Topic Introduction

Welcome back! Today, we're diving into one of the most fundamental and versatile tools in your coding interview toolkit: **graphs with weighted edges**.

What Are Weighted Graphs?

A **graph** is a collection of nodes (also called vertices) connected by edges. In a **weighted graph**, each edge has a numerical value (weight) associated with it. This weight might represent distance, cost, time, or even a mathematical ratio.

How Does It Work?

Imagine a city map where intersections are nodes, roads are edges, and each road has a travel time. Finding the fastest way from one place to another? That's a classic weighted graph problem!

Why Are Weighted Graphs Useful?

Interviews love these problems because they model real-world scenarios—like finding the cheapest flight, the fastest route, or even solving equations in disguise. They're also a natural way to test your understanding of Breadth-First Search (BFS), Depth-First Search (DFS), and Dijkstra's algorithm.

Quick Example (Not From Our Problems)

Suppose you have three cities: A, B, and C.

- A <-> B (cost: 5)
- B <-> C (cost: 10)

If you want to travel from A to C, the total cost is 5 + 10 = 15.

Easy, right? But what if you had to answer queries like, "What's the cheapest cost from A to C given these connections?" That's where weighted graphs shine!

The Three Problems Today

We're exploring:

- Evaluate Division
- Network Delay Time
- Cheapest Flights Within K Stops

Why are these grouped together?

All three are graph problems with weighted edges:

- **Evaluate Division**: Treats variables and equations as nodes and edges, using weights to represent ratios.
- **Network Delay Time**: Finds the shortest time (weight) for a signal to reach all nodes.
- **Cheapest Flights Within K Stops**: Finds the minimum cost (weight) path within a limited number of hops.

We'll start with the most intuitive, then build up to more complex constraints.

Problem 1: Evaluate Division

Problem link: [Evaluate Division - LeetCode]

In Your Own Words

Given equations like `a / b = 2.0` and queries like `a / c`, compute the result using the given relationships. If the answer cannot be determined, return -1.0.

Example Input/Output

Input:

- equations = [["a", "b"], ["b", "c"]]
- values = [2.0, 3.0]
- queries = [["a", "c"], ["c", "a"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"]]

Output: `[6.0, 1/6.0, 0.5, -1.0, 1.0, -1.0]`

Explanation:

- a / c = (a / b) *(b / c) = 2.0* 3.0 = 6.0
- c / a = 1 / (a / c) = 1/6.0
- b / a = 1 / (a / b) = 0.5
- a / e = -1.0 (e not connected)
- a / a = 1.0 (any number divided by itself)
- x / x = -1.0 (x not present in equations)

Try this one on paper:

- equations = [["x", "y"], ["y", "z"]], values = [4.0, 0.5], queries = [["x", "z"], ["z", "x"], ["x", "x"], ["w", "w"]]

Brute Force Approach

Try all possible paths for each query. This is slow: for every query, you might have to search all possible ways to connect the variables. **Time Complexity:** Exponential in the number of variables.

Optimal Approach: Graph + DFS

Core Pattern:

Model variables as nodes and equations as edges with weights. For `a / b = 2.0`, create an edge from `a` to `b` with weight 2.0, and from `b` to `a` with weight 0.5.

Steps:

- Build a graph from the equations and values.
- For each query, use DFS to try to find a path from the numerator to the denominator, multiplying the weights along the way.
- If no path exists, return -1.0.

Let's look at the code!

```
from collections import defaultdict

def calcEquation(equations, values, queries):
    # Step 1: Build the graph
    graph = defaultdict(dict)
    for (a, b), val in zip(equations, values):
        graph[a][b] = val
        graph[b][a] = 1 / val

    # Helper DFS function
    def dfs(start, end, visited):
        if start not in graph or end not in graph:
            return -1.0
        if start == end:
            return 1.0
        visited.add(start)
        for neighbor, weight in graph[start].items():
            if neighbor in visited:
                continue
            res = dfs(neighbor, end, visited)
            if res != -1.0:
                return res * weight
        return -1.0

    # Step 2: Answer each query
    results = []
    for num, denom in queries:
        results.append(dfs(num, denom, set()))
    return results
```

Time Complexity: O(Q * N), where Q is the number of queries and N is the number of variables (since in the worst case, DFS could visit all variables for each query).

Space Complexity: O(N + E), for the graph.

Code Walkthrough

- **graph:** Creates a mapping from each variable to its neighbors, storing the ratio.
- **dfs:** Recursively searches for a path from `start` to `end`, multiplying weights, and avoiding cycles with the `visited` set.
- **results:** For each query, runs dfs and records the result.

Example Trace

Let’s trace this input:

- equations = [["a", "b"], ["b", "c"]], values = [2.0, 3.0], queries = [["a", "c"]]

- Build graph:
  - a: {b: 2.0}
  - b: {a: 0.5, c: 3.0}
  - c: {b: 1/3.0}

- Query `a / c`:
  - dfs("a", "c", set()):
    - a's neighbor: b (weight 2.0)
    - dfs("b", "c", {"a"}):
      - b's neighbor: c (weight 3.0)
      - dfs("c", "c", {"a", "b"}): returns 1.0
      - returns 3.0 * 1.0 = 3.0
    - returns 2.0 * 3.0 = 6.0

Try this one yourself:

- equations = [["m", "n"], ["n", "o"]], values = [5.0, 2.0], queries = [["m", "o"], ["o", "m"], ["x", "y"]]

Take a moment to solve this on your own before checking the code!

Problem 2: Network Delay Time

Problem link: [Network Delay Time - LeetCode]

In Your Own Words

Given a network with `N` nodes and directed edges with travel times, determine how long it takes for a signal sent from node K to reach all nodes. If not all nodes can be reached, return -1.

Example Input/Output

Input:

- times = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
- N = 4
- K = 2

Output: 2

Explanation:

From node 2:

- 2 -> 1 (time 1)
- 2 -> 3 (time 1)
- 3 -> 4 (time 1 more, total 2)

Total time to reach all nodes is 2.

Try this one:

- times = [[1, 2, 1], [2, 3, 2], [1, 3, 4]], N = 3, K = 1

Brute-Force Approach

Try all paths from K to every other node, tracking the shortest. This is essentially running BFS/DFS from K to each node, which can be slow if the network is large.

Optimal Approach: Dijkstra's Algorithm

Core Pattern:

Find the shortest path from one node to all others in a weighted graph with non-negative weights.

Steps:

- Build an adjacency list from the input.
- Use a min-heap (priority queue) to always process the next node with the smallest known distance.
- Track the shortest time to each node.
- When all nodes are reached, return the longest (i.e., slowest) shortest path. If some nodes are unreachable, return -1.

Pseudocode:

```
Initialize graph as adjacency list: node -> list of (neighbor, time)
Initialize min-heap with (0, K)
Initialize dictionary dist to track the shortest time to each node

While heap is not empty:
    Pop (curr_time, node)
    If node already in dist: continue
    Set dist[node] = curr_time
    For all neighbors of node:
```

```
        If neighbor not in dist:
            Push (curr_time + time to neighbor, neighbor) to heap

If all N nodes are in dist:
    Return max(dist.values())
Else:
    Return -1
```

Example Trace

Input: times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2

- Start at 2, push (0, 2).
- Pop (0, 2): neighbors 1 and 3, push (1, 1), (1, 3).
- Pop (1, 1): neighbors none.
- Pop (1, 3): neighbor 4, push (2, 4).
- Pop (2, 4): neighbors none.

dist = {2:0, 1:1, 3:1, 4:2}
All nodes reached. Max is 2.

Try this one:

- times = [[1,2,1],[2,3,2]], N = 3, K = 1

Time Complexity: O(E log N), where E is the number of edges and N the number of nodes.
Space Complexity: O(N + E).

Problem 3: Cheapest Flights Within K Stops

Problem link: [Cheapest Flights Within K Stops - LeetCode]

In Your Own Words

Given a list of flights as `from`, `to`, `cost`, find the cheapest price from `src` to `dst` with at most `K` stops. If no such route exists, return -1.

Example Input/Output

Input:

- n = 4, flights = [[0,1,100],[1,2,100],[2,3,100],[0,3,500]]
- src = 0, dst = 3, K = 1

Output: 200

Explanation:
Route: 0 -> 1 -> 2 -> 3 (cost 300, 2 stops, too many)
Route: 0 -> 3 (cost 500, 0 stops)
Route: 0 -> 1 -> 2 -> 3 (first two legs, 2 stops; but with K=1, can only have at most 1 stop!)
So route: 0 -> 1 -> 2 (not end)
So, 0 -> 1 -> 2 -> 3 is too many stops.
But 0 -> 1 -> 2 -> 3 with K=2 is allowed.
But from 0 to 1 to 2 costs 200, but doesn't reach 3.
But the best with 1 stop is 0 -> 1 -> 2 (not reaching 3).

Wait: Let's clarify.
With K=1, at most 1 stop between src and dst (i.e., at most 2 edges).
So, 0 -> 1 -> 2 (1 stop), but doesn't reach 3.

Only valid paths:

- 0 -> 3 (0 stops), cost 500
- 0 -> 1 -> 2 (1 stop), but does not reach 3.

So, answer is 500.

Correction: Let's pick a more illustrative example.

Example:

- n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]]
- src=0, dst=2, K=1

Possible routes:

- 0 -> 2: cost 500, 0 stops
- 0 -> 1 -> 2: cost 200, 1 stop

So, answer is 200.

Try this one:

- n=3, flights=[[0,1,100],[1,2,100],[0,2,500]], src=0, dst=2, K=0

Brute-Force Approach

Try all possible routes from src to dst with no more than K stops. This is exponential in the number of flights and stops.

Optimal Approach: Modified BFS or Dijkstra

Core Pattern:
We need the cheapest path with at most K stops. This is like BFS with an extra constraint: the number of stops.

Steps:

- Build an adjacency list for the flights.
- Use a min-heap or BFS queue, each element tracking: (current cost, current node, stops so far).
- For each node, if the number of stops exceeds K+1, skip.
- If at destination, track the minimum cost.
- For each neighbor, push (cost + edge cost, neighbor, stops + 1) to the heap/queue.

Pseudocode:

```
Initialize adjacency list: node -> list of (neighbor, cost)
Initialize min-heap with (0, src, 0)
While heap not empty:
    Pop (cost, node, stops)
    If node == dst: return cost
    If stops > K: continue
    For neighbor, edge_cost in graph[node]:
        Push (cost + edge_cost, neighbor, stops + 1)
If no path found: return -1
```

Try this one:

- n=4, flights=[[0,1,1],[0,2,5],[1,2,1],[2,3,1]], src=0, dst=3, K=1

Time Complexity: O(E * K), since each node can be visited multiple times with different stop counts.
Space Complexity: O(N + E).

Summary and Next Steps

What Did We Learn?

All three problems are about exploring **weighted graphs**:

- **Evaluate Division:** Graph traversal to compute ratios using DFS.
- **Network Delay Time:** Shortest-path to all nodes using Dijkstra.
- **Cheapest Flights Within K Stops:** Shortest-path with a constraint using BFS or Dijkstra.

Key Patterns:

- Modeling the problem as a graph.
- Choosing the right traversal (DFS, BFS, Dijkstra).
- Using data structures like heaps or queues to enforce priority or constraints.

Common Traps:

- Not handling cycles (in Evaluate Division).
- Forgetting to track visited nodes or shortest paths.

- Off-by-one errors in stop count (in Cheapest Flights).
- Not building the graph correctly (especially for directed/undirected edges).

Action List

- Solve all 3 problems on your own, including the one with a sample code.
- Try solving Problem 2 and 3 using a different approach, like BFS for Network Delay or Bellman-Ford for Cheapest Flights.
- Explore similar problems: e.g., "Word Ladder" or "Minimum Spanning Tree" for more graph practice.
- Compare your solution with others—look for edge-case handling and code style.
- If you get stuck, don’t worry! The key is to keep practicing and reflecting.

Happy graph traversing!