Graph Algorithms

Compute Connectivity, Paths, and Structures Over Graphs

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction

What are Graph Algorithms?

Graph Algorithms: Methods to solve problems on graph structures

Key Problems:

- **Traversal**: Visit all vertices/edges
- **Connectivity**: Determine if vertices are connected
- **Shortest Paths**: Find minimum cost paths
- **Spanning Trees**: Connect all vertices with minimum cost
- **Flow**: Maximize throughput in networks

Real-World Applications:

- Social networks (connections, recommendations)
- Navigation systems (GPS, routing)
- Computer networks (packet routing, topology)
- Task scheduling (dependencies, ordering)
- Bioinformatics (protein interactions, gene networks)

Graph Basics Recap

Graph: $G = (V, E)$ where V = vertices, E = edges

Types:

- **Directed vs Undirected**: Edges have direction or not
- **Weighted vs Unweighted**: Edges have costs or not
- **Cyclic vs Acyclic**: Contains cycles or not (DAG)
- **Connected vs Disconnected**: All vertices reachable or not

Complexity Notation:

- V = number of vertices
- E = number of edges
- Dense graph: $E \approx V^2$
- Sparse graph: $E \approx V$

BFS/DFS Applications

Breadth-First Search (BFS)

Level-by-Level Exploration Using Queue

Algorithm:

1. Start from source vertex
2. Add to queue and mark visited
3. Dequeue, process, enqueue unvisited neighbors
4. Repeat until queue empty

Characteristics:

- **Time**: $O(V + E)$
- **Space**: $O(V)$ for queue and visited set
- **Data Structure**: Queue (FIFO)

Key Property:

- Finds **shortest path** in unweighted graphs
- Visits vertices in order of distance from source

BFS Implementation

```python
from collections import deque

def bfs(graph, start):
    """BFS traversal from start node"""
    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

# Example
graph = {
    0: [1, 2],
    1: [0, 3, 4],
    2: [0, 4],
    3: [1],
    4: [1, 2]
}
print(bfs(graph, 0))  # [0, 1, 2, 3, 4]
```

BFS Applications

1. **Shortest Path in Unweighted Graph:**
   - Find minimum number of edges from source to target
   - Track parent pointers to reconstruct path
2. **Level-Order Traversal:**
   - Process nodes by distance from source
   - Used in tree level-order traversal
3. **Connected Components:**
   - Find all disconnected subgraphs
   - Run BFS from each unvisited vertex
4. **Bipartite Check:**
   - Determine if graph is 2-colorable
   - Alternate colors during BFS
5. **Web Crawling:**
   - Explore web pages level by level

Depth-First Search (DFS)

Explore as Deep as Possible Using Stack/Recursion

Algorithm:

1. Start from source vertex
2. Mark visited, explore first unvisited neighbor
3. Recursively DFS on neighbor
4. Backtrack when no unvisited neighbors

Characteristics:

- **Time**: $O(V + E)$
- **Space**: $O(V)$ for recursion stack (or explicit stack)
- **Data Structure**: Stack (LIFO) or recursion

Key Property:

- Explores entire branch before backtracking
- Can detect cycles (back edges)

DFS Implementation

```python
def dfs_recursive(graph, node, visited=None):
    """DFS traversal (recursive)"""
    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

def dfs_iterative(graph, start):
    """DFS traversal (iterative)"""
    visited = set()
    stack = [start]
    result = []

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            result.append(node)
            stack.extend(reversed(graph[node]))  # Maintain order

    return result
```

DFS Applications

1. **Cycle Detection:**
   - Find back edges (edge to ancestor)
   - Detect if graph contains cycles
2. **Path Finding:**
   - Find any path between two nodes
   - Find all paths (with backtracking)
3. **Topological Sorting:**
   - Order vertices in DAG
   - Process finish times in reverse
4. **Maze Solving:**
   - Find path through grid
   - Backtrack on dead ends
5. **Strongly Connected Components:**
   - Kosaraju's and Tarjan's algorithms

BFS vs DFS Comparison

Feature	**BFS**	**DFS**
Data Structure	Queue	Stack/Recursion
Path Found	Shortest	Any path
Memory (worst)	$O(V)$ (wide)	$O(h)$ (height)
Best For	Shortest path	Cycle detection
Tree Traversal	Level-order	Pre/In/Post-order
Completeness	Yes	Yes

When to Use BFS:

- Find shortest path in unweighted graph
- Process nodes by distance
- Graph is very deep (avoid stack overflow)

When to Use DFS:

- Detect cycles, find topological order
- Explore all paths, backtracking problems

Shortest Paths: Dijkstra/Bellman-Ford

Dijkstra's Algorithm

Single-Source Shortest Path (Non-Negative Weights)

Algorithm:

1. Initialize distances: source = 0, others = ∞
2. Use min-heap to get vertex with minimum distance
3. For each neighbor, relax edge if shorter path found
4. Mark vertex as visited
5. Repeat until all vertices processed

Characteristics:

- **Time**: $O((V+E)\log V)$ with min-heap
- **Space**: $O(V)$
- **Requirement**: Non-negative edge weights

Key Idea:

- Greedy approach: always pick closest unvisited vertex
- Correctness requires non-negative weights

Dijkstra Implementation

```python
import heapq

def dijkstra(graph, start):
    """Find shortest paths from start to all vertices"""
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    pq = [(0, start)]  # (distance, node)
    visited = set()

    while pq:
        d, node = heapq.heappop(pq)
        if node in visited:
            continue
        visited.add(node)

        for neighbor, weight in graph[node]:
            new_dist = d + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))

    return dist

# Example: graph[node] = [(neighbor, weight), ...]
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('C', 1), ('D', 5)],
    'C': [('D', 8), ('E', 10)],
    'D': [('E', 2)],
    'E': []
}
```

Bellman-Ford Algorithm

Single-Source Shortest Path (Handles Negative Weights)

Algorithm:

1. Initialize distances: source = 0, others = ∞
2. Relax all edges $V - 1$ times
3. Check for negative cycles (one more iteration)

Characteristics:

- **Time**: $O(V \times E)$
- **Space**: $O(V)$
- **Advantage**: Handles negative weights, detects negative cycles

Why $V - 1$ Iterations?

- Longest simple path has $V - 1$ edges
- Each iteration extends shortest path by one edge
- After $V - 1$ iterations, all shortest paths found

Bellman-Ford Implementation

```
def bellman_ford(edges, n, start):
    """
    Find shortest paths, detect negative cycles
    edges: list of (u, v, weight)
    """
    dist = [float('inf')] * n
    dist[start] = 0

    # Relax edges V-1 times
    for _ in range(n - 1):
        for u, v, weight in edges:
            if dist[u] != float('inf') and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight

    # Check for negative cycles
    for u, v, weight in edges:
        if dist[u] != float('inf') and dist[u] + weight < dist[v]:
            return None  # Negative cycle detected

    return dist

# Example
edges = [(0, 1, 4), (0, 2, 2), (1, 2, -3), (2, 3, 2), (3, 1, 1)]
n = 4
print(bellman_ford(edges, n, 0))
```

Dijkstra vs Bellman-Ford

Feature	Dijkstra	Bellman-Ford
Time Complexity	$O(E \log V)$	$O(V \times E)$
Negative Weights	No	Yes
Negative Cycles	Cannot detect	Detects
Implementation	Min-heap	Nested loops
Best For	Fast, non-negative	Negative weights

Floyd-Warshall (All-Pairs Shortest Paths):

- **Time**: $O(V^3)$
- Finds shortest paths between all pairs
- Handles negative weights
- Uses dynamic programming

Choosing Algorithm:

- Non-negative weights $\rightarrow$ Dijkstra (faster)
- Negative weights $\rightarrow$ Bellman-Ford

Minimum Spanning Trees: Kruskal/Prim

Minimum Spanning Tree (MST)

Tree Connecting All Vertices with Minimum Total Weight

Properties:

- Connects all V vertices
- Has exactly $V - 1$ edges
- No cycles (it's a tree)
- Minimum total edge weight

Applications:

- Network design (minimize cable length)
- Approximation algorithms (TSP)
- Clustering algorithms
- Image segmentation

Two Main Algorithms:

- **Kruskal's**: Sort edges, add if no cycle (edge-based)
- **Prim's**: Grow tree from vertex (vertex-based)

Kruskal's Algorithm

Sort Edges, Add If Doesn't Create Cycle

Algorithm:

1. Sort all edges by weight (ascending)
2. Initialize Union-Find structure
3. For each edge (u, v):
   - If u and v in different components, add edge
   - Union the components
4. Stop when $V - 1$ edges added

Characteristics:

- **Time**: $O(E \log E)$ (dominated by sorting)
- **Space**: $O(V)$ for Union-Find
- **Best for**: Sparse graphs

Key Data Structure:

- Union-Find (Disjoint Set Union) for cycle detection

Kruskal Implementation

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False  # Already in same set
        if self.rank[px] < self.rank[py]:
            self.parent[px] = py
        elif self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[py] = px
            self.rank[px] += 1
        return True

def kruskal(n, edges):
    """edges: list of (weight, u, v)"""
    edges.sort()  # Sort by weight
    uf = UnionFind(n)
    mst, total = [], 0

    for weight, u, v in edges:
        if uf.union(u, v):
```

Prim's Algorithm

Grow Tree from Starting Vertex

Algorithm:

1. Start with any vertex
2. Add to MST
3. Repeat until all vertices added:
    - Find minimum weight edge from MST to non-MST vertex
    - Add that edge and vertex to MST

Characteristics:

- **Time**: $O(E \log V)$ with min-heap
- **Space**: $O(V)$
- **Best for**: Dense graphs

Key Data Structure:

- Min-heap to efficiently find minimum edge

Similarity to Dijkstra:

Prim Implementation

```python
import heapq

def prim(graph, start):
    """
    graph: {node: [(neighbor, weight), ...]}
    """
    mst, total = [], 0
    visited = {start}
    edges = [(weight, start, neighbor)
             for neighbor, weight in graph[start]]
    heapq.heapify(edges)

    while edges:
        weight, u, v = heapq.heappop(edges)
        if v in visited:
            continue

        visited.add(v)
        mst.append((u, v, weight))
        total += weight

        for neighbor, w in graph[v]:
            if neighbor not in visited:
                heapq.heappush(edges, (w, v, neighbor))

    return total, mst

# Example
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('A', 4), ('C', 1), ('D', 5)]
```

Kruskal vs Prim

Feature	Kruskal	Prim
Time Complexity	$O(E \log E)$	$O(E \log V)$
Approach	Edge-based	Vertex-based
Data Structure	Union-Find	Min-heap
Best For	Sparse graphs	Dense graphs
Starting Point	N/A (all edges)	Any vertex
Works on Disconnected	Partial MST	No

Time Complexity Notes:

- Kruskal: $O(E \log E) = O(E \log V)$ since $E \leq V^2$
- Prim with Fibonacci heap: $O(E + V \log V)$ (theoretical)

Practical Choice:

- Sparse graph ($E \approx V$): Kruskal slightly better
- Dense graph ($E \approx V^2$): Prim slightly better
- Both give same MST weight (may differ in edges)

Topological Sort and DAG DP

Topological Sort

Linear Ordering of Vertices in DAG

Definition:

- For every directed edge (u, v), u comes before v in ordering
- Only exists for Directed Acyclic Graphs (DAGs)
- Can be multiple valid orderings

Applications:

- **Course scheduling**: Prerequisite dependencies
- **Build systems**: Compile dependencies (Makefile)
- **Task scheduling**: Task dependencies
- **Formula evaluation**: Dependency graphs

Two Main Algorithms:

- **Kahn's Algorithm**: BFS-based, uses in-degrees
- **DFS-based**: Process vertices by finish time

Topological Sort: Kahn's Algorithm

```python
from collections import deque

def topological_sort(graph, n):
    """
    Kahn's algorithm (BFS-based)
    graph: adjacency list
    Returns: topological order or None if cycle exists
    """
    # Calculate in-degrees
    in_degree = [0] * n
    for node in range(n):
        for neighbor in graph[node]:
            in_degree[neighbor] += 1

    # Start with vertices having in-degree 0
    queue = deque([i for i in range(n) if in_degree[i] == 0])
    result = []

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

        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # If not all vertices processed, graph has cycle
    return result if len(result) == n else None
```

DAG Dynamic Programming

DP on Directed Acyclic Graphs

Key Idea:

- Process vertices in topological order
- Each vertex computed after all dependencies
- No cycles → no circular dependencies

Common Problems:

- **Longest/Shortest Path in DAG**: $O(V + E)$
- **Count paths**: Number of paths from source to sink
- **Critical Path Method**: Project scheduling

Template:

1. Compute topological order
2. Initialize DP array
3. Process vertices in topological order
4. Update DP based on edges

DAG DP: Longest Path

```
def longest_path_dag(graph, n):
    """Find longest path in DAG"""
    topo_order = topological_sort(graph, n)
    if topo_order is None:
        return None  # Cycle exists

    dp = [0] * n

    for node in topo_order:
        for neighbor in graph[node]:
            dp[neighbor] = max(dp[neighbor], dp[node] + 1)

    return max(dp)

# Example: Course scheduling with prerequisites
# Find longest chain of courses
graph = {
```

Strongly Connected Components

Strongly Connected Components (SCC)

Maximal Subgraphs Where Every Vertex Reaches Every Other

Definition:

- In directed graph, SCC is maximal set of vertices
- For any u, v in SCC, there exists path $u \to v$ and $v \to u$
- Condensation graph (SCC graph) is always a DAG

Applications:

- **2-SAT**: Satisfiability of Boolean formulas
- **Reachability queries**: Which vertices can reach which
- **Deadlock detection**: Circular dependencies
- **Web page ranking**: Identify tightly connected clusters

Algorithms:

- **Kosaraju's**: Two-pass DFS (simpler)
- **Tarjan's**: Single-pass DFS (more efficient)

Kosaraju's Algorithm

Two-Pass DFS Approach

```
def kosaraju_scc(graph, n):
    """Find strongly connected components"""
    # Step 1: DFS to get finish order
    visited = [False] * n
    finish_order = []

    def dfs1(node):
        visited[node] = True
        for neighbor in graph[node]:
            if not visited[neighbor]:
                dfs1(neighbor)
        finish_order.append(node)

    for i in range(n):
        if not visited[i]:
            dfs1(i)

    # Step 2: Transpose graph
    transpose = [[] for _ in range(n)]
    for u in range(n):
        for v in graph[u]:
            transpose[v].append(u)

    # Step 3: DFS on transpose in reverse finish order
    visited = [False] * n
    components = []

    def dfs2(node, component):
        visited[node] = True
```

Kosaraju's Algorithm: How It Works

Three Steps:

Step 1: First DFS

- Run DFS on original graph
- Record finish times (when vertex fully explored)
- Vertices in same SCC finish close together

Step 2: Transpose Graph

- Reverse all edge directions
- If $u \to v$ in G, then $v \to u$ in G^T
- SCCs remain the same

Step 3: Second DFS

- Process vertices in reverse finish order
- Each DFS tree in G^T is one SCC

Complexity:

Flow Algorithms Overview

Maximum Flow Problem

Find Maximum Flow from Source to Sink

Definition:

- Given: Directed graph with edge capacities
- Find: Maximum amount of flow from source s to sink t
- Constraints: Flow $\leq$ capacity, flow conservation

Key Concepts:

- **Capacity**: Maximum flow on edge
- **Residual graph**: Remaining capacity after flow
- **Augmenting path**: Path with available capacity
- **Cut**: Partition of vertices into two sets

Max-Flow Min-Cut Theorem:

- Maximum flow value = Minimum cut capacity
- Fundamental result in network flow theory

Ford-Fulkerson Method

Find Augmenting Paths Until No More Exist

Algorithm:

1. Initialize flow to 0
2. While augmenting path exists:
    - Find augmenting path (any path with capacity)
    - Compute bottleneck capacity
    - Add flow along path
    - Update residual graph

Edmonds-Karp Algorithm:

- Ford-Fulkerson with BFS for finding paths
- **Time**: $O(V \times E^2)$ (guaranteed polynomial)
- Always finds shortest augmenting path

Characteristics:

- **Time**: $O(V \times E^2)$ for Edmonds-Karp
- **Space**: $O(V^2)$ for flow/capacity matrices

Edmonds-Karp Implementation

```python
from collections import deque

def max_flow(capacity, source, sink):
    """Edmonds-Karp algorithm for maximum flow"""
    n = len(capacity)
    flow = [[0] * n for _ in range(n)]

    def bfs():
        """Find augmenting path using BFS"""
        parent = [-1] * n
        visited = [False] * n
        visited[source] = True
        queue = deque([(source, float('inf'))])

        while queue:
            u, min_cap = queue.popleft()

            for v in range(n):
                if not visited[v] and capacity[u][v] - flow[u][v] > 0:
                    visited[v] = True
                    parent[v] = u
                    new_cap = min(min_cap, capacity[u][v] - flow[u][v])

                    if v == sink:
                        return parent, new_cap
                    queue.append((v, new_cap))

        return None, 0

    total_flow = 0
    while True:
```

Flow Applications

1. **Maximum Bipartite Matching:**
   - Model as flow problem
   - Add source to left vertices, sink from right vertices
   - Maximum flow = maximum matching
2. **Minimum Cut:**
   - Find minimum capacity cut separating s and t
   - Max flow = min cut (by theorem)
   - Applications: Network reliability, image segmentation
3. **Network Routing:**
   - Optimize data flow through network
   - Consider bandwidth constraints
4. **Assignment Problems:**
   - Match workers to tasks optimally
   - Constraint satisfaction

Summary

Key Takeaways

Traversal Algorithms:

- **BFS**: Shortest path, level-order, $O(V + E)$
- **DFS**: Cycle detection, topological sort, $O(V + E)$

Shortest Path Algorithms:

- **Dijkstra**: Non-negative weights, $O(E \log V)$
- **Bellman-Ford**: Negative weights, detects cycles, $O(VE)$

Minimum Spanning Tree:

- **Kruskal**: Sort edges, Union-Find, $O(E \log E)$
- **Prim**: Grow tree, min-heap, $O(E \log V)$

Advanced Topics:

- **Topological Sort**: Order DAG vertices, $O(V + E)$
- **SCC**: Kosaraju's/Tarjan's, $O(V + E)$
- **Max Flow**: Edmonds-Karp, $O(VE^2)$

Complexity Summary

Algorithm	**Time**	**Space**
BFS	$O(V+E)$	$O(V)$
DFS	$O(V+E)$	$O(V)$
Dijkstra	$O((V+E)\log V)$	$O(V)$
Bellman-Ford	$O(VE)$	$O(V)$
Kruskal	$O(E\log E)$	$O(V)$
Prim	$O(E\log V)$	$O(V)$
Topological Sort	$O(V+E)$	$O(V)$
Kosaraju SCC	$O(V+E)$	$O(V)$
Edmonds-Karp	$O(VE^2)$	$O(V^2)$

Practice Problems

BFS/DFS:

- Number of islands (LeetCode 200)
- Word ladder (LeetCode 127)
- Course schedule (LeetCode 207, 210)

Shortest Paths:

- Network delay time (LeetCode 743)
- Cheapest flights (LeetCode 787)

MST:

- Min cost to connect all points (LeetCode 1584)

Advanced:

- Critical connections (LeetCode 1192)
- Alien dictionary (LeetCode 269)

Resources

Books:

- "Introduction to Algorithms" (CLRS) - Chapters 22-26
- "Algorithm Design" (Kleinberg & Tardos)

Online:

- VisuAlgo - Graph algorithm visualizations
- LeetCode - Graph problems
- Codeforces - Graph theory tutorials

Advanced Topics:

- A* search algorithm
- Network simplex for min-cost flow
- Hopcroft-Karp for bipartite matching
- Tarjan's algorithm for SCC