Shortest Path Algorithms

Data Structures and Algorithms Tutorial

IIITH - S26

Bellman-Ford, Dijkstra, Floyd-Warshall (C++)

Agenda

1. Bellman-Ford
2. Dijkstra Revisited
3. Floyd-Warshall
4. Problems

Bellman-Ford

What Breaks Dijkstra?

Dijkstra works great until someone adds a negative weight edge.

The problem: Dijkstra commits to a node's distance the moment it pops from the priority queue. It's greedy. If a cheaper path exists through a negative edge discovered later, Dijkstra has already moved on.

Bellman-Ford doesn't commit:

- Tries every single edge, over and over
- If a shorter path exists, it will find it eventually
- Handles negative weights
- Detects negative cycles

Trade-off: $O(n \cdot m)$ vs $O((n + m) \log n)$ for Dijkstra.

The Core Insight

Claim: Any shortest path in a graph with n nodes uses at most $n - 1$ edges.

Why? A path with more than $n - 1$ edges must revisit some node, which means there's a cycle. If that cycle has positive total weight, skip it. If it has negative weight, "shortest path" becomes undefined.

So the algorithm:

1. dist[source] = 0, everything else = ∞
2. Repeat $n - 1$ times: try to relax every edge
3. Relax $(u \to v, w)$: if dist[u] + w < dist[v], update dist[v]

After $n - 1$ rounds, if no negative cycles exist, all shortest paths are finalized.

Relaxation: What It Means

"Relax" just means: can we do better?

```
if dist[u] + w < dist[v]:
    dist[v] = dist[u] + w
```

Round 1 can only improve 1-hop paths. Round 2 can improve 2-hop paths. After $n - 1$ rounds, all paths of any length are covered.

Early exit: if an entire round passes with zero updates, stop. No more improvements possible.

VisuAlgo: Bellman-Ford visualization

Negative Cycles

If a negative cycle exists, you can loop it forever and make paths arbitrarily short. There's no meaningful "shortest path" anymore.

Detection: run one extra (n-th) round. If any distance still decreases, the graph has a negative cycle reachable from source.

Reconstruction (see code/cycle_finding.cpp):

- Track parent[v] every time you relax
- The node that relaxes in round n is on or near a cycle
- Walk n steps back through parent[] to land inside the cycle
- Trace forward until you revisit a node: that's your cycle

Implementation: Main Loop

```
bool bellman_ford(int source) {
    for(int i = 0; i < n; i++) dist[i] = INF;
    dist[source] = 0;

    for(int iter = 0; iter < n-1; iter++) {
        bool updated = false;
        for(int i = 0; i < m; i++) {
            int u = edges[i].from, v = edges[i].to;
            ll w = edges[i].weight;
            if(dist[u] != INF && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                updated = true;
            }
        }
        if(!updated) break;
    }
```

Negative cycle check: one more round

```
    for(int i = 0; i < m; i++) {
        if(dist[edges[i].from] != INF &&
           dist[edges[i].from] + edges[i].weight < dist[edges[i].to])
            return true;
    }
    return false;
}
```

Returns true if a negative cycle was detected. Full code in code/bellman_ford.cpp.

Complexity & When to Use

Time: $O(n \cdot m)$ ($n - 1$ rounds, m edges per round)

Space: $O(n + m)$

Use Bellman-Ford when:

- Graph has negative weight edges
- You need to detect or reconstruct negative cycles
- $n \leq 10^3$, $m \leq 10^4$ (roughly)

Don't use it when:

- All weights are non-negative (use Dijkstra instead)
- Graph is large (will TLE)

Dijkstra Revisited

Why Negative Edges Break Dijkstra

Edges: 1->2 (4), 1->3 (2), 3->2 (-5)

Shortest path to 2 should be 1 -> 3 -> 2 = $2 + (-5) = -3$.

What Dijkstra does:

1. dist = [0, INF, INF]
2. Pop node 1, set dist[2] = 4, dist[3] = 2
3. Pop node 3, try to update dist[2] = min(4, -3) = -3… but node 2 is already finalized

Dijkstra assumes: once you finalize a node, no future path can beat it. With negative edges, this assumption is wrong.

Floyd-Warshall

Think Bigger: All Pairs

Dijkstra and Bellman-Ford: shortest path from one source.

Floyd-Warshall: shortest path between every pair of nodes, all at once.

Why not run Dijkstra n times? You can, and for sparse graphs it's fine ($O(n \cdot (n + m) \log n)$). Floyd-Warshall wins when:

- The graph is dense and n is small ($n \leq 500$)
- Negative edges are present (no negative cycles)
- You want 10 lines of code instead of 30

Read more: USACO Guide: Shortest Paths

The DP Idea

Key observation: a shortest path from i to j either goes directly, or passes through some intermediate node k.

Build up shortest paths by adding one intermediate node at a time:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \quad \text{dist}[i][k] + \text{dist}[k][j])$$

Do this for every pair (i, j), for every possible intermediate k. The outer loop over k must come first: you can only use intermediates $1..k$ at step k.

Example: Matrix Evolution

Graph: 1-2 (5), 1-4 (9), 1-5 (1), 2-3 (2), 3-4 (7), 4-5 (2)

Initial (direct edges only):

```
    1   2   3   4   5
1 [ 0   5 INF   9   1 ]
2 [ 5   0   2 INF INF ]
3 [INF  2   0   7 INF ]
4 [ 9 INF   7   0   2 ]
5 [ 1 INF INF   2   0 ]
```

After $k = 1$: node 1 as intermediate. New paths appear:

- dist[2][4] = min(INF, 5+9) = 14
- dist[2][5] = min(INF, 5+1) = 6
- dist[4][5] = min(2, 9+1) = 2 (no improvement)

Example: Continued

After $k = 2$ (intermediates: 1 or 2):

- dist[1][3] = min(INF, 5+2) = 7
- dist[3][5] = min(INF, 2+6) = 8

After $k = 3$: dist[2][4] = min(14, 2+7) = 9

Final matrix:

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

Shortest path from 2 to 4 is 8: goes through nodes 1 and 5.

Implementation

```
void floyd_warshall() {
    for(int k = 1; k <= n; k++)
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                if(dist[i][k] < INF && dist[k][j] < INF)
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
}
```

Initialization:

```
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
        dist[i][j] = (i == j) ? 0 : INF;

for each edge (u, v, w):
    dist[u][v] = min(dist[u][v], w);
    dist[v][u] = min(dist[v][u], w);
```

Three things to remember:

- Guard against INF + INF overflow before adding
- Take min for parallel edges on init
- Negative cycle detection: after running, dist[i][i] < 0 means node i is on one

Problems

Problem 1: Investigation

CSES 1202 - Investigation

Given a directed weighted graph, find all of the following for paths from node 1 to node n:

- The minimum path length
- Number of minimum-length paths (mod $10^9 + 7$)
- Minimum number of edges among all shortest paths
- Maximum number of edges among all shortest paths

$n, m \leq 2 \times 10^5$, edge weights up to 10^9.

Problem 1: Hint

Dijkstra gives shortest distances. But here you need to track four things per node simultaneously, not one.

Key question: when you relax edge $u \to v$ and find a shorter path, what happens to the count and edge-count? What if the new path has the same length as what you already found?

These two cases (strictly better vs. equal distance) need completely different handling. Think about what it means to "merge" two equally-good paths into one node's state.

Problem 1: Approach

Alongside dist[], maintain cnt[], min_e[], max_e[].

When relaxing $u \to v$ with weight w:

- If dist[u] + w < dist[v]: update dist, reset cnt/min_e/max_e from u's values (+1 on edge counts)
- If dist[u] + w == dist[v]: keep dist, merge: add counts, take min/max of edge counts

Answer at node n: read cnt[n], min_e[n], max_e[n] directly.

The equal-distance merge case is the whole problem.

Problem 2: Shortcut

USACO 2019 Jan Gold - Shortcut

There are n farms and m roads. Every farm sends one cow to farm 1 each day along a shortest path. You can add one new road of fixed length T between any two farms. Minimize the total distance all cows travel.

$n \leq 10^4, m \leq 5 \times 10^4$.

Problem 2: Hint

Run Dijkstra from node 1. The shortest paths form a DAG rooted at 1. Cows flow "up" this DAG toward the root.

A new direct road from node v to node 1 with length T helps every cow whose path currently passes through v. The benefit is: each such cow saves `dist[v] - T` steps (if that's positive).

So the question becomes: how many cows pass through each node v? That's a subtree count on the shortest-path DAG.

Problem 2: Approach

1. Dijkstra from node 1, get dist[] for all nodes
2. Build the shortest-path DAG: edge $u \to v$ is in the DAG if dist[u] = dist[v] + w(v,u) (i.e., v is on a shortest path to 1 via u)
3. Reverse-topological DFS to compute subtree[v]: how many nodes have v on their shortest path to 1
4. For each candidate v: savings = subtree[v] * (dist[v] - T) if dist[v] > T
5. Pick maximum savings

Total: $O((n+m)\log n)$ for Dijkstra + $O(n+m)$ for the DAG traversal.

Problem 3: Greg and Graph

CF 295B - Greg and Graph

n nodes are revealed one by one. When node k is revealed, all edges between k and previously revealed nodes appear. After each reveal, output the sum of shortest distances between all pairs of currently active nodes.

$n \leq 500$.

Problem 3: Hint

Running Floyd-Warshall from scratch after each reveal is $O(n^4)$ total. Too slow.

Look at the Floyd-Warshall loop structure. The outer loop adds one intermediate node at a time. Does that remind you of anything?

What if you process reveals in reverse order, and treat each newly added node as the next intermediate in Floyd-Warshall?

Problem 3: Approach

Process node reveals in reverse order. Initialize the full adjacency matrix with edge weights.

When adding node k as the next intermediate, run one FW "layer":

```
for all active i, j:
    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

After each layer, sum dist[i][j] over all currently active pairs. Store answers in reverse, output in original order.

$O(n^3)$ total. The key insight: FW intermediates can be added one at a time, in any order.

Questions?

Thank you!