Graphs

Modeling Relationships and Connectivity

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction to Graphs

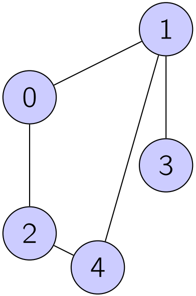
What is a Graph?

Definition: A collection of **nodes (vertices)** connected by **edges**.

Formal notation: G = (V, E)

- V = Set of vertices
- E = Set of edges (pairs of vertices)

Example Graph:



Components:

- V = {0, 1, 2, 3, 4}
- E = {(0,1), (0,2), (1,3), (1,4), (2,4)}
- |V| = 5 vertices
- |E| = 5 edges

Why Use Graphs?

Graphs model relationships between entities:

- **Social networks:** People connected by friendships
- **Maps:** Cities connected by roads
- **Internet:** Websites connected by hyperlinks
- **Dependencies:** Tasks connected by prerequisites
- **Molecules:** Atoms connected by bonds
- **Recommendations:** Users/items connected by preferences

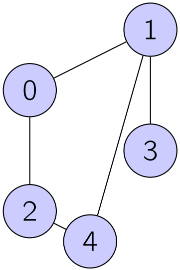
Fundamental questions:

- Is there a path from A to B?
- What's the shortest path?
- Are all nodes connected?
- Does the graph contain cycles?

Graph Representations

Adjacency List Representation

Idea: Store a list of neighbors for each vertex.



Adjacency List:

- 0: [1, 2]
- 1: [0, 3, 4]
- 2: [0, 4]
- 3: [1]
- 4: [1, 2]

Properties:

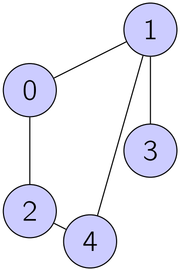
- **Space:** $O(V + E)$
- **Add edge:** $O(1)$
- **Check edge:** $O(\text{degree})$
- **Best for:** Sparse graphs ($E \ll V^2$)

Adjacency List Implementation

```python
# Using dictionary
graph = {
    0: [1, 2],
    1: [0, 3, 4],
    2: [0, 4],
    3: [1],
    4: [1, 2]
}

# Using list of lists
graph = [
    [1, 2],       # neighbors of vertex 0
    [0, 3, 4],    # neighbors of vertex 1
    [0, 4],       # neighbors of vertex 2
    [1],          # neighbors of vertex 3
    [1, 2]        # neighbors of vertex 4
]
```

Adjacency Matrix Representation

Idea: 2D array where matrix[i][j] = 1 if edge (i,j) exists.



Adjacency Matrix:

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

Properties:

- **Space:** $O(V^2)$
- **Add/check edge:** $O(1)$
- **Get neighbors:** $O(V)$
- **Best for:** Dense graphs ($E \approx V^2$)

Adjacency Matrix Implementation

```
# Unweighted graph (0 = no edge, 1 = edge)
n = 5
matrix = [
    [0, 1, 1, 0, 0],
    [1, 0, 0, 1, 1],
    [1, 0, 0, 0, 1],
    [0, 1, 0, 0, 0],
    [0, 1, 1, 0, 0]
]

# Weighted graph (0 or inf = no edge, value = weight)
inf = float('inf')
matrix = [
    [0,   5,   3,   inf, inf],
    [5,   0,   inf, 2,   1],
    [3,   inf, 0,   inf, 4],
    [inf, 2,   inf, 0,   inf],
```

Comparison: List vs Matrix

Operation	Adjacency List	Adjacency Matrix
Space	$O(V + E)$	$O(V^2)$
Add edge	$O(1)$	$O(1)$
Remove edge	$O(\text{degree})$	$O(1)$
Check edge	$O(\text{degree})$	$O(1)$
Get neighbors	$O(\text{degree})$	$O(V)$
Iterate all edges	$O(V + E)$	$O(V^2)$
Best for	Sparse graphs	Dense graphs

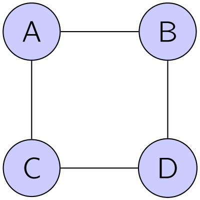
When to use:

- **Adjacency List:** Most real-world graphs (social, web, roads)
- **Adjacency Matrix:** Dense graphs, need fast edge queries, matrix algorithms

Types of Graphs

Directed vs Undirected Graphs

Undirected Graph:
Edges are bidirectional

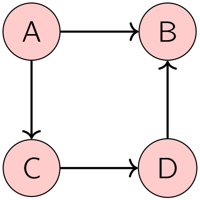


Examples:

- Friendships
- Two-way roads
- Collaborations

Edge (A,B) means:

Directed Graph:
Edges have direction



Examples:

- Twitter follows
- Web links
- Dependencies

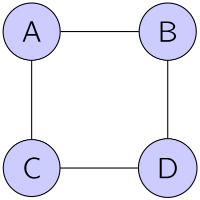
Edge A→B means:

Directed Graph Implementation

```
# Undirected: add edge in both directions
def add_undirected_edge(graph, u, v):
    graph[u].append(v)
    graph[v].append(u)

# Example: friendship network
friends = {
    'Alice': ['Bob', 'Charlie'],
    'Bob': ['Alice', 'David'],
    'Charlie': ['Alice', 'David'],
    'David': ['Bob', 'Charlie']
}

# Directed: add edge in one direction only
def add_directed_edge(graph, u, v):
    graph[u].append(v)
```

Weighted vs Unweighted Graphs

Unweighted Graph:
All edges have equal cost

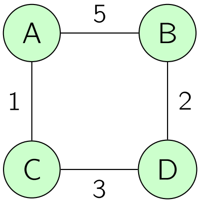


Use cases:

- Social networks
- Maze solving
- Connectivity

Shortest path:

Weighted Graph:
Edges have costs/weights



Use cases:

- Road networks
- Flight routes
- Cost optimization

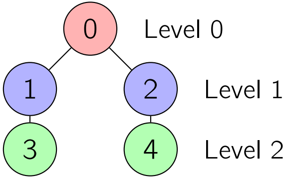
Shortest path:

Graph Traversals

Breadth-First Search (BFS)

Strategy: Explore level by level (nearest neighbors first)

Data Structure: Queue (FIFO)



BFS from 0: [0, 1, 2, 3, 4]

Algorithm:

1. Start at source
2. Add to queue
3. While queue not empty:
   - Dequeue vertex
   - Visit it
   - Enqueue unvisited neighbors

Applications:

- Shortest path (unweighted)
- Level-order traversal
- Connected components

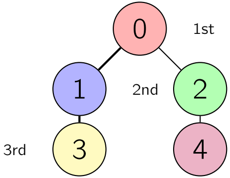
BFS Implementation

```python
from collections import deque

def bfs(graph, start):
    visited = set([start])
    queue = deque([start])
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return result
```

Depth-First Search (DFS)

Strategy: Explore as deep as possible before backtracking

Data Structure: Stack (or recursion)



DFS from 0: [0, 1, 3, 2, 4]

Algorithm:

1. Start at source
2. Mark as visited
3. For each unvisited neighbor:
   - Recursively DFS from it
4. Backtrack

Applications:

- Cycle detection
- Topological sorting
- Strongly connected components

DFS Implementation

```
# Recursive
def dfs_recursive(graph, node, visited=None):
    if visited is None:
        visited = set()

    visited.add(node)
    result = [node]

    for neighbor in graph[node]:
        if neighbor not in visited:
            result.extend(dfs_recursive(graph, neighbor, visited))

    return result

# Iterative
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]
    result = []
```

BFS vs DFS Comparison

Feature	BFS	DFS
Data structure	Queue	Stack/Recursion
Order	Level by level	Deep first
Shortest path	Yes (unweighted)	No
Memory	More (stores level)	Less (stores path)
Completeness	Yes	Yes (with visited)
Time	O(V + E)	O(V + E)
Space	O(V)	O(V)

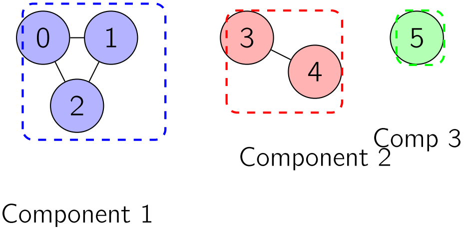
When to use:

- **BFS:** Shortest path, level processing, closer nodes first
- **DFS:** Cycle detection, topological sort, exploring all paths

Connected Components and Cycles

Connected Components

Definition: Maximal subgraphs where every pair of vertices is connected.



Example: 3 connected components: {0,1,2}, {3,4}, {5}

Algorithm: Run BFS/DFS from each unvisited vertex

- Each DFS/BFS finds one component
- Count number of DFS/BFS calls needed

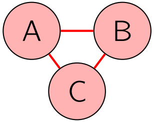
Finding Connected Components

```
def count_components(graph):
    visited = set()
    count = 0

    def dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)

    for node in graph:
        if node not in visited:
            dfs(node)
            count += 1

    return count

def find_components(graph):
    visited = set()
```

Cycle Detection

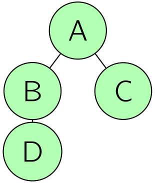
Cycle: A path that starts and ends at the same vertex.

Has Cycle:



Cycle: A → B → C → A

No Cycle (Tree):



Tree: No cycles

Detection Methods:

- **Undirected:** DFS with parent tracking
- **Directed:** DFS with color marking (White/Gray/Black)

Cycle Detection - Undirected

```
def has_cycle_undirected(graph):
    visited = set()

    def dfs(node, parent):
        visited.add(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                return True  # Back edge found (cycle)

        return False

    for node in graph:
        if node not in visited:
```

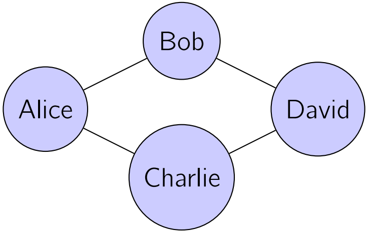
Cycle Detection - Directed

```
def has_cycle_directed(graph):
    WHITE, GRAY, BLACK = 0, 1, 2
    color = {node: WHITE for node in graph}

    def dfs(node):
        color[node] = GRAY  # Currently exploring

        for neighbor in graph[node]:
            if color[neighbor] == GRAY:
                return True  # Back edge (cycle)
            if color[neighbor] == WHITE and dfs(neighbor):
                return True

        color[node] = BLACK  # Finished exploring
        return False

    for node in graph:
        if color[node] == WHITE:
            if dfs(node):
```

Real-World Applications

Social Networks

Model: Users as vertices, relationships as edges



Applications:

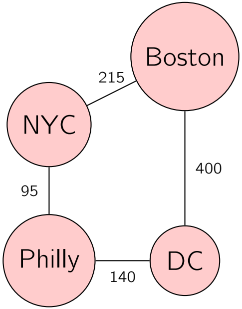
- **Friend recommendations:** Friends of friends (2-hop neighbors)
- **Influence analysis:** Find most connected users (degree centrality)
- **Community detection:** Find clusters/groups
- **Six degrees of separation:** Shortest path between users
- **Viral spread:** Model information propagation

Friend Recommendations

```python
def recommend_friends(graph, user):
    """Find friends of friends"""
    friends = set(graph[user])
    recommendations = set()

    for friend in friends:
        for friend_of_friend in graph[friend]:
            if friend_of_friend != user and \
               friend_of_friend not in friends:
                recommendations.add(friend_of_friend)

    return recommendations

def find_influencers(graph, top_k=10):
    """Find most connected users (degree centrality)"""
    degrees = [(node, len(graph[node])) for node in graph]
    degrees.sort(key=lambda x: x[1], reverse=True)
```

Maps and Navigation

Model: Locations as vertices, roads as weighted edges

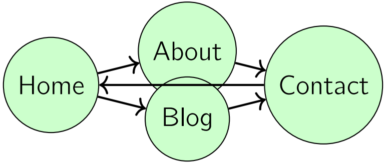


Applications:

- **Route planning:** Shortest path (Dijkstra's, A*)
- **Traffic optimization:** Alternative routes, congestion avoidance
- **Delivery routing:** Traveling salesman problem (TSP)
- **Public transit:** Multi-modal routing (bus + train + walk)

Web and Internet

Model: Pages/routers as vertices, links/connections as edges



Applications:

- **PageRank:** Rank web pages by importance/authority
- **Web crawling:** BFS/DFS to discover new pages
- **Network routing:** Find optimal packet paths
- **Load balancing:** Distribute traffic across servers
- **Link analysis:** Detect spam, find related pages

Other Applications

Dependency Graphs:

- Build systems: Compile order (topological sort)
- Package managers: Install dependencies
- Task scheduling: Prerequisite handling

Recommendation Systems:

- User-item bipartite graphs
- Collaborative filtering
- Content-based recommendations

Science and Research:

- Biology: Protein interaction networks, phylogenetic trees
- Chemistry: Molecular structures, reaction networks
- Physics: Particle interactions
- Social science: Citation networks, collaboration graphs

Games and AI:

Complexity Analysis

Time Complexity Summary

Operation	Adjacency List	Adjacency Matrix
Add vertex	O(1)	$O(V^2)$ (resize)
Add edge	O(1)	O(1)
Remove vertex	O(E)	$O(V^2)$
Remove edge	O(E)	O(1)
Check edge	O(degree)	O(1)
Get neighbors	O(degree)	O(V)
BFS/DFS	O(V + E)	$O(V^2)$
Dijkstra	O((V+E) log V)	$O(V^2)$

Key insight:

- Adjacency list better for sparse graphs ($E \ll V^2$)
- Adjacency matrix better for dense graphs ($E \approx V^2$)

Space Complexity

Representation	Space	Best For
Adjacency List	$O(V + E)$	Sparse graphs
Adjacency Matrix	$O(V^2)$	Dense graphs
Edge List	$O(E)$	Simple storage

Graph Density:

- **Sparse:** $E = O(V)$, few edges
  - Example: Social networks (avg degree $\sim$100)
  - Use adjacency list
- **Dense:** $E = O(V^2)$, many edges
  - Example: Complete graph (all pairs connected)
  - Use adjacency matrix

Memory Calculations

Example: 1 million vertices

Sparse graph (avg degree = 10):

- E ≈ 5 million edges
- Adjacency list: (V + E) × 8 bytes = 40 MB
- Adjacency matrix: V × V × 1 byte = 1 TB (impractical!)

Dense graph ($E = V^2/2$):

- E ≈ 500 billion edges
- Adjacency list: (V + E) × 8 bytes = 4 TB
- Adjacency matrix: V × V × 1 byte = 1 TB (better!)

Practical considerations:

- Small graphs ($V < 1000$): Either works
- Medium graphs ($V < 100K$): Adjacency list usually better
- Large graphs ($V > 1M$): Must use adjacency list
- Very dense: Consider compressed formats

Summary

Key Concepts Recap

Graph Fundamentals:

- Graph G = (V, E): vertices and edges
- Models relationships between entities
- Directed vs undirected, weighted vs unweighted

Representations:

- **Adjacency list:** O(V + E) space, best for sparse
- **Adjacency matrix:** $O(V^2)$ space, best for dense

Traversals:

- **BFS:** Level by level, shortest path (unweighted)
- **DFS:** Deep first, cycle detection, topological sort

Analysis:

- Connected components: Find separate subgraphs
- Cycle detection: Identify circular dependencies
- Both use BFS/DFS: O(V + E)

Applications Recap

Major Use Cases:

- **Social networks:** Friend recommendations, influence analysis
- **Maps/navigation:** Route planning, traffic optimization
- **Web:** PageRank, web crawling, network routing
- **Dependencies:** Build systems, package managers
- **Science:** Biology, chemistry, physics networks

Common Algorithms:

- BFS, DFS: O(V + E)
- Dijkstra's shortest path: O((V+E) log V)
- Topological sort: O(V + E)
- Connected components: O(V + E)
- Cycle detection: O(V + E)

Practice Problems

Basic:

- Implement adjacency list and matrix representations
- Implement BFS and DFS
- Count connected components
- Detect cycles in undirected graph

Intermediate:

- Find shortest path in unweighted graph (BFS)
- Check if graph is bipartite
- Find all paths from source to destination
- Detect cycles in directed graph
- Clone a graph

Advanced:

- Implement Dijkstra's algorithm
- Topological sort (course schedule problems)
- Strongly connected components (Kosaraju's/Tarjan's)

Implementation Tips

Best Practices:

- Use adjacency list for most problems
- Always track visited nodes in BFS/DFS
- Handle disconnected graphs (multiple components)
- Consider edge cases: empty graph, single vertex, cycles

Common Pitfalls:

- Forgetting to mark nodes as visited (infinite loops)
- Not handling directed vs undirected correctly
- Using wrong algorithm (BFS for shortest weighted path)
- Memory issues with large dense graphs

Optimization:

- Use sets for O(1) visited checks
- Use deque for BFS (efficient popleft)
- Consider Union-Find for connectivity problems
- Profile before optimizing representation

Further Learning

Advanced Topics:

- **Shortest paths:** Dijkstra's, Bellman-Ford, Floyd-Warshall, A*
- **Minimum spanning trees:** Prim's, Kruskal's
- **Network flow:** Ford-Fulkerson, max flow min cut
- **Strongly connected components:** Kosaraju's, Tarjan's
- **Graph coloring:** Chromatic number, map coloring

Resources:

- Practice on LeetCode graph problems
- Study graph algorithms in CLRS textbook
- Visualize with tools: Graphviz, NetworkX
- Build real projects: social network, route planner

Projects:

- Implement social network with friend recommendations
- Build shortest path finder for maps
- Create web crawler using BFS

Thank You!

Questions?

Graphs: Connecting the World Through Data