One-Way Streets

Robert Karapetyan

02/03/2025

Contents

1 Introduction

Think of a city laid out with one-way streets. Each intersection represents a subproblem that must be solved before you can continue your journey. Because every street goes forward—not back—you resolve each intersection's subproblem just once and carry that knowledge onward. That's dynamic programming: break the main task into smaller pieces, solve them in order, and use those solutions to build the final answer—no backtracking required.

2 The Blueprint: Directed Acyclic Graphs (DAGs)

A Directed Acyclic Graph (DAG) models our city:

- **Nodes (Intersections):** Each represents a subproblem.
- **Edges (One-Way Streets):** They enforce a single direction of travel from one subproblem to the next.
- **No Loops:** Ensuring a clear, forward-moving path.

2.1 A Weighted City Map

Figure 1 shows a DAG with intersections labeled 1 through 6. Arrows indicate one-way streets with weights representing travel costs.

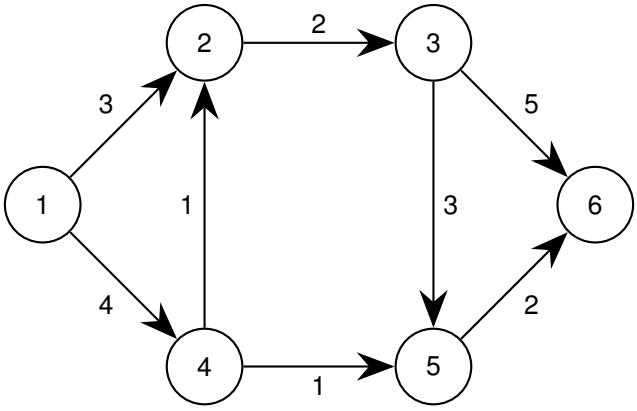

Figure 1: A weighted DAG representing one-way streets between intersections.

2.2 Adjacency Matrix

Another way to view the graph is through its weighted adjacency matrix. Each cell indicates the cost of traveling directly between two intersections (a zero indicates no direct path).

	1	2	3	4	5	6
1	0	3	0	4	0	0
2	0	0	2	0	0	0
3	0	0	0	0	3	5
4	0	1	0	0	1	0
5	0	0	0	0	0	2
6	0	0	0	0	0	0

Table 1: Adjacency matrix showing travel costs between intersections.

2.3 Linearized DAG

While the full DAG (Figure 1) and the matrix provide a complete picture, a linearized layout can simplify understanding of the dynamic programming process. Figure 2 arranges the intersections in order from start to destination to highlight the dependency chain.

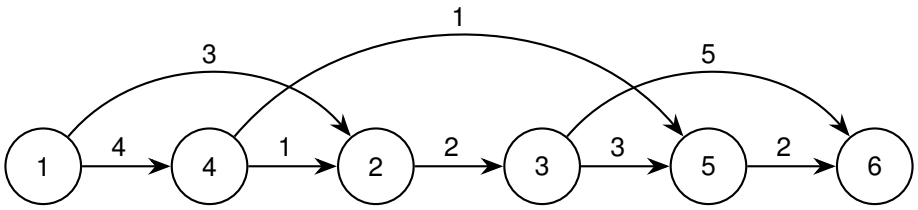

Figure 2: Linearized version of the weighted DAG with edge labels.

3 Problem: Finding the Shortest Path

Our objective is to find the least costly path from the starting intersection (1) to the destination (6). We do this by breaking the problem into smaller pieces. Define:

$$dp[i] = \text{minimum cost from intersection } i \text{ to intersection 6}$$

Because the DAG has no cycles, we can compute these values in reverse topological order. In practice, this means starting at the destination and working backward—ensuring that when we calculate $dp[i]$, every intersection reachable from i has already been processed.

3.1 Step-by-Step Calculation

Using the linearized layout as a guide, we compute the dp values:

- **Intersection 6:** Destination. Set:

$$dp[6] = 0.$$

- **Intersection 5:** Only one outgoing street: $5 \to 6$ (cost 2). Thus:

$$dp[5] = 2 + dp[6] = 2.$$

- **Intersection 3:** Two routes:
  - $3 \to 5$ (cost 3)
  - $3 \to 6$ (cost 5)

  Choose the cheaper:

$$dp[3] = \min(3 + dp[5],\, 5 + dp[6]) = \min(3 + 2,\, 5 + 0) = 5.$$

- **Intersection 2:** One move: $2 \to 3$ (cost 2). Hence:

$$dp[2] = 2 + dp[3] = 2 + 5 = 7.$$

- **Intersection 4:** Two options:
  - $4 \to 2$ (cost 1)
  - $4 \to 5$ (cost 1)

  So,

$$dp[4] = \min(1 + dp[2],\ 1 + dp[5]) = \min(1 + 7,\ 1 + 2) = \min(8, 3) = 3.$$

- **Intersection 1:** Two routes:
  - $1 \to 2$ (cost 3)
  - $1 \to 4$ (cost 4)

  Therefore,

$$dp[1] = \min(3 + dp[2],\ 4 + dp[4]) = \min(3 + 7,\ 4 + 3) = \min(10, 7) = 7.$$

3.2 Pseudocode Implementation

The following pseudocode summarizes the approach. It initializes the dp array, then processes nodes in reverse topological order—ensuring every dependency is resolved before a node's value is computed.

```
function ShortestPathDAG(graph, start, destination):
    // Initialize dp array with infinite cost for all nodes
    for each node in graph:
        dp[node] = Infinity
    dp[destination] = 0

    // Process nodes in reverse topological order
    for node in reverse_topological_order(graph):
        for each edge (node, neighbor) with cost c:
            dp[node] = min(dp[node], c + dp[neighbor])

    return dp[start]
```

3.3 Runtime Analysis

The algorithm works in three stages:

- **Initialization:** Setting up the dp array takes $O(n)$ time.
- **Reverse Topological Sorting:** Determining the processing order requires $O(n+m)$ time, where m is the number of edges.
- **Dynamic Programming Update:** Each edge is examined exactly once, contributing $O(m)$ time.

Overall, the runtime is:

$$O(n) + O(n+m) + O(m) = O(n+m)$$

This efficient complexity ensures that the solution scales even for large graphs.

4 Guidelines

A well-structured dynamic programming solution should clearly communicate your thought process and technical approach. Using our shortest path problem in a directed acyclic graph (DAG) as an example, follow these guidelines:

4.1 Problem Statement

- **Describe the Challenge:** Clearly state the problem in plain language. *Example:* "Find the minimum cost path from intersection 1 to intersection 6 in a DAG, where intersections are subproblems and one-way streets carry travel costs."

4.2 Subproblem Definition

- **Clarify the Table Entry:** Define what each entry in your dynamic programming table represents.

$$dp[i] = \text{minimum cost from intersection } i \text{ to intersection 6.}$$

4.3 Base Case

- **Establish the Foundation:** Identify and explain the base case that starts your recurrence.

$$dp[6] = 0.$$

Explanation: At the destination, no travel cost is incurred.

4.4 Recurrence Relation

- **Express the Transition:** Define how each subproblem relates to its subsequent subproblems.

$$dp[i] = \min_{(i \to j) \in E} \{c_{ij} + dp[j]\},$$

where c_{ij} is the cost from intersection i to j. *Explanation:* For each intersection i, consider every outgoing edge $(i \to j)$ and choose the one that minimizes the sum of the travel cost and the optimal cost from j onward.

4.5 Pseudocode Implementation

- **Outline the Algorithm:** Present concise pseudocode to illustrate the implementation.

```
function ShortestPathDAG(graph, start, destination):
    // Initialize dp table with high cost (Infinity) for all nodes
    for each node in graph:
        dp[node] = Infinity
```

```
dp[destination] = 0

// Process nodes in reverse topological order
for node in reverse_topological_order(graph):
    for each edge (node, neighbor) with cost c:
        dp[node] = min(dp[node], c + dp[neighbor])

return dp[start]
```

4.6 Runtime Analysis

- **Analyze Efficiency:** Briefly discuss the time complexity. *Example:* The algorithm runs in $O(n + m)$ time, where n is the number of intersections and m is the number of one-way streets.

4.7 Visual Aids

- **Enhance Understanding:** Include diagrams such as a graphical representation of the DAG and its linearized form, as well as a table (like the weighted adjacency matrix) to visually illustrate the problem and solution.

4.8 Summary and Key Takeaways

- **Recap the Approach:** Summarize how the problem was broken down:
  1. Define the subproblem ($dp[i]$).
  2. Set the base case ($dp[6] = 0$).
  3. Formulate the recurrence ($dp[i] = \min\{c_{ij} + dp[j]\}$).
  4. Implement the solution in pseudocode.
  5. Analyze the runtime.
- **Emphasize Clarity:** Each section of the solution should build on the previous one to form a complete, understandable, and efficient dynamic programming strategy.