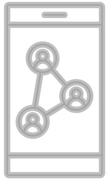
METROPOLITAN TIRANA UNIVERSITY

Course: Data Structures and Algorithms

Graphs



Evis Plaku

Why graphs?

Because everything is a graph

Foundations of graph theory

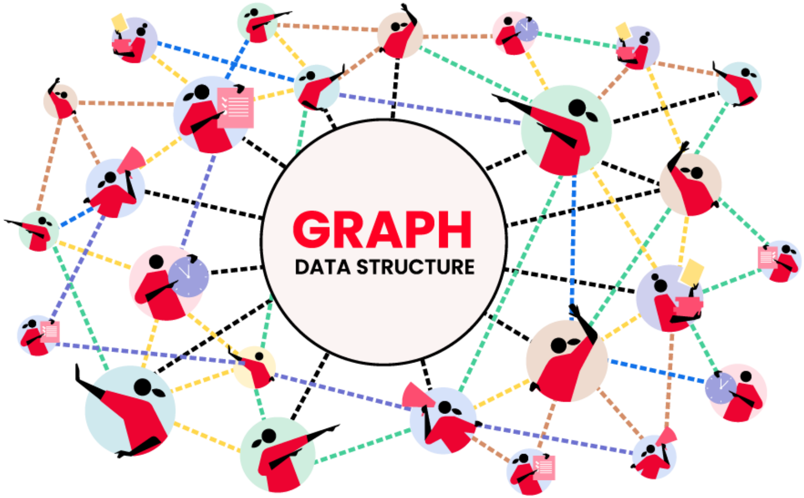
Introduction to graphs

Graphs model relationships between entities using nodes (vertices) and connections (edges)

- Nodes represent objects or entities

- Edges represent relationships between nodes

- Widely used in real-world applications



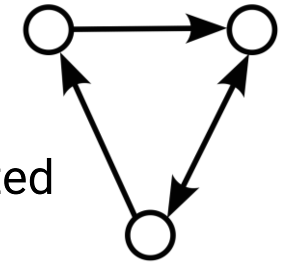
Introduction to graphs (more formal)

METROPOLITAN TIRANA UNIVERSITY

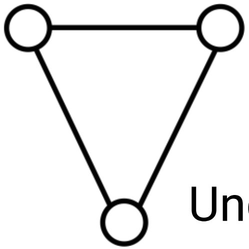
A graph G is an ordered pair (V, E)

- V is a non-empty set of **vertices** (or **nodes**)
- $E \subseteq \{(u, v) \mid u, v \in V\}$ is a set of **edges** connecting pairs of vertices

- In a **directed graph (digraph)**, edges are ordered pairs
- In an **undirected graph**, edges are unordered pairs



Directed

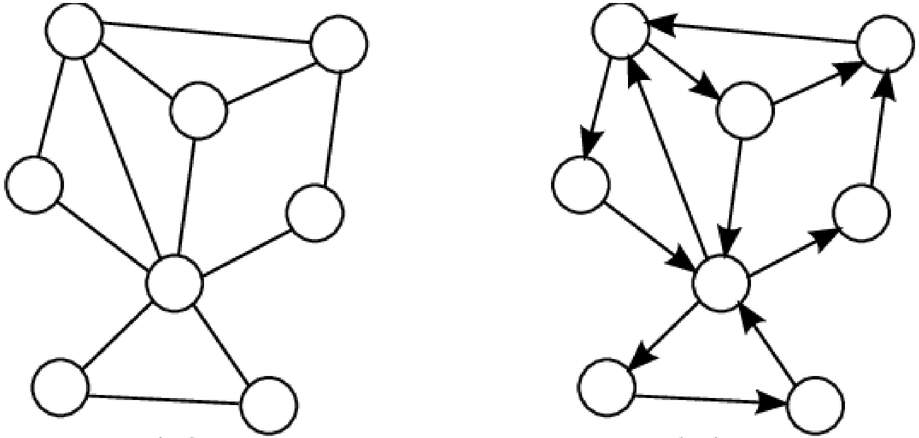


Undirected

Types of graphs

Graphs vary based on edge direction and whether edges carry weights or not

Undirected: edges go both ways



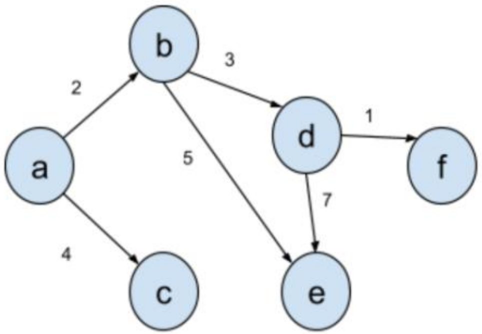
(a) (b)

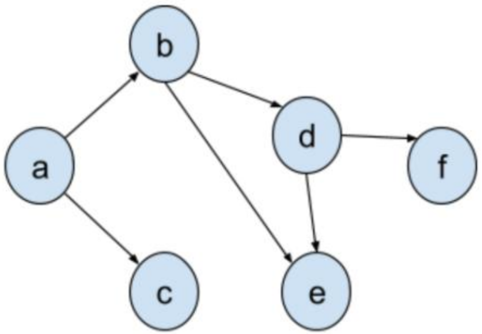
Directed: edges have a direction

Types of graphs

Graphs vary based on edge direction and whether edges carry weights or not

Weighted: edges have numeric values



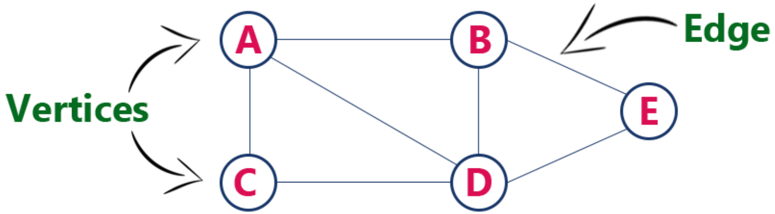


Unweighted: all edges are equal

Graph terminology

Understanding graph terms helps describe structure, movement, and connectivity within graph models

- **Node:** an element $v \in V$ representing a distinct entity within the graph

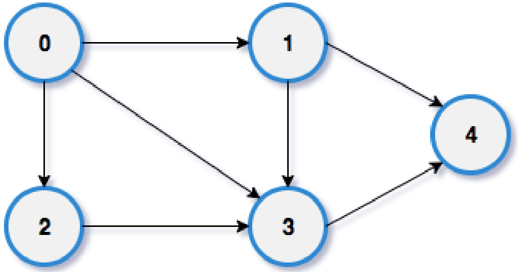


- **Edge:** an ordered pair (u, v) in directed graphs or an unordered pair $\{u, v\}$ in undirected graphs, where $u, v \in V$ represent a connection between two vertices

Graph terminology

Understanding graph terms helps describe structure, movement, and connectivity within graph models

- **Degree:** the number of edges incident to a vertex $v \in V$
  - **In-degree**: number of incoming edges (directed graphs)
  - **Out-degree**: number of outgoing edges (directed graphs)
  - **Total degree**: sum of in-degree and out-degree

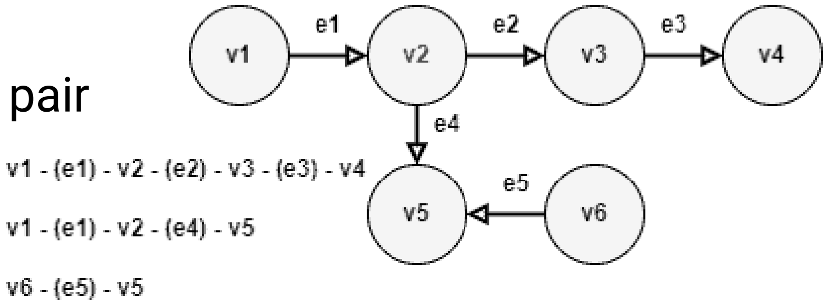


- Node 3 in-degree is 3
- Node 3 out-degree is 1
- Node 3 total degree is 4

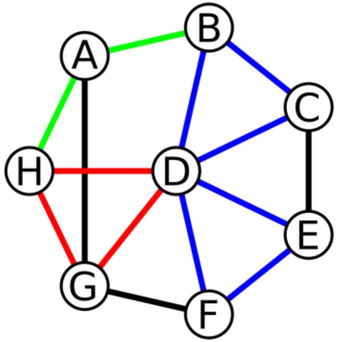
Graph terminology

Understanding graph terms helps describe structure, movement, and connectivity within graph models

- **Path:** a finite sequence of vertices $(v_1, v_2, \dots, v_k)$ such that each consecutive pair (v_i, v_{i+1}) is connected by an edge



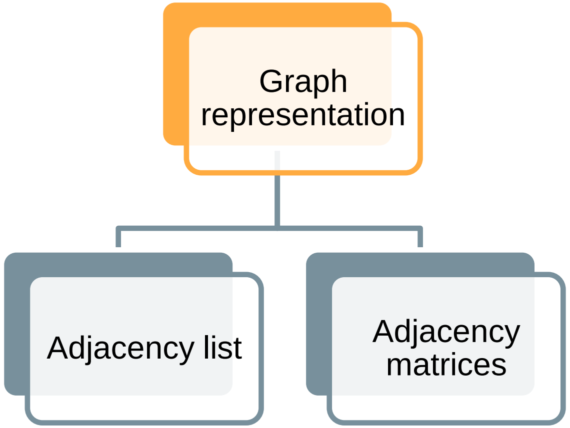
- **Cycle:** a cycle is a path where the first and last vertices are the same, and no other vertices are repeated.



Graph representation overview

Choosing the right graph representation improves storage efficiency and speeds up operations

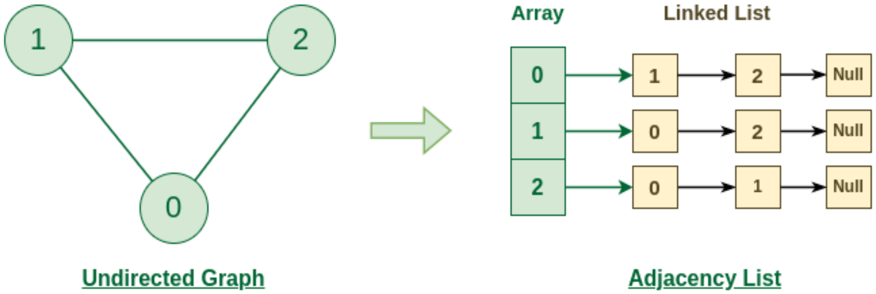
- Store and manipulate graph data effectively
- Impacts memory use and algorithm performance
- Different representations suit different graph types



Adjacency list representation

Adjacency lists store each node’s neighbors efficiently, saving space for sparse graphs

- List each vertex with its adjacent vertices
- Good for graphs with few edges
- Easy to add or remove edge



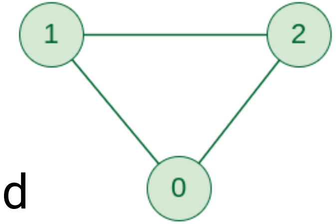
Representation: use an array of lists

indices represent vertices; values store list of adjacent vertices

Adjacency matrices representation

Adjacency matrices use a 2D array to directly record all edges between vertices

- Each entry: 1 if edge exists, 0 if not
- Good when most vertices are connected
- Direct access to any edge between vertices



Undirected Graph

	0	1	2
0		1	1
1	1		1
2	1	1	

Adjacency Matrix

Representation: 2D array (matrix) where rows and columns represent vertices

Graph representation comparison

Choosing between adjacency list and matrix depends on graph density and algorithm needs

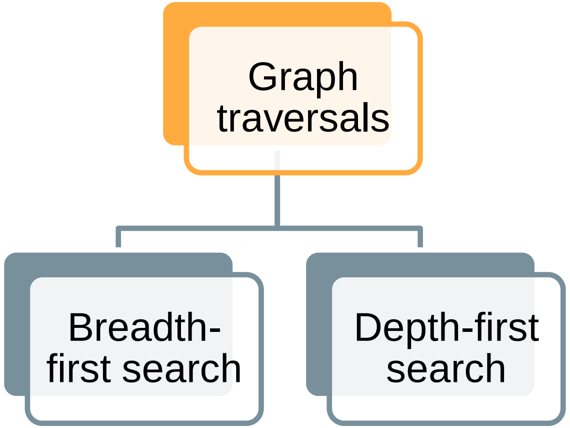
Aspect	Adjacency List	Adjacency Matrix
Best for	Sparse graphs (few edges)	Dense graphs (many edges)
Space complexity	$O(V + E)$	$O(V^2)$
Edge lookup	$O(V)$ for traversal, $O(E)$ for search	$O(1)$ (constant time for any edge check)
Ease of Use	More flexible for varying edge counts	Simple structure but not space-efficient

Graph traversals overview

Graph traversals

Traversal: process of visiting all vertices and edges in a graph

- Helps in searching for specific nodes or paths
- Essential for pathfinding algorithms (e.g., shortest path)
- Critical in problems like network analysis or web crawling



Graph traversal: Depth-First Search (DFS)

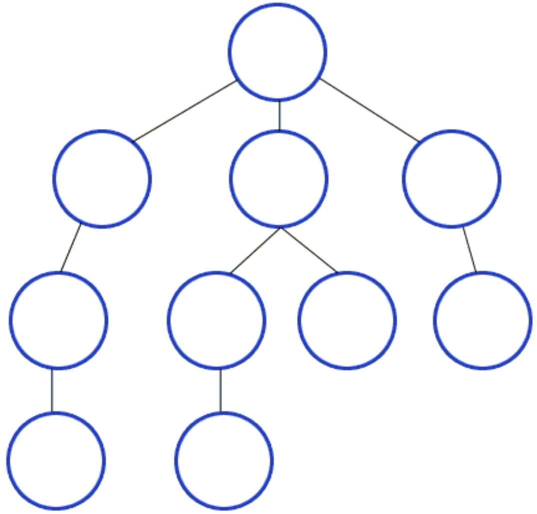
METROPOLITAN TIRANA UNIVERSITY

DFS explores a graph by going deeper into each branch before backtracking to previous nodes.

- Explores as far down a branch as possible before backtracking

- Uses a stack (recursive or iterative) for node exploration

- Marks nodes as visited to avoid revisiting



Graph traversal: Depth-First Search (DFS)

METROPOLITAN TIRANA UNIVERSITY

DFS explores a graph by going deeper into each branch before backtracking to previous nodes.

Recursive

```
DFS(vertex):
    mark vertex as visited
    for each neighbor of vertex:
        if neighbor is not visited:
            DFS(neighbor)
```

Choose recursive DFS for simplicity, iterative DFS for handling large graphs and avoiding recursion limits

Iterative

```
DFS-Iterative(start_vertex):
    create a stack
    push start_vertex onto stack
    while stack is not empty:
        vertex = pop from stack
        if vertex is not visited:
            mark vertex as visited
        for each neighbor of vertex:
            if neighbor is not visited:
                push neighbor onto stack
```

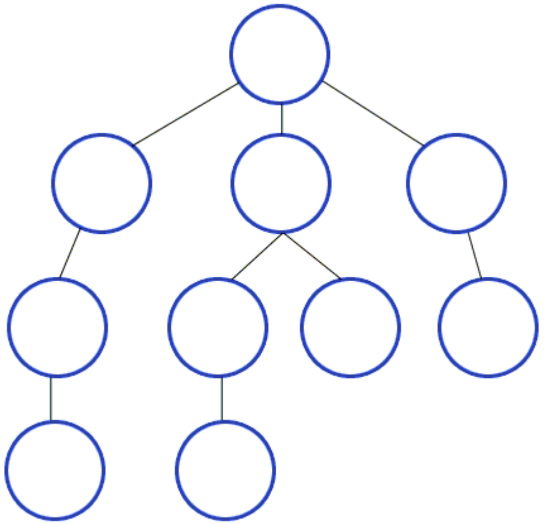
Graph traversal: Breadth-First Search (BFS)

BFS explores neighbors level by level
ensuring all nodes at each depth are visited first

- Explores all neighbors at the current level before moving to the next level

- Uses a queue for managing nodes in the exploration process

- Guarantees shortest path in unweighted graphs



Graph traversal: Breadth-First Search (BFS)

BFS explores neighbors level by level
ensuring all nodes at each depth are visited first

- Uses a queue to explore nodes level by level
- Mark nodes as visited to avoid revisiting
- Processes nodes in the order they are discovered

```
BFS(start_vertex):
    create a queue
    enqueue start_vertex onto queue
    mark start_vertex as visited
    while queue is not empty:
        vertex = dequeue from queue
        process vertex
        for each neighbor of vertex:
            if neighbor is not visited:
                enqueue neighbor onto queue
                mark neighbor as visited
```

DFS vs BFS

DFS explores deeply along a branch,
while BFS explores all neighbors level by level

Use Case	DFS	BFS
Shortest Path	Not ideal for unweighted graphs	Ideal for unweighted graphs
Cycle Detection	Effective for cycle detection in directed/undirected graphs	Can also detect cycles, but DFS is preferred
Pathfinding	Works for pathfinding in **complex** graphs, **deep** solutions	Best for finding the **shortest** path in unweighted graphs
Maze or Puzzle Solving	Suitable for problems requiring exhaustive search (e.g., N-Queens, Sudoku)	Better for shortest solution (e.g., shortest maze path)

Graph applications

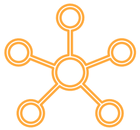
Graph applications

METROPOLITAN TIRANA UNIVERSITY

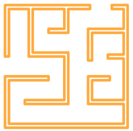
Social networks

Routing and navigation

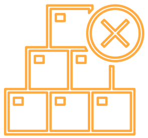
Computer networks



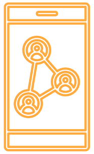
Puzzle Solving



Supply Chains



Recommendation systems

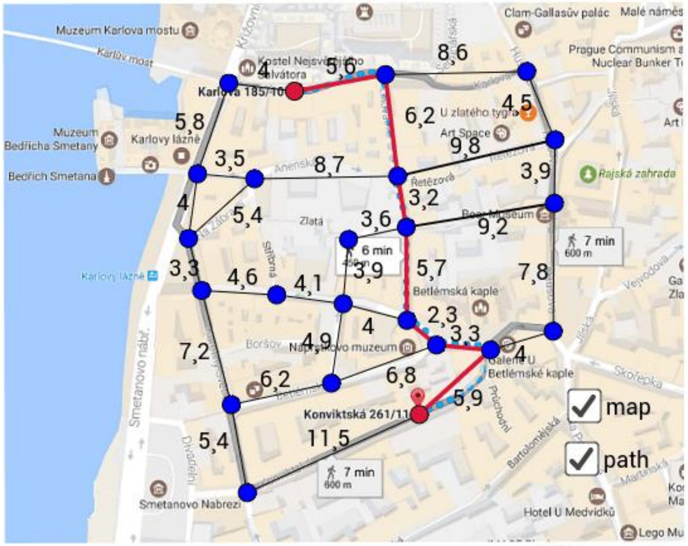


Graph applications: routing and navigation

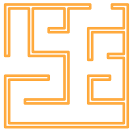
METROPOLITAN TIRANA UNIVERSITY

Navigating through cities and road networks is complex due to traffic, congestion, and inefficient routing

- Graphs model cities and roads
- Nodes: Locations, intersections
- Edges: Roads, paths with weights (distance, time, or cost)
- Shortest path: Use **Dijkstra**'s or **A*** to find the quickest route between two points



Graph applications: games and mazes

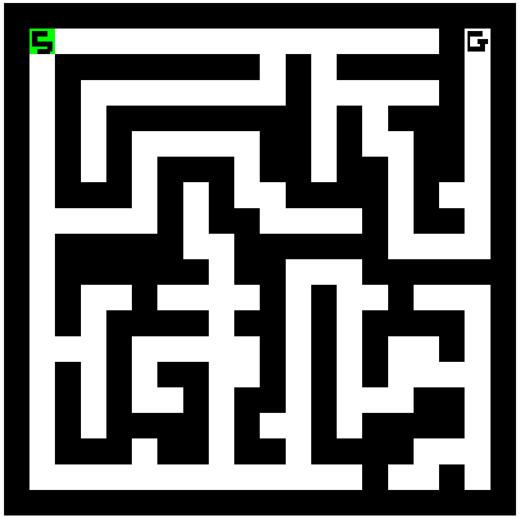


Game AI needs to make strategic decisions in complex environments, and solving puzzles like mazes requires efficient pathfinding

- Graphs model game environments or mazes as interconnected nodes (positions) and edges (paths)

- Nodes: positions in the game or maze

- Edges: possible moves, paths between nodes



Use **BFS** for the shortest path and **DFS** for exhaustive search through mazes

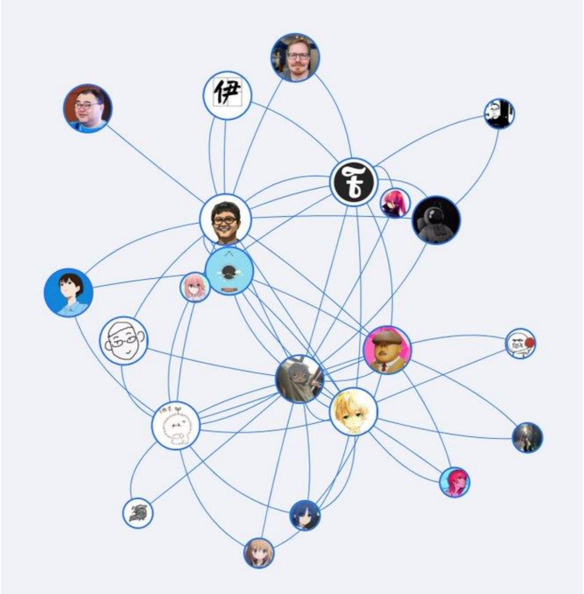
Graph applications: social networks

Game AI needs to make strategic decisions in complex environments, and solving puzzles like mazes requires efficient pathfinding

- Graphs represent users as nodes and their relationships (friendships, followers, etc.) as edges

- Nodes: users in the network

- Edges: connections (friendships, followers, likes)



Use **collaborative filtering** to suggest friends or content based on graph connectivity

Graph applications: what's next

- Explore Google's Knowledge Graph and how it connects information
- Investigate Facebook's Social Graph for friend suggestions and communities
- Analyze how Google Maps finds the fastest route using graphs
- Understand how Netflix suggests shows using user-item graphs
- Learn how chatbots use knowledge graphs to link ideas
- Study how web crawlers model the internet as a graph
- Investigate how supply chains are optimized with graph models

Quote of the Week

In graphs, it's all about connections

just like in networking and life