Foundations of Algorithm Design and Machine Learning - Graph Theory

[24BM6JP21]

Week 04 Summary

Summary of Graph Algorithms and BFS/DFS

1. Introduction to Graph Theory

Graph theory is fundamental in computer science and mathematics. A **graph** is a structure comprising:

- **Vertices (Nodes):** Represent entities (e.g., cities, users in a social network).
- **Edges:** Represent connections between nodes, which can be directed or undirected.

Mathematically, a graph is represented as **G = (V, E)**, where:

- **V:** Set of vertices (nodes).
- **E:** Set of edges representing relationships between vertices.

2. Graph Representation

Graphs can be represented in multiple ways:

Adjacency Matrix

- Uses a 2D array to indicate connections between nodes.
- **Time Complexity:**
    - Space Complexity: $O(V^2)$
    - Edge lookup: O(1)
    - Iterating over all edges: $O(V^2)$
- Suitable for dense graphs.

Adjacency List

- Maintains a list for each vertex with its neighbors.
- **Time Complexity:**
    - Space Complexity: O(V+E)
    - Edge lookup: O(V)
    - Iterating over all edges: O(V+E)
- Suitable for sparse graphs.

3. Depth-First Search (DFS)

DFS explores a graph deeply, visiting each branch before backtracking. It is used for:

- **Cycle Detection**
- **Topological Sorting**
- **Finding Connected Components**

DFS Algorithm Steps (Pseudocode give in weekly slides):

1. Start at the initial node and mark it as visited.
2. Recursively explore each unvisited neighbor.
3. Backtrack when no unvisited neighbors are left.
4. Repeat until all nodes are visited.

DFS Time Complexity:

- O(V+E) for adjacency list
- $O(V^2)$ for adjacency matrix

Applications of DFS:

- Solving Constraint Satisfaction Problems (CSPs) like crosswords and Sudoku.
- Detecting cycles in scheduling problems.
- Finding shortest paths in board games like Snakes and Ladders.

---

4. Breadth-First Search (BFS)

BFS explores a graph level by level, using a **queue** to keep track of nodes.

BFS Steps (Pseudocode give in weekly slides):

1. Enqueue the starting node and mark it as visited.
2. Dequeue the front node and explore its neighbors.
3. Enqueue each unvisited neighbor.
4. Repeat until the queue is empty.

BFS Time Complexity:

- O(V+E) for adjacency list
- $O(V^2)$ for adjacency matrix

Applications of BFS:

- Finding the shortest path in unweighted graphs.
- Solving network connectivity problems.

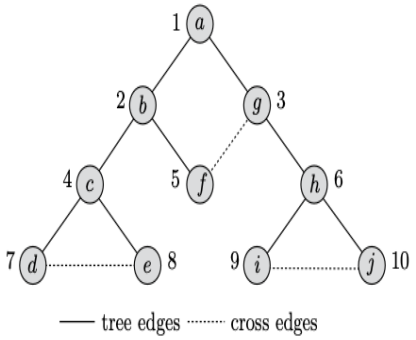


Fig. 8.7. An example of breadth-first search traversal of an undirected graph.

Directed Cyclic and Acyclic Graphs (DAGs)

In graph theory, graphs can either have cycles (cyclic) or be free of cycles (acyclic). When dealing with **directed graphs**, understanding the distinction between cyclic and acyclic graphs is crucial for applications like task scheduling, dependency resolution, and shortest path computation.

Directed Graphs:

A **directed graph** (also called a digraph) is a graph where each edge has a direction, indicating a one-way relationship between nodes.

Examples:

- **Webpages linking to each other** (A → B means A links to B)
- **Prerequisite tasks in a project** (Task A must be completed before Task B)

1. Directed Cyclic Graphs:

A **cyclic graph** contains one or more cycles.
A **cycle** is a path in which you can start at a node, follow directed edges, and return to the same node.

Example:

Consider nodes **A → B → C → A**. This forms a cycle, as starting at node A leads back to A.

Applications of Cycle Detection:

- Detecting **deadlocks** in operating systems
- **Circular dependencies** in software packages or databases

Cycle Detection with DFS:

- A **backward edge** in a DFS traversal indicates a cycle.

2. Directed Acyclic Graph (DAG):

A **DAG** is a directed graph without any cycles.
This type of graph is widely used in many fields because it represents dependencies and processes that must be done in a specific order.

Example:

- **Task scheduling:** A DAG can represent tasks in a project where certain tasks depend on the completion of others.
- **Course prerequisites:** Each course depends on completing earlier courses, forming a dependency structure.
- **Data processing pipelines:** Stages of data transformations must follow a specific sequence.

Topological Sorting in DAGs:

A **topological order** is a linear ordering of nodes such that for every directed edge u→v, node u comes before v. This ensures tasks or operations are processed in the correct sequence.
How to Find Topological Order:

- Use **DFS** to assign exit times.
- The **reverse order of exit times** gives the topological order.

Applications of DAGs:

- **Task Scheduling:** Ensures tasks are executed in the correct sequence.
- **Expression Trees:** DAGs represent mathematical expressions in compilers.
- **Version Control:** DAGs track changes and dependencies between commits.
- **Shortest Path Algorithms:** More efficient for DAGs than general graphs.

Conclusion

Understanding graph theory concepts, DFS, BFS, and their applications is essential for solving complex real-world problems, such as scheduling, shortest path computation, and constraint satisfaction.

Shortest Path in a DAG Algorithm

1. Directed Weighted Acyclic Graphs (DAGs)

Directed Weighted Acyclic Graphs are used to solve the shortest path problem more efficiently than general graphs by utilizing **topological sorting** and **edge relaxation**.

Shortest Path in a DAG Algorithm

The algorithm finds the shortest path from a source node to a destination node using a **priority queue (min-heap)**.

1. **Initialization:**
    - Create a distance array dist[ ] initialized to ∞ for all nodes except the source node (dist[source] = 0).
    - Maintain a queue and enqueue the source node with distance 0.
2. **Node Processing:**
    - Extract the node with the smallest distance from the queue.

- Relax all outgoing edges—update the distance for adjacent nodes if a shorter path is found and reinsert them into the queue.

Time Complexity: O(V+E) where V is the number of vertices and E is the number of edges.

---

2. Recursive Formulation of Dijkstra's Shortest Path

This formulation computes the shortest path by recursively exploring all successors and selecting the one with the smallest cost.

1. **Base Cases:**
   - If s=g, the shortest path from a node to itself is 0.
   - If G=NULL, the goal node is unreachable (path cost = ∞).
2. **Recursive Case:**
   - For each successor m of s, calculate the cost of traveling from s to m.
   - Recursively compute the shortest path from m to g and select the minimum cost.

---

3. Spanning Trees and Minimum Spanning Trees (MSTs)

A **spanning tree** of a graph connects all vertices without forming cycles. The goal of an **MST** is to find the tree with the minimum total edge weight.

Applications of MSTs:

- Network design (e.g., minimizing cable length in a communication network).
- Circuit design.
- Traveling Salesman Problem (sub-problem).

MST Algorithms:

1. **Prim's Algorithm:**
   - Greedily adds the minimum-weight edge connecting a vertex inside the tree to a vertex outside, ensuring no cycles are formed.
     Time Complexity: O(ElogV).

2. **Kruskal's Algorithm:**
    - Sorts all edges by weight and adds the smallest edges to the MST while avoiding cycles using the union-find data structure.
      Time Complexity: O(ElogV).

Conclusion

Understanding shortest path algorithms, recursive formulations, and minimum spanning trees is essential for optimizing network designs and solving real-world optimization problems. Techniques like Dijkstra's algorithm and MSTs have practical applications in routing, scheduling, and network optimization.

Shortest Path in DAG Using DFS and BFS

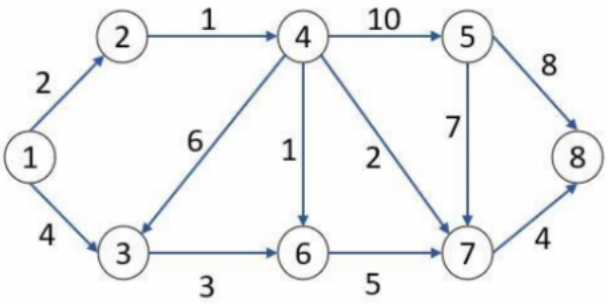
1. Shortest Path in Directed Weighted Acyclic Graph (DAG) using DFS with Memoization

In this approach, **Depth-First Search (DFS)** is used with **memoization** to optimize the shortest path calculation in a Directed Acyclic Graph (DAG).
Memoization helps store previously computed results to avoid redundant calculations, making it efficient compared to general graph algorithms like Dijkstra or Bellman-Ford.

Algorithm Steps:

1. Start at the source node.
2. Recursively explore all paths to the destination node.
3. Track and update the shortest cost for each node.
4. Use memoization to avoid recalculating costs for already visited nodes.

5. **Time Complexity:** O(V+E) — each node and edge is processed once.



Example:
Shortest path from node 1 to node 8:

- **Path:** 1 → 2 → 4 → 7 → 8
- **Total Cost:** 9

2. Shortest Path in DAG using Best-First Search (BFS)

Best-First Search (BFS) is another approach to finding the shortest path in a DAG. This method uses a **priority queue (min-heap)** to always expand the node with the current lowest cost, ensuring an optimal solution.

Algorithm Steps:

1. **Initialization:** Create a distance array (dist[]) initialized to ∞∞ and set the source node distance to 0. Use a priority queue to manage nodes.
2. **Processing Nodes:**
    - Extract the node with the smallest distance from the queue.
    - Relax all its outgoing edges.
    - If a shorter path is found, update the distance and reinsert it into the priority queue.
3. **Termination:** When all nodes are processed, return the shortest path cost to the destination.

Example:

- **Path:** 1 → 2 → 4 → 7 → 8
- **Total Cost:** 9