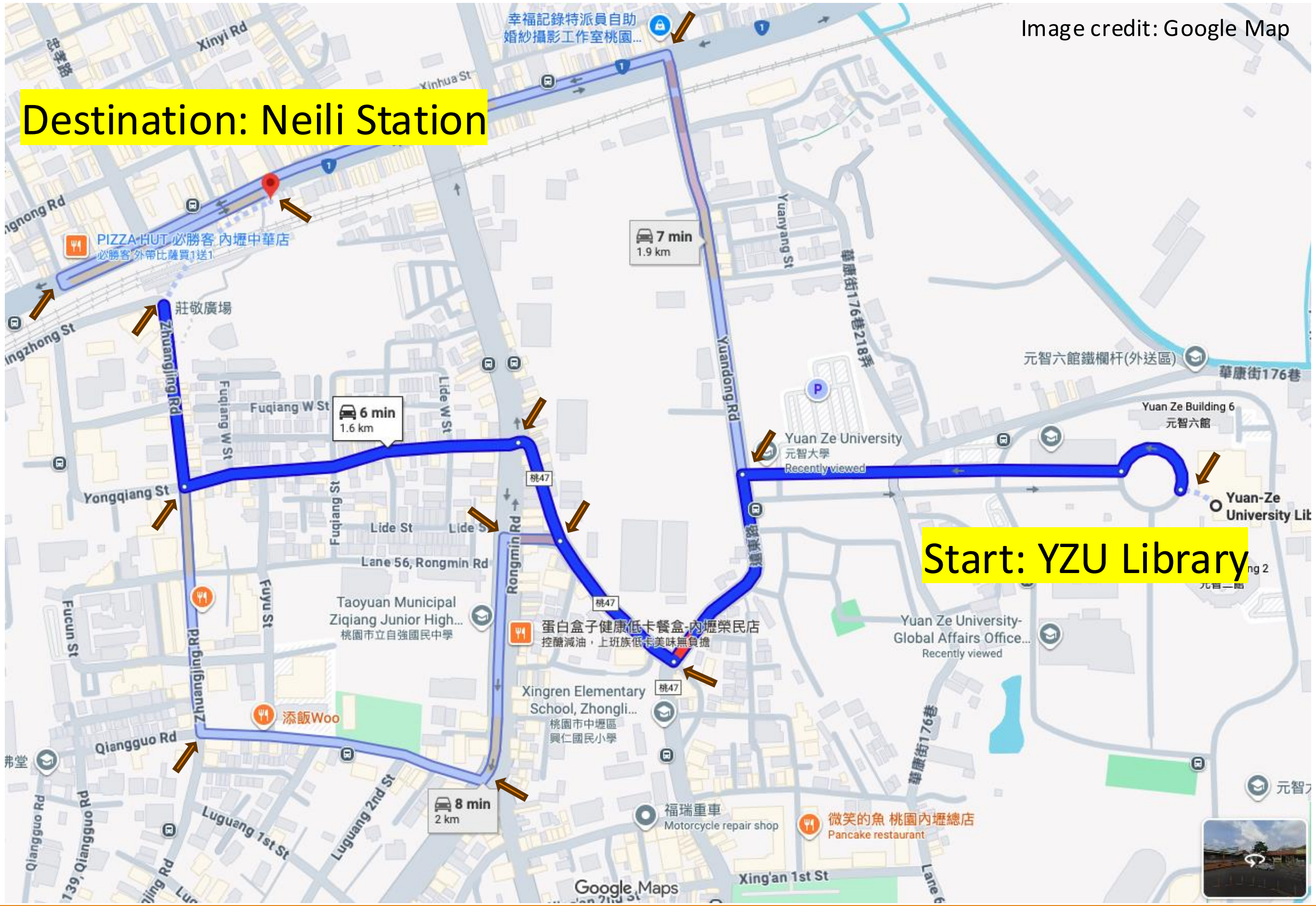


Image credit: Google Map
Destination: Neili Station
Start: YZU Library
7 min
1.9 km
6 min
1.6 km
8 min
2 km

Seven Bridges of Königsberg

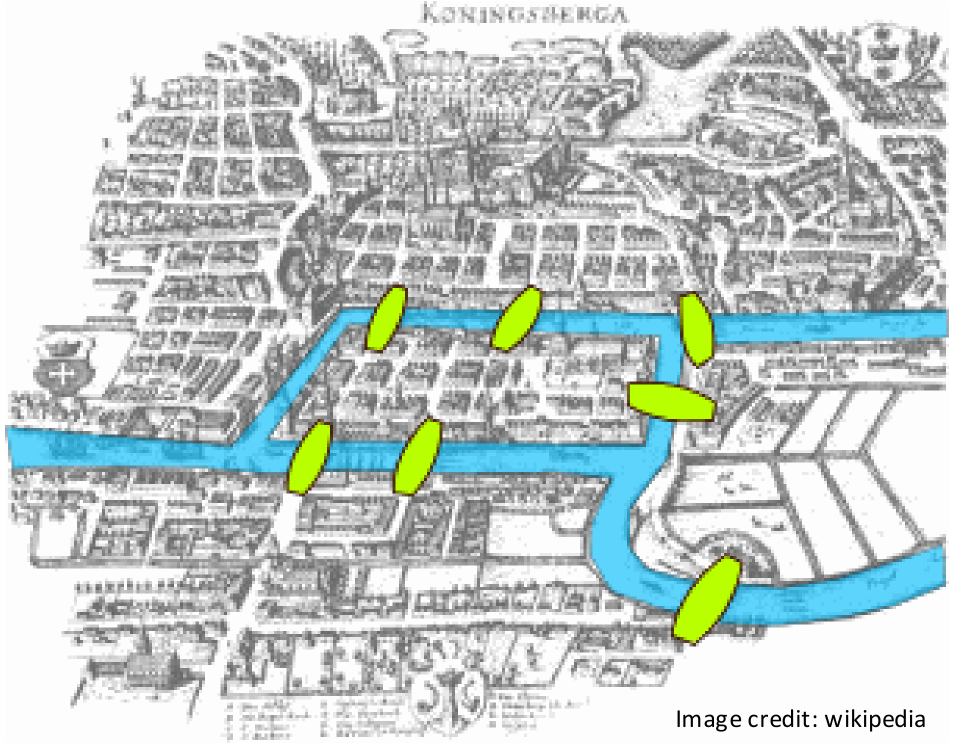


Image credit: wikipedia

Leonhard Euler

Image credit: wikipedia

Graph in Discrete Mathematics

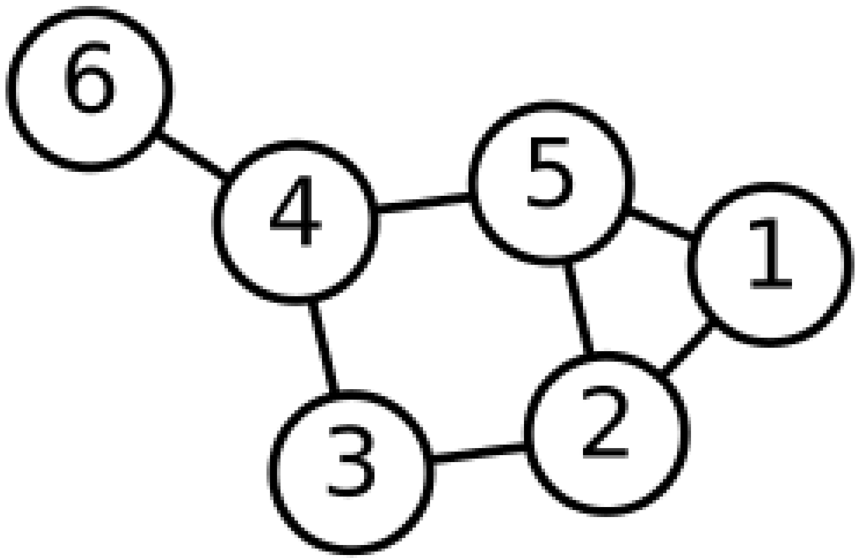


Image credit: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

G = (V, E)

- V: a set of vertices (also called nodes or points)
- $E \subseteq \{ \{x, y\} \in V \text{ and } x \neq y\}$, a set of edges (also called links or lines), which are unorder pairs of vertices

Seven Bridges of Königsberg

Image credit: https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

Any Ideas to Represent this Graph

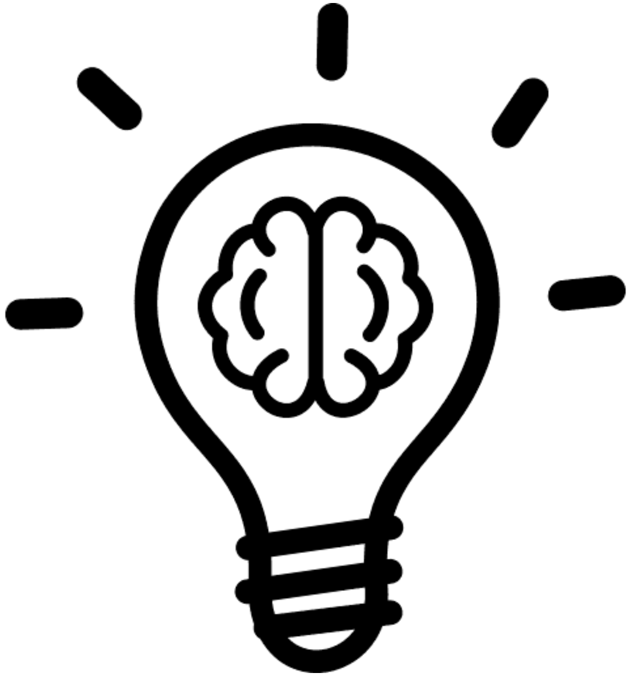
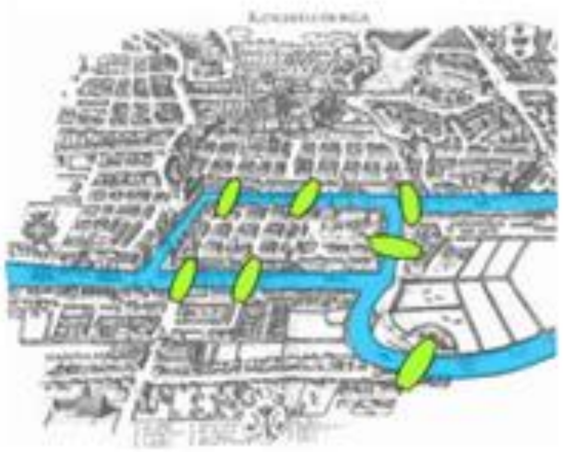
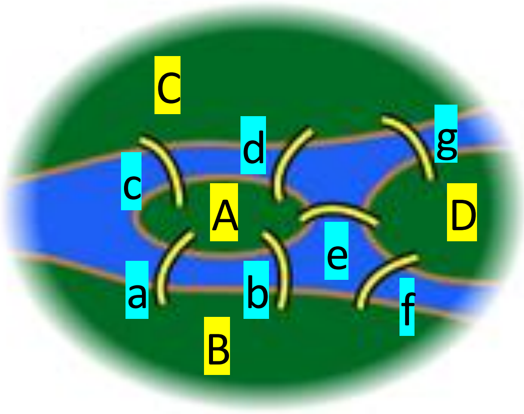


Image credit: https://uxwing.com/idea-icon/

Seven Bridges of Königsberg





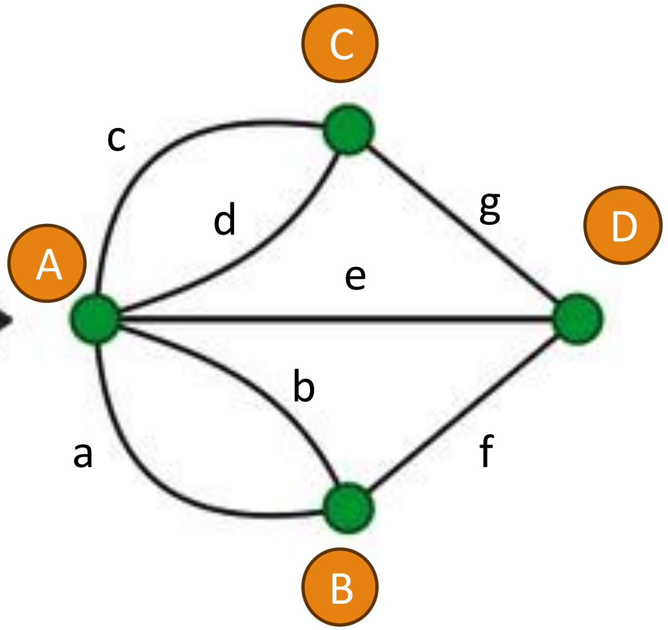


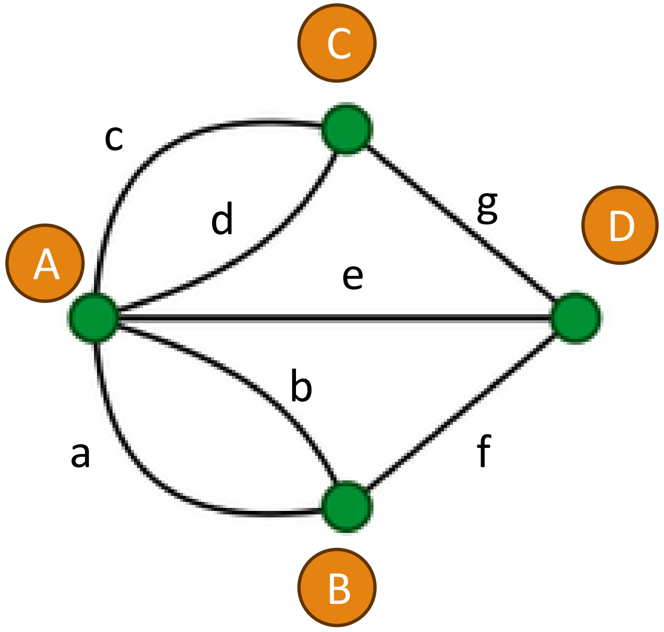
Image credit: https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

Seven Bridges of Königsberg

Adjacency Matrix (by vertex)

Array[*i*][*j*] = number of bridges between vertex *i* and vertex *j*

	A	B	C	D
A	0	0	2	1
B	0	0	2	1
C	2	2	0	1
D	1	1	1	0



Graph

A collection of vertices (nodes) connected by edges that can represent relationships between entities.

Unlike trees, graphs can have cycles and edges can be directed or undirected.

Graphs are used to model networks like social connections, transportation systems, or web pages with hyperlinks.

Node (vertex)

Edge

Graph Components

Node (vertex)

Node (vertex) with label A

Edge

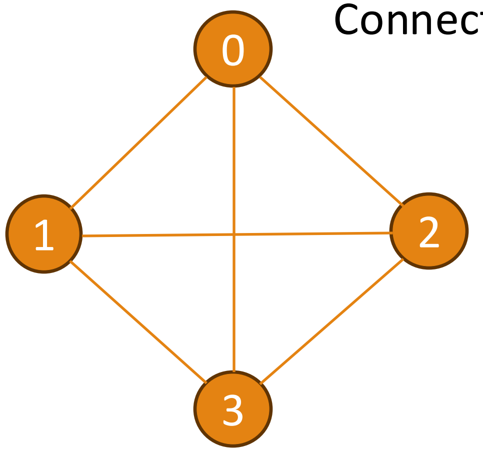
Edge with weight 8

Edge with direction

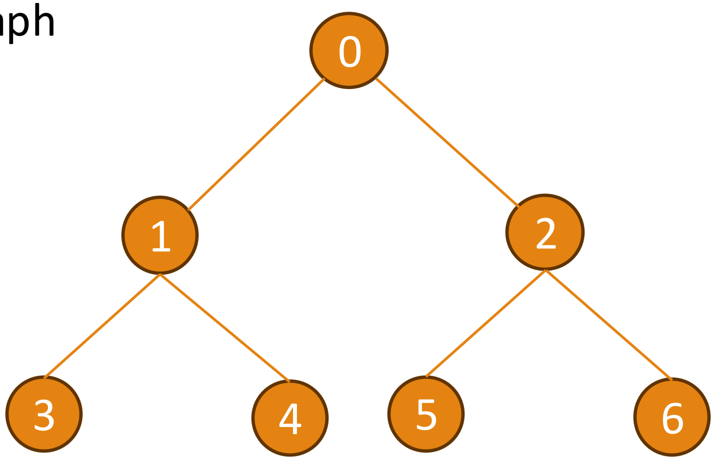
Edge with direction and weight 8

Edge with label e1

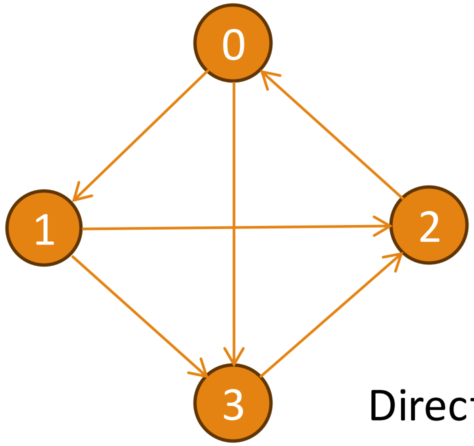
Graph



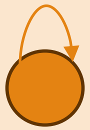
Connected Graph



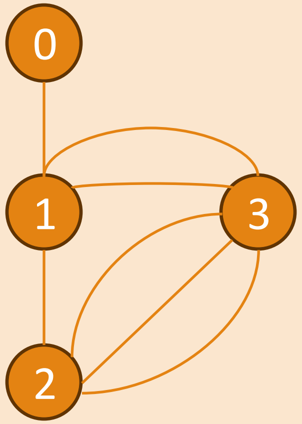
Tree



Directed graph (digraph)



Self edge



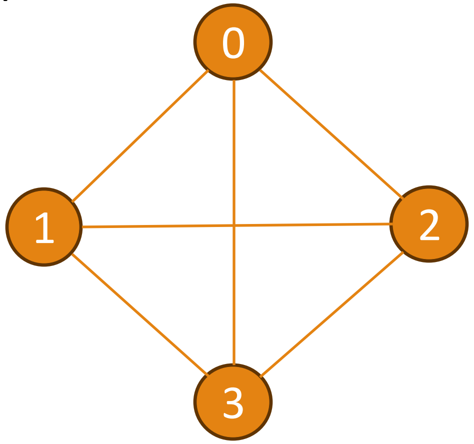
Multigraph

Graphlike Structures

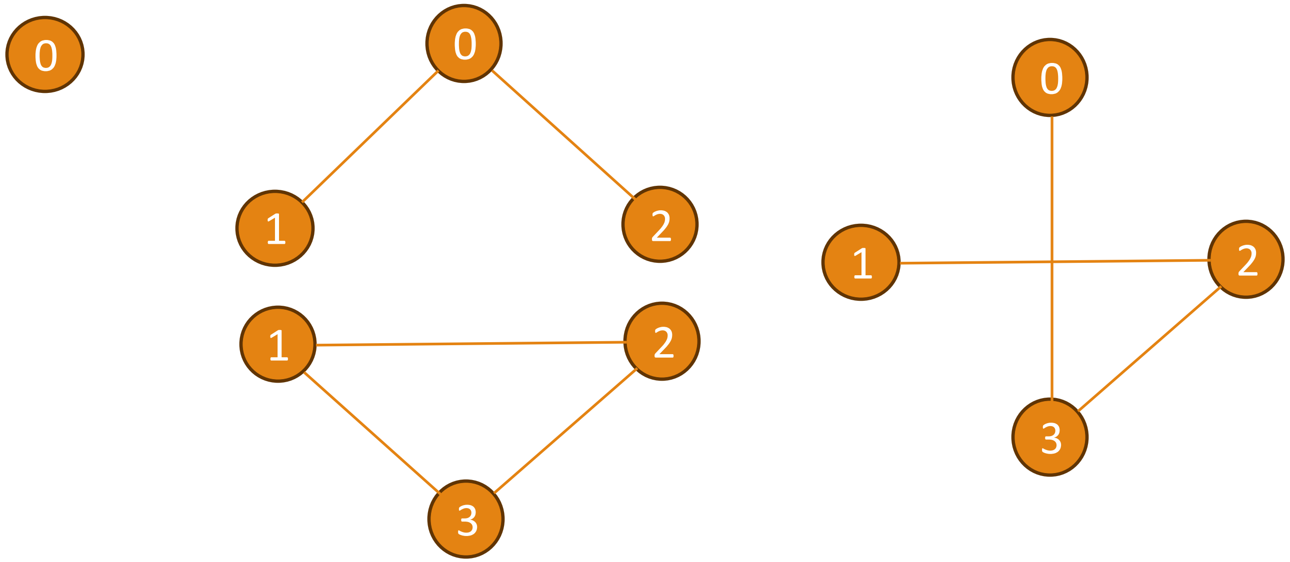
Subgraph

A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G).

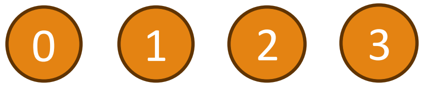
Graph



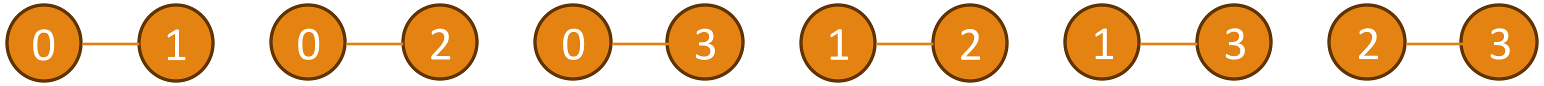
Subgraph



V(G):

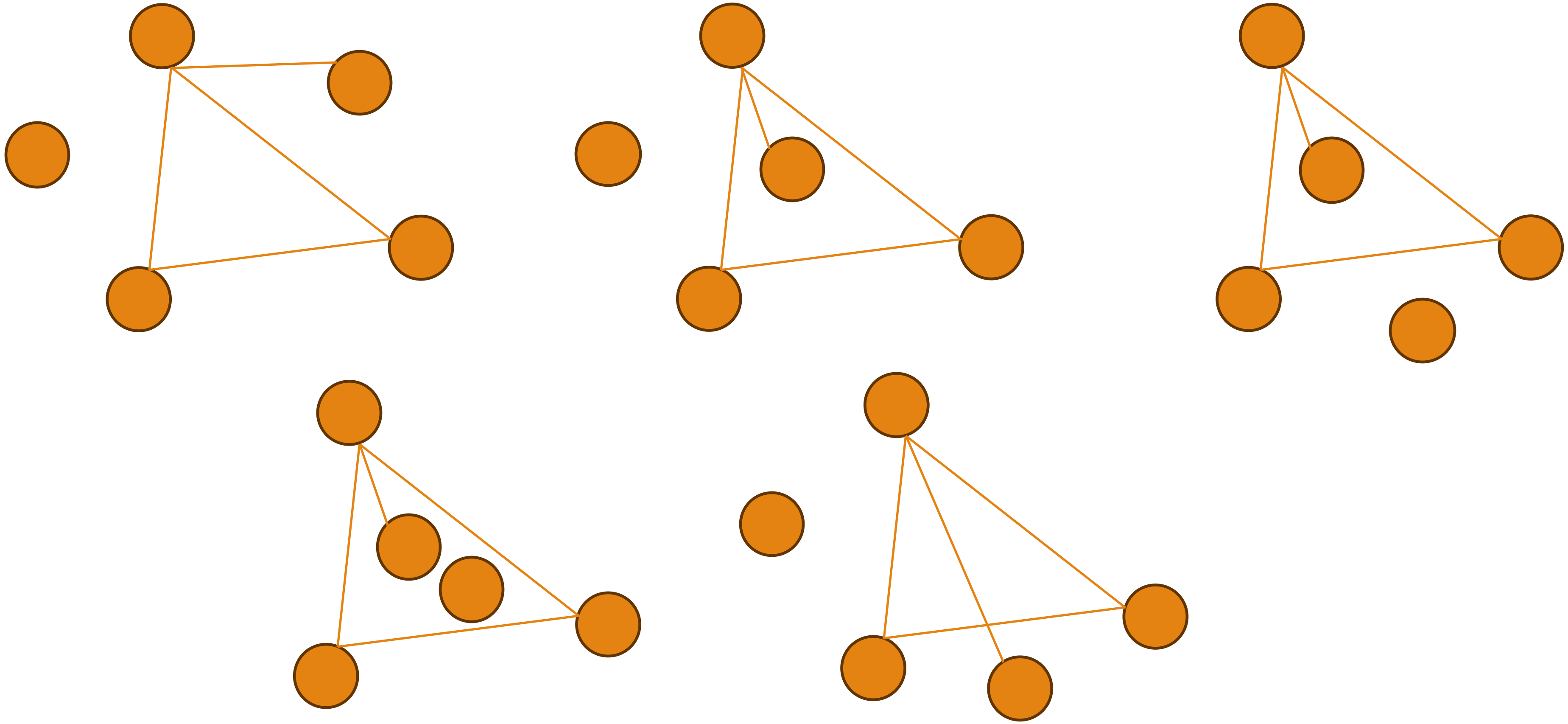


E(G):



Isomorphism / Isomorphic

Original

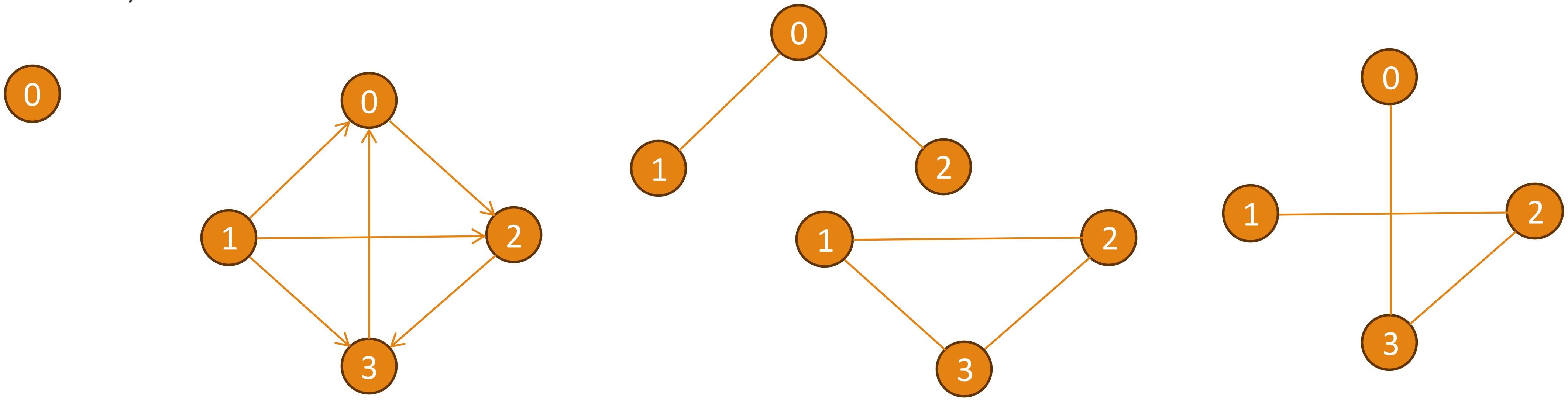


Degree

The degree of a vertex is the number of edges incident to that vertex.

In directed graph

- In-degree of a vertex v to the number edges for which v is the head (How many edges point *into* this vertex.)
- Out-degree of a vertex v to the number edges for which v is the tail (How many edges start *from* this vertex.)



Classification

Type	Description	Example
Undirected Graph	Edges have no direction	Friendship network
Directed Graph (Digraph)	Edges have direction	Instagram “following”
Weighted Graph	Each edge has a cost	Google Maps distance
Unweighted Graph	All edges equal	Board game map
Cyclic Graph	Has loops	City ring road
Acyclic Graph	No loops	Family tree
Connected Graph	Every node reachable	Road network
Disconnected Graph	Some nodes isolated	Islands without bridge

Graph Representation

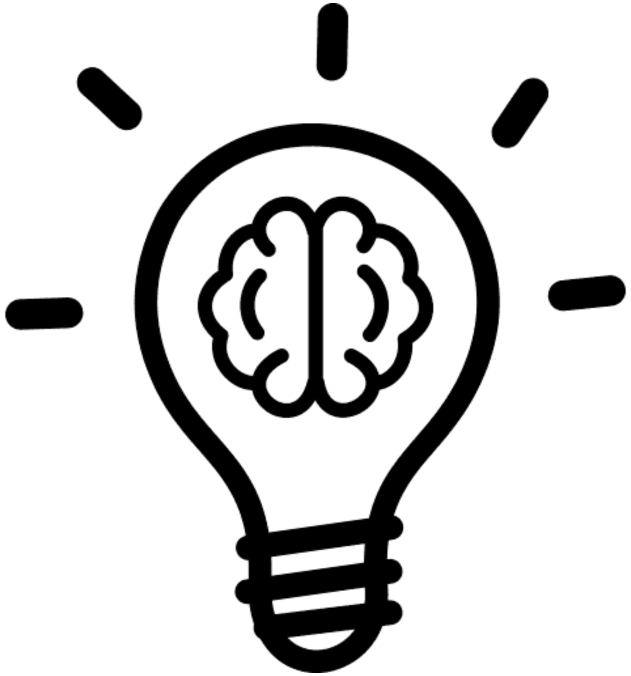


Image credit: https://uxwing.com/idea-icon/

Which data structure is commonly used to represent a graph?

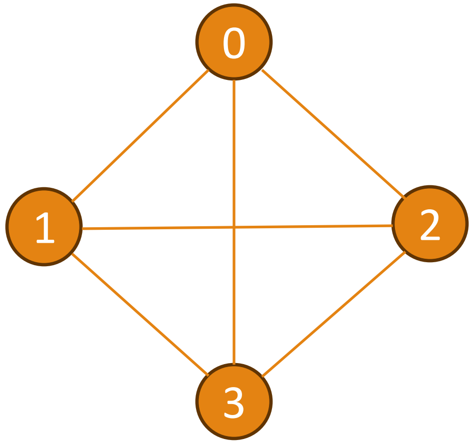
Which data structure allows for quick edge existence checks in a graph?

Graph Representation

Adjacency Matrix

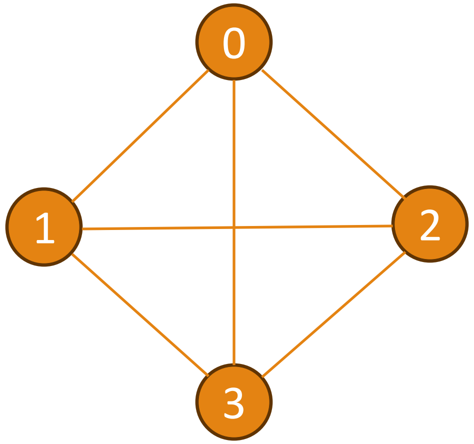
Adjacency List

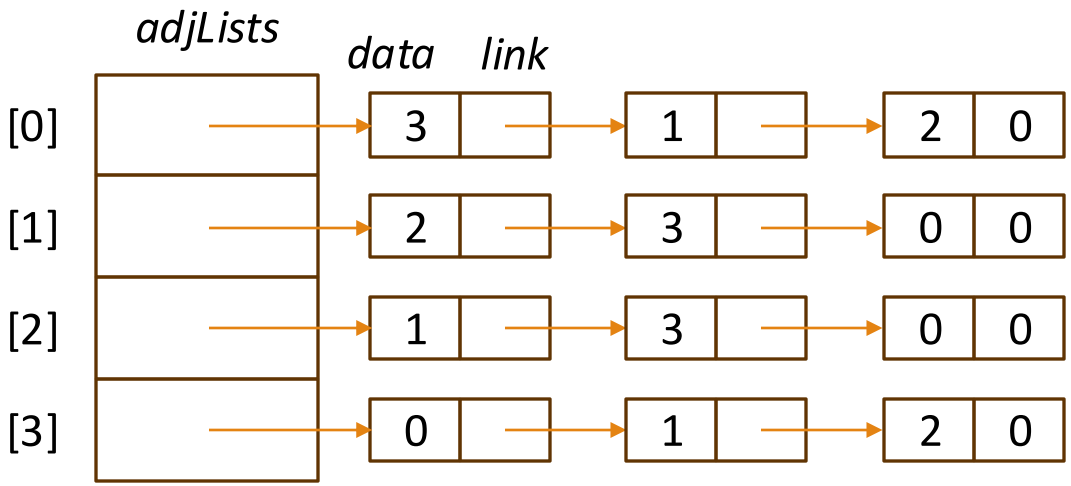
Adjacency Matrix



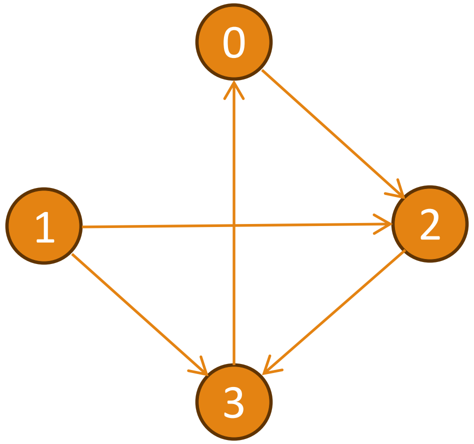
	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

Adjacency Lists



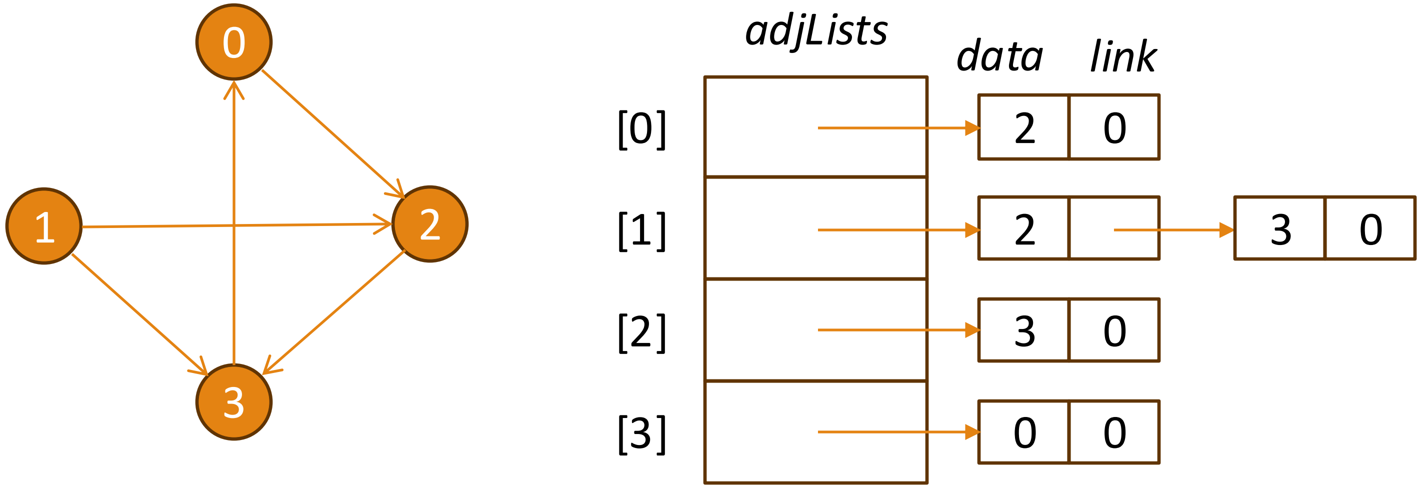


Adjacency Matrix



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

Adjacency Lists



Pros & Cons Study

Adjacency Matrix

Adjacency List

Pros & Cons Study: Adjacency Matrix

A $V \times V$ matrix that records whether an edge exists between two vertices.

Pros

- $O(1)$ edge lookup → matrix[u][v] is immediate
- Simple implementation → easy to code, easy to visualize
- Works well for dense graphs (many edges)
- Good for algorithms requiring fast access, e.g., Floyd–Warshall
- Natural fit for storing weights in weighted graphs

Cons

- $O(V^2)$ space, even if there are very few edges
- Wasteful for sparse graphs (most real-world graphs)
- Getting neighbors requires scanning the whole row → $O(V)$
- Harder to dynamically insert/remove vertices

Best for

- Dense graphs
- Graphs where fast edge lookup is important

Pros & Cons Study: Adjacency List

A list where each vertex stores only its neighbors.

Pros

- O(V + E) space → excellent for sparse graphs
- Fast traversal: neighbors of a vertex can be accessed in O(deg(v))
- Very efficient for BFS/DFS → O(V + E)
- Easy to scale to large graphs (millions of nodes)
- Insert/delete edges is O(1)

Cons

- Checking if edge (u, v) exists is O(deg(u))
- Slightly more complex implementation (nodes + pointers)
- Memory overhead if using many small linked-list nodes

Best for:

- Sparse graphs (most real-world graphs: social networks, maps)
- BFS/DFS, Dijkstra, Prim, Kruskal (all adjacency-list friendly)
- Large, dynamic graphs

Comparison: Sparse Matrix vs. Sparse List

Concept	Meaning	Efficient Representation
Sparse Matrix	Mostly zeros (few edges)	Adjacency List
Sparse List	List with few items	Ideal: list only stores existing edges

Feature	Adjacency Matrix	Adjacency List
Edge lookup	**O(1)**	**O(deg(v))**
Space	**$O(V^2)$**	**O(V + E)**
Traversal BFS/DFS	**$O(V^2)$**	**O(V + E)**
Best for	Dense graphs	Sparse graphs
Neighbor iteration	O(V)	O(deg(v))
Implementation	Simple	Moderate
Dynamic graph?	Hard	Easy

ADT: Graph

ADT *Graph* is

objects:

a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.

functions:

for all *graph* ∈ *Graph*, v, v_1, and v_2 ∈ *Vertices*

Graph Create()	::=	**return** an empty graph.
Graph InsertVertex(*graph*, v)	::=	**return** a graph with v inserted. v has no incident edge.
Graph InsertEdge(*graph*, v_1, v_2)	::=	**return** a graph with new edge between v_1 and v_2
Graph DeleteVertex(*graph*, v)	::=	**return** a graph in which v and all edges incident to it are removed
Graph DeleteEdge(*graph*, v_1, v_2)	::=	**return** a graph in which the edge (v_1, v_2) is removed
Boolean IsEmpty (*graph*)	::=	if (*graph* == empty graph) **return** *TRUE* **else return** *FALSE*
List Adjacent(*graph*, v)	::=	**return** a list of all vertices that are adjacent to v

end *Graph*

Definition: Graph

A **graph G(V, E)** consists of:

V = set of vertices (nodes)

E = set of edges connecting vertices

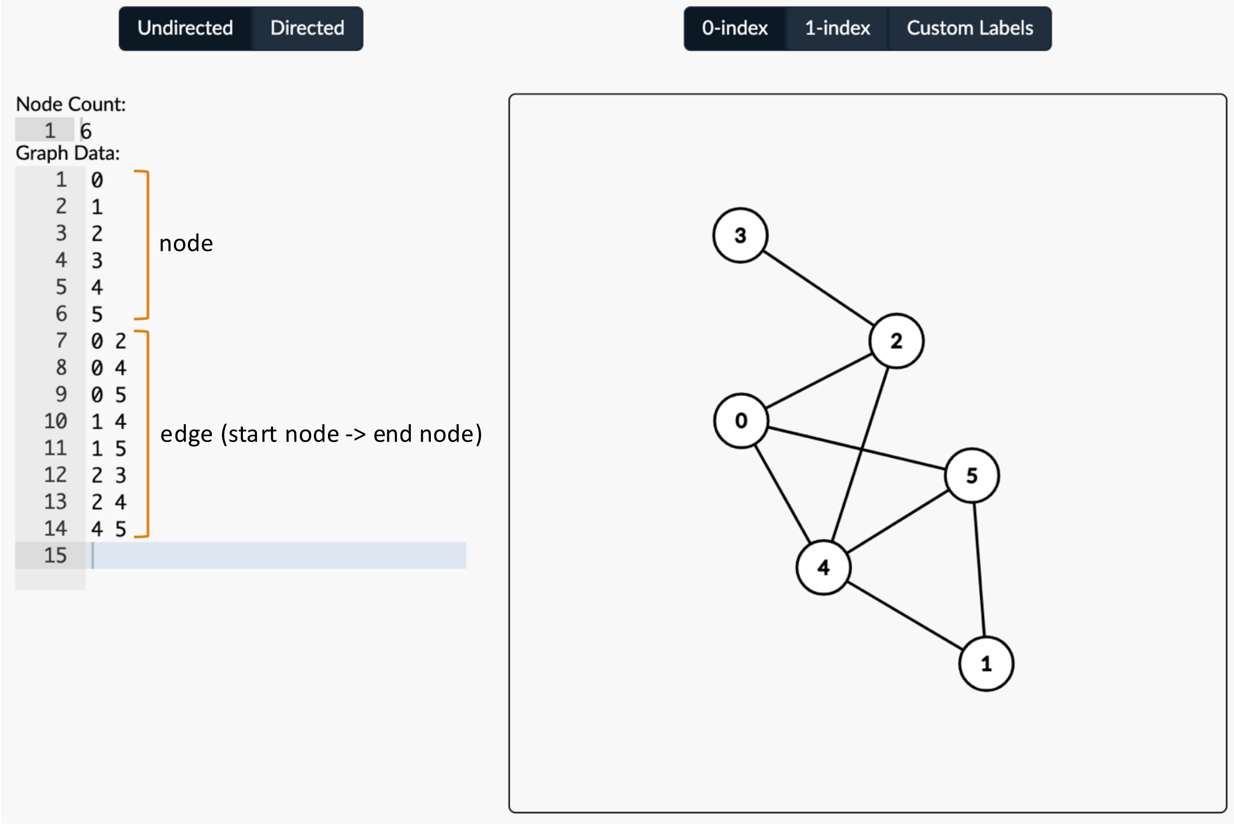


Image credit: https://csacademy.com/app/graph_editor/

Definition: Graph (cont.)

V = {0, 1, 2, 3, 4, 5}

E = {{0, 2}, {0, 4}, {0, 5},

{1, 4}, {1, 5},

{2, 3}, {2, 4},

{4, 5}}

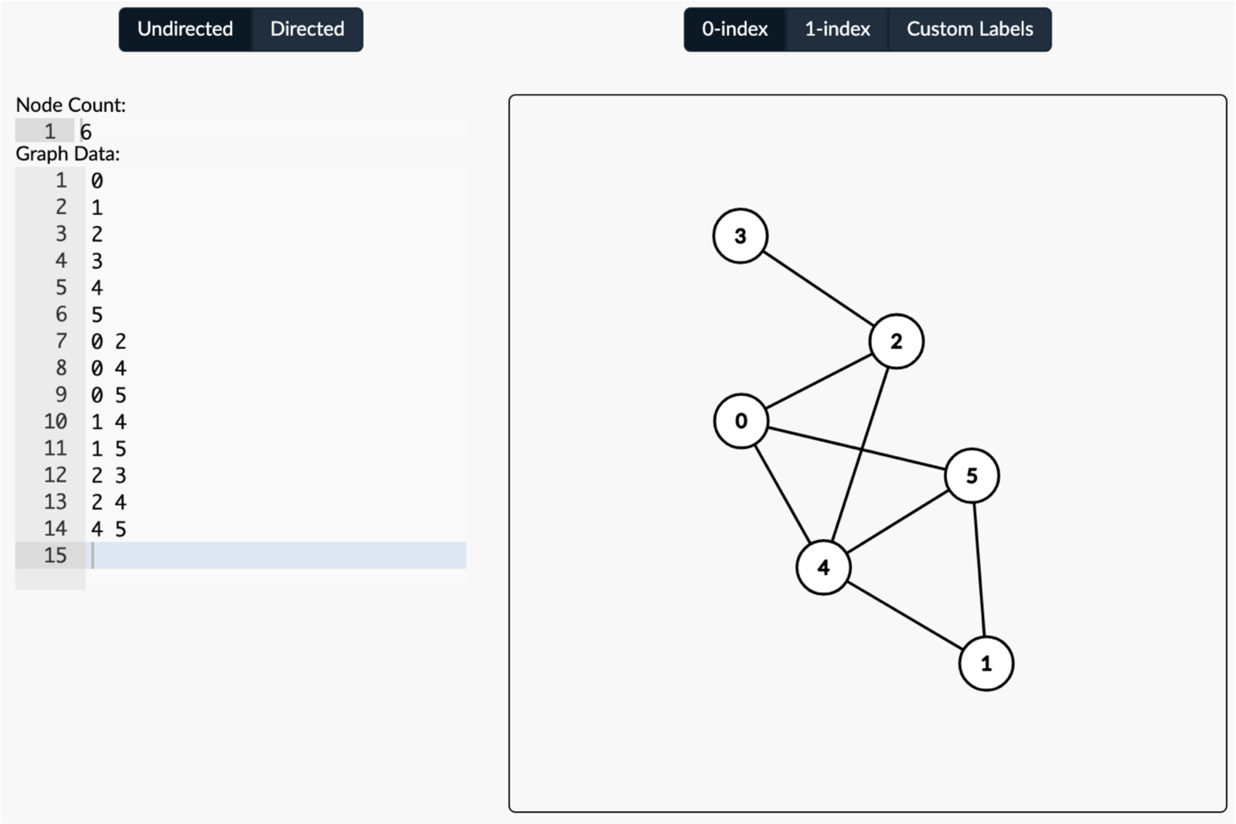
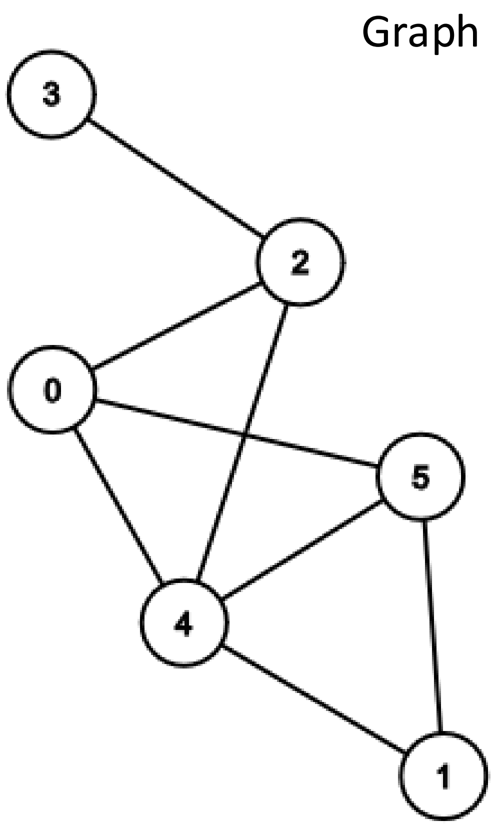
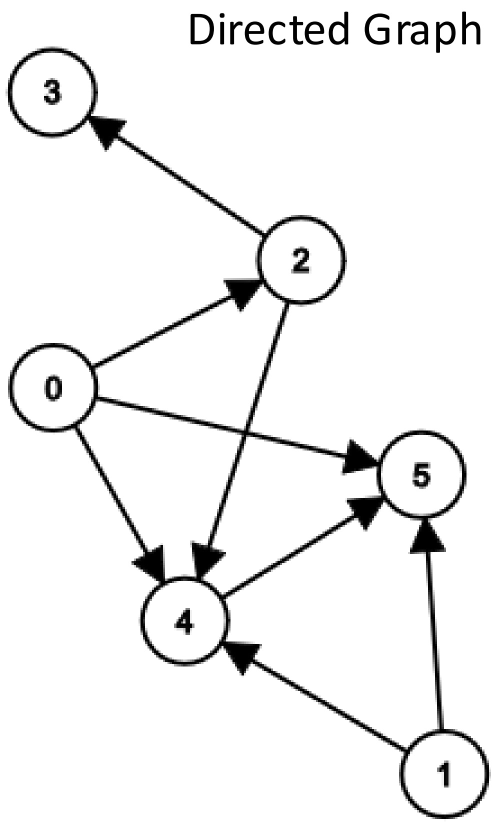


Image credit: https://csacademy.com/app/graph_editor/

Graph Traversal





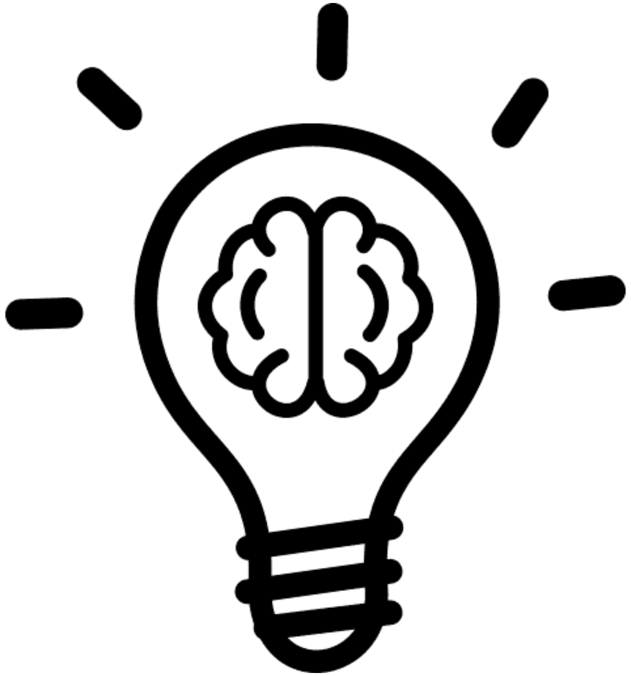


Image credit: https://uxwing.com/idea-icon/

Image credit: https://csacademy.com/app/graph_editor/

Graph Traversal / Graph Search

In the tree, we can traverse the tree by depth-first traversal and breadth-first traversal.

Graph vs. Tree

- Graph: general structure, can have cycles, any shape
- Tree: a connected acyclic graph

Similar

- A visited strategy
- A recursive depth-first approach (DFS)
- A queue-based breadth-first approach (BFS)
- Systematic exploration of nodes

Tree Traversal	Equivalent Graph Traversal
Preorder DFS	DFS
Level-order BFS	BFS

Graph vs. Tree

Property	Tree	Graph
Connectivity	Always connected	May be disconnected
Cycles	No	Yes
Direction	Not directed	Directed or undirected
Hierarchy	Yes (rooted)	No inherent hierarchy

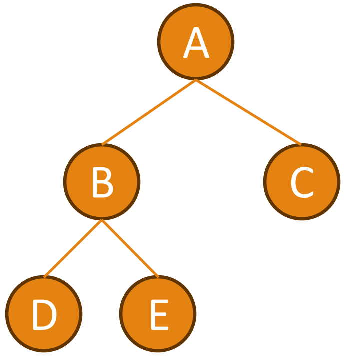
Traversal Algorithms

Traversal Type	Tree	Graph
DFS	Preorder, Inorder, Postorder	DFS (general)
BFS	Level-order	BFS (general)
Basis	Parent-child	Neighbor adjacency
Need visited[]	No	Yes

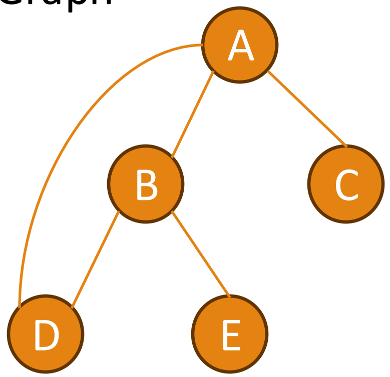
Graph traversal = Tree traversal **+ visited[]** to avoid cycles.

Graph vs. Tree

Tree



Graph



- Cycle
- Multiple paths
- visited[]

- DFS (Preorder): ABDEC
- BFS: ABCDE

- DFS (Preorder): ABDEC (one possible output, any node can be the starting node)
- BFS: ABCDE (one possible output, any node can be the starting node)

Graph: Breadth-first Search

BFS Algorithm

Create an empty queue and an empty visited set.

Enqueue the starting vertex.

While the queue is not empty:

- Dequeue a vertex v
- If v is not visited:
  - Mark v as visited and print v
  - Enqueue all unvisited neighbors of v

This is identical to tree level-order traversal, except:

Graphs may have cycles → must check visited

Graph Traversal Handling

Graph traversal must handle:

1. Cycles
2. Multiple entry paths
3. Arbitrary topology
4. Disconnected components
5. Directed edges (in-degree & out-degree)
6. Edge weights (for shortest-path problems)

Graph Traversal: Difference

Topic	Tree	Graph	What changes?
Cycles	❌ none	✅ yes	Must track visited nodes to prevent infinite loop
Parent–child relationship	✅ defined	❌ not defined	Graph traversal has no natural parent-child structure
Direction	❌ always undirected & acyclic	✅ directed or undirected	In-degree / out-degree matter
Disconnected components	❌ none	✅ possible	Need full traversal: run DFS/BFS for each component
Multiple paths between nodes	❌ only one path	✅ many paths possible	Graph traversal must choose and prune paths
Traversal order guarantee	✅ deterministic	❌ depends on adjacency representation	Order varies by adjacency list/set
Goal	hierarchical visit	exploration and connectivity	Graphs used for shortest path, search, cycle detection

Time Complexity

Operation	Adjacency Matrix	Adjacency List	Explanation
Check if edge (u, v) exists	O(1)	O(deg(u))	Matrix has direct index look-up
Get all neighbors of u	O(V)	O(deg(u))	Matrix scans entire row; list only stores neighbors
Add edge (u, v)	O(1)	O(1)	Both trivial
Remove edge (u, v)	O(1)	O(deg(u))	List must search to remove
Traversal (DFS/BFS)	$O(V^2)$	O(V + E)	Matrix scans row; list only walks actual edges
Space usage	$O(V^2)$	O(V + E)	Matrix dense, list sparse
Suitable for	Dense graphs	Sparse graphs	

References

VisuAlgo (DFS/BFS)
https://visualgo.net/en/dfsbfs

Graph Online (draw graphs)
https://graphonline.ru/en/

Graph Editor

https://csacademy.com/app/graph_editor/

Do not edit
How to change the design

Any Suggestion?

The Slido app must be installed on every computer you're presenting from

slido