All Pair Shortest Path

Problem Variations Collection

Comprehensive Guide to APSP Problems

A complete collection of variations based on shortest path algorithms including Floyd-Warshall, Johnson's Algorithm, and cycle detection techniques.

Contents

1 Basic Shortest Path Variations

1.1 Problem 1.1: Classic APSP

Shortest Routes Between Cities

Description: Given a weighted graph representing cities and roads, answer multiple queries about shortest distances between city pairs.
Input:

- First line: n, m, q (cities, roads, queries) where $1 \leq n \leq 500$, $1 \leq m \leq n^2$, $1 \leq q \leq 10^5$
- Next m lines: a, b, c (road from a to b with length c, bidirectional)
- Next q lines: a, b (query for shortest path from a to b)

Output: For each query, print shortest distance or -1 if no path exists.
Algorithm: Floyd-Warshall

1.2 Problem 1.2: APSP with Path Reconstruction

Shortest Path with Route

Description: Same as Problem 1.1, but also output the actual path taken.
Output: For each query, print:

- Distance (or -1 if no path)
- If path exists: the sequence of cities in the shortest path

Algorithm: Floyd-Warshall with predecessor matrix

1.3 Problem 1.3: Directed Graph APSP

One-Way Roads

Description: All roads are one-way (directed edges).
Input: Same format as 1.1, but edge (a, b) only allows travel from a to b.
Note: Distance from a to b may differ from distance from b to a.
Algorithm: Floyd-Warshall (works for directed graphs)

1.4 Problem 1.4: APSP with Node Weights

Cities with Toll

Description: Each city has a toll cost. When passing through a city (except source and destination), add the toll to total cost.
Input:

- After n, m, q: line with n integers (toll for each city)
- Roads and queries as usual

Algorithm: Modified Floyd-Warshall or graph transformation

1.5 Problem 1.5: K-Shortest Paths

Second Best Route

Description: For each query, find the k-th shortest path between two cities.
Input: Additional parameter k in each query line.
Output: Length of k-th shortest path or -1 if fewer than k paths exist.
Algorithm: Yen's algorithm or recursive enumeration

2 Graph Diameter and Center Problems

2.1 Problem 2.1: Graph Diameter

Maximum Distance

Description: Find the diameter of the graph (maximum shortest distance between any two vertices).
Output: Single integer representing the diameter, or -1 if graph is disconnected.
Algorithm: Floyd-Warshall, then find maximum finite distance

2.2 Problem 2.2: Graph Center

Optimal City Location

Description: Find the city that minimizes the maximum distance to any other city.
Output: City number and the maximum distance from this city to any other city.
Algorithm: Floyd-Warshall, then for each vertex find max distance to others

2.3 Problem 2.3: Graph Radius

Minimum Eccentricity

Description: Find the radius of the graph (minimum eccentricity among all vertices).
Output: The radius value.
Note: Eccentricity of vertex v = max distance from v to any other vertex.

2.4 Problem 2.4: All Peripheral Vertices

Boundary Cities

Description: Find all vertices with eccentricity equal to the diameter.
Output: List of all peripheral vertices.
Algorithm: Compute all eccentricities using APSP

3 Cycle Detection and Analysis

3.1 Problem 3.1: Negative Cycle Detection

Detect Anomalies

Description: Determine if the graph contains any negative weight cycle.
Output: "YES" if negative cycle exists, "NO" otherwise.
Algorithm: Floyd-Warshall, check if $dist[i][i] < 0$ for any i

3.2 Problem 3.2: Minimum Weight Cycle

Smallest Loop

Description: Find the minimum weight cycle in the graph.
Output: Weight of minimum cycle, or -1 if no cycle exists.
Algorithm: Modified Floyd-Warshall checking $dist[i][j] + w(j, i)$

3.3 Problem 3.3: Girth of Graph

Shortest Cycle Length

Description: Find the length (number of edges) of the shortest cycle.
Output: Number of edges in shortest cycle, or -1 if acyclic.
Algorithm: Modified Floyd-Warshall with edge count

4 Currency Exchange and Arbitrage

4.1 Problem 4.1: Basic Arbitrage Detection

Currency Trading Opportunity

Description: Given exchange rates, determine if arbitrage is possible.
Input:

- n currencies (names)
- m exchange rates: source currency, rate, destination currency

Output: "Yes" or "No"
Algorithm: Convert to negative log weights, detect positive cycle using Floyd-Warshall

4.2 Problem 4.2: Maximum Arbitrage Path

Best Trading Sequence

Description: Find the sequence of currency exchanges that maximizes profit.
Output:

- Maximum multiplication factor
- Sequence of currencies in the arbitrage cycle

Algorithm: Bellman-Ford or modified Floyd-Warshall with path reconstruction

4.3 Problem 4.3: Arbitrage with Transaction Costs

Real-World Trading

Description: Each exchange has a fixed transaction cost or percentage fee.
Input: Additional cost parameter for each exchange rate.
Output: "Yes" if arbitrage still possible after costs, "No" otherwise.
Algorithm: Adjust rates by costs, then detect cycle

4.4 Problem 4.4: Multi-Currency Optimization

Best Exchange Path

Description: Given starting currency and amount, find best way to convert to target currency.
Input:

- Source currency and amount
- Target currency
- Exchange rates

Output: Maximum amount obtainable in target currency.
Algorithm: Floyd-Warshall with multiplicative weights

4.5 Problem 4.5: Time-Based Arbitrage

Temporal Exchange Rates

Description: Exchange rates change over time. Can you make profit with timed trades?
Input: Multiple sets of exchange rates with timestamps.
Output: Maximum profit possible with timeline constraints.
Algorithm: Dynamic programming with temporal graph

5 Constrained Shortest Path Problems

5.1 Problem 5.1: Limited Edge Count

Maximum K Hops

Description: Find shortest path using at most k edges.
Input: Additional parameter k for each query.
Output: Shortest distance using $\leq k$ edges.
Algorithm: Modified Floyd-Warshall with edge count dimension

5.2 Problem 5.2: Exactly K Edges

Fixed Length Path

Description: Find shortest path using exactly k edges.
Algorithm: Dynamic programming: $dp[k][i][j]$ = shortest path from i to j using exactly k edges

5.3 Problem 5.3: Forbidden Nodes

Avoiding Dangerous Cities

Description: Each query specifies cities to avoid.
Input: Each query has: source, destination, and list of forbidden intermediate cities.
Algorithm: Remove forbidden nodes temporarily and run modified Floyd-Warshall

5.4 Problem 5.4: Mandatory Waypoints

Tourist Route

Description: Path must visit specific waypoints in any order.
Input: Each query includes set of mandatory waypoints.
Output: Shortest path visiting all waypoints.
Algorithm: TSP-like approach with APSP preprocessing

5.5 Problem 5.5: Capacity Constraints

Weight Limited Roads

Description: Each road has weight limit. Find path with capacity $\geq w$.
Input: Each road has capacity. Each query has weight requirement.
Algorithm: Filter edges by capacity, then APSP

6 Dynamic and Update Problems

6.1 Problem 6.1: Single Edge Update

Road Repair

Description: After computing APSP, one edge weight changes. Update distances efficiently.
Input: Initial graph, then update: edge and new weight.
Algorithm: Incremental Floyd-Warshall or recompute affected paths

6.2 Problem 6.2: Multiple Edge Updates

Dynamic Road Network

Description: Handle sequence of edge weight updates and distance queries.
Algorithm: Rebuild APSP periodically or use dynamic algorithms

6.3 Problem 6.3: Edge Deletion

Closed Roads

Description: Handle queries after deleting edges.
Algorithm: Recompute or maintain alternative paths

6.4 Problem 6.4: Vertex Deletion

City Isolation

Description: Remove a city and update all shortest paths.
Algorithm: Remove vertex from consideration in Floyd-Warshall

7 Optimization Variants

7.1 Problem 7.1: Minimax Path

Minimize Maximum Edge

Description: Find path where maximum edge weight is minimized.
Output: For each query, the minimum possible maximum edge weight.
Algorithm: Modified Floyd-Warshall: $dist[i][j] = \min(dist[i][j], \max(dist[i][k], dist[k][j]))$

7.2 Problem 7.2: Maximin Path

Maximize Minimum Edge

Description: Find path where minimum edge weight is maximized (widest path).
Algorithm: Modified Floyd-Warshall with max-min operations

7.3 Problem 7.3: Maximum Reliability Path

Most Reliable Route

Description: Each edge has reliability probability. Find path with maximum probability.
Algorithm: Use logarithms and shortest path

7.4 Problem 7.4: Minimum Bottleneck Path

Largest Minimum Capacity

Description: Find path where minimum edge capacity is maximized.
Application: Network flow, bandwidth optimization

8 Counting and Enumeration

8.1 Problem 8.1: Count All Shortest Paths

Number of Optimal Routes

Description: Count number of shortest paths between each pair.
Output: For each query, number of distinct shortest paths.
Algorithm: Modified Floyd-Warshall maintaining path count

8.2 Problem 8.2: Count Paths with Length L

Fixed Distance Paths

Description: Count paths of exactly length L between vertices.
Algorithm: Matrix exponentiation or DP

8.3 Problem 8.3: Sum of All Distances

Total Network Distance

Description: Find sum of all pairwise shortest distances.
Output: Single number: $\sum_{i<j} dist(i,j)$
Algorithm: Floyd-Warshall then sum upper triangle

9 Special Graph Types

9.1 Problem 9.1: Grid Graph Shortest Paths

City Grid Navigation

Description: Cities arranged in grid, movements in 4 or 8 directions.
Algorithm: Floyd-Warshall or exploit grid structure

9.2 Problem 9.2: Layered Graph APSP

Multi-Level Network

Description: Graph has multiple layers with inter-layer connections.
Algorithm: 3D Floyd-Warshall or graph transformation

9.3 Problem 9.3: Planar Graph Shortest Paths

Map Navigation

Description: Guaranteed planar graph structure.
Algorithm: Exploit planarity for better complexity or use standard APSP

10 Transitive Closure Variants

10.1 Problem 10.1: Reachability Matrix

Connection Check

Description: Determine which pairs of vertices are connected.
Output: $n \times n$ boolean matrix.
Algorithm: Floyd-Warshall without weights (just connectivity)

10.2 Problem 10.2: Strongly Connected Components

Mutual Reachability

Description: Find all groups where every pair is mutually reachable.
Algorithm: Tarjan's or Kosaraju's, but can use APSP for small graphs

10.3 Problem 10.3: Transitive Reduction

Minimal Edge Set

Description: Remove redundant edges while preserving reachability.
Algorithm: Compute transitive closure, identify redundant edges

11 Time-Dependent and Stochastic Variants

11.1 Problem 11.1: Time-Varying Edge Weights

Rush Hour Traffic

Description: Edge weights change based on time of day.
Input: For each edge, function $w(e, t)$ giving weight at time t.
Output: Fastest path considering time-dependent weights.
Algorithm: Modified Dijkstra with time-dependent relaxation

11.2 Problem 11.2: Stochastic Shortest Paths

Uncertain Travel Times

Description: Edge weights are random variables with known distributions.
Output: Path minimizing expected travel time or maximizing reliability.
Algorithm: Expected value computation or probabilistic analysis

11.3 Problem 11.3: Scheduled Networks

Bus/Train Timetables

Description: Can only use edges at specific scheduled times.
Input: Each edge has departure/arrival times.
Algorithm: Time-expanded network model

12 Multi-Objective Optimization

12.1 Problem 12.1: Bi-criteria Shortest Path

Minimize Distance AND Cost

Description: Each edge has both distance and cost. Find Pareto-optimal paths.
Output: Set of non-dominated paths.
Algorithm: Multi-objective DP or label-setting algorithms

12.2 Problem 12.2: Time-Cost Tradeoff

Fast vs Cheap Routes

Description: Given budget constraint, minimize time. Or given time limit, minimize cost.
Algorithm: Constrained APSP with resource limits

12.3 Problem 12.3: Risk-Averse Routing

Minimize Variance

Description: Choose path with minimum variance in travel time.
Algorithm: Compute mean and variance for all paths

13 Resource-Constrained Problems

13.1 Problem 13.1: Battery-Limited Paths

Electric Vehicle Routing

Description: Vehicle has limited battery. Find path with charging stations.
Input: Battery capacity, consumption per edge, charging station locations.
Algorithm: Modified APSP with resource states

13.2 Problem 13.2: Fuel Constraints

Aircraft Routing

Description: Find path where fuel stations are reachable.
Algorithm: Resource-constrained shortest path

13.3 Problem 13.3: Multi-Resource Constraints

Complex Vehicle Routing

Description: Track multiple resources: fuel, time, money, cargo capacity.
Algorithm: Multi-dimensional DP

14 Turn Restrictions and Movement Constraints

14.1 Problem 14.1: No U-Turns

One-Way Streets

Description: Cannot reverse direction on consecutive edges.
Algorithm: Expand state space to include direction

14.2 Problem 14.2: Turn Costs

Intersection Delays

Description: Left turns cost more time than right turns.
Input: Turn costs for each vertex based on incoming/outgoing edges.
Algorithm: Expand graph to include edge-to-edge transitions

14.3 Problem 14.3: Limited Turns

Maximum K Turns

Description: Path can make at most k turns.
Algorithm: DP with turn count dimension

15 Advanced Graph Modifications

15.1 Problem 15.1: Vertex Duplication

Multiple Visits

Description: Can visit vertices multiple times with different costs.
Algorithm: Modified graph structure

15.2 Problem 15.2: Edge Reversal

Can Reverse K Edges

Description: Allowed to reverse direction of up to k edges.
Algorithm: Combinatorial optimization

15.3 Problem 15.3: Edge Addition

Build New Roads

Description: Can add up to k new edges. Which edges minimize total distance?
Algorithm: Optimization over possible edge additions

16 Meeting and Rendezvous Problems

16.1 Problem 16.1: Equidistant Meeting Point

Fair Meeting Location

Description: Find point where maximum distance from any person is minimized.
Algorithm: APSP + minimax optimization

16.2 Problem 16.2: Weighted Meeting Point

VIP Considerations

Description: Some people have higher weight/importance.
Algorithm: Weighted distance minimization

16.3 Problem 16.3: Sequential Meetings

Multiple Meeting Schedule

Description: Schedule k meetings at different locations optimally.
Algorithm: TSP-like problem with APSP preprocessing

17 Network Resilience and Robustness

17.1 Problem 17.1: Critical Edge Identification

Which Roads Are Essential?

Description: Find edges whose removal most increases shortest paths.
Algorithm: Test removal of each edge, measure impact

17.2 Problem 17.2: K-Edge Fault Tolerance

Backup Routes

Description: Find paths that remain valid if up to k edges fail.
Algorithm: Find edge-disjoint paths

17.3 Problem 17.3: Maximum Disconnection

Worst-Case Sabotage

Description: Remove k edges to maximize graph diameter.
Algorithm: Adversarial optimization

18 Metric and Distance Variations

18.1 Problem 18.1: Euclidean Shortest Path

Geometric Obstacles

Description: Find shortest path in 2D/3D space with obstacles.
Algorithm: Visibility graph + Dijkstra

18.2 Problem 18.2: Manhattan Distance

Grid Navigation

Description: Movement restricted to axis-aligned directions.
Algorithm: L1 metric instead of L2

18.3 Problem 18.3: Chebyshev Distance

8-Directional Movement

Description: Can move diagonally with same cost.
Algorithm: L-infinity metric

19 Historical and Special Variations

19.1 Problem 19.1: Steiner Tree Problem

Connect Terminal Vertices

Description: Connect specific vertices using minimum total edge weight.
Note: NP-hard, but related to APSP
Algorithm: Approximation algorithms or exact for small instances

19.2 Problem 19.2: Chinese Postman Problem

Traverse All Edges

Description: Find shortest walk that uses every edge at least once.
Algorithm: APSP + matching on odd-degree vertices

19.3 Problem 19.3: Traveling Salesman (Metric)

Visit All Cities Once

Description: Minimum cost tour visiting each city exactly once.
Algorithm: DP with APSP preprocessing

20 Practice Problems

Example

Problem A: Meeting Point
Given n people at different cities, find the city that minimizes total travel distance.
Solution: Compute APSP, then for each city sum distances from all people.

Example

Problem B: Network Latency
In a computer network, find the pair of computers with maximum communication delay.
Solution: Graph diameter problem using Floyd-Warshall.

Example

Problem C: Supply Chain
Multiple warehouses, multiple stores. Find optimal warehouse for each store.
Solution: APSP, then for each store find nearest warehouse.

Example

Problem D: Emergency Services
Place k ambulance stations to minimize maximum distance to any location.
Solution: K-center problem using APSP

Example

Problem E: Tourist Itinerary
Visit k landmarks, starting and ending at hotel. Minimize total distance.
Solution: TSP on subset after computing APSP

21 Rarely Seen But Possible Variations

21.1 Problem 20.1: Approximate APSP

Fast Approximation

Description: Find $(1 + \epsilon)$-approximate shortest paths faster.
Algorithm: Sampling or randomized techniques

21.2 Problem 20.2: Parallel/Distributed APSP

Multi-Core Computation

Description: Compute APSP using multiple processors.
Algorithm: Block-based Floyd-Warshall parallelization

21.3 Problem 20.3: Persistent Data Structure

Historical Queries

Description: Query shortest paths at any historical graph state.
Algorithm: Persistent APSP data structure

21.4 Problem 20.4: Monotone Shortest Paths

Always Increasing Coordinates

Description: Path must be monotone in some direction.
Algorithm: Specialized DP

21.5 Problem 20.5: Arrival Time Dependent

FIFO Property

Description: Later departure never arrives earlier.
Algorithm: Time-dependent network analysis

22 Hybrid and Combined Problems

22.1 Problem 21.1: APSP + Matching

Optimal Pairing

Description: Match n sources to n destinations minimizing total distance.
Algorithm: APSP + Hungarian algorithm

22.2 Problem 21.2: APSP + Flow

Multi-Commodity Flow

Description: Route multiple flows with capacity constraints.
Algorithm: Successive shortest paths

22.3 Problem 21.3: APSP + Coloring

Colored Edge Constraints

Description: Path can only use certain color sequences.
Algorithm: State-space expansion

22.4 Problem 21.4: APSP on Dynamic Graphs

Fully Dynamic Maintenance

Description: Handle arbitrary edge insertions and deletions.
Algorithm: Dynamic APSP algorithms (complex)

23 Domain-Specific Applications

23.1 Problem 22.1: Social Network Distance

Degrees of Separation

Description: Find distance between any two people in social network.
Algorithm: Unweighted APSP (BFS-based)

23.2 Problem 22.2: Circuit Delay Analysis

Electronic Timing

Description: Compute signal delays between all component pairs.
Algorithm: Weighted APSP on circuit graph

23.3 Problem 22.3: Protein Interaction Networks

Biological Pathways

Description: Find interaction distances in protein networks.
Algorithm: Specialized APSP with domain constraints

23.4 Problem 22.4: Urban Planning

Accessibility Analysis

Description: Compute accessibility scores for all city locations.
Algorithm: APSP + weighted aggregation

24 Exam Strategy and Recognition

24.1 How to Recognize APSP Problems

Key indicators that a problem needs APSP:

- Problem asks about "all pairs" or "every pair"
- Need to answer multiple (¿100) shortest path queries
- $n \leq 500$ (small enough for $O(n^3)$)
- Problem involves: graph diameter, center, median, eccentricity
- Currency exchange / arbitrage problems
- Transitive closure or reachability queries
- Problems starting with "for every pair of cities/nodes..."

Disguised APSP problems:

- "Find the two most distant points in network"
- "Identify the most central location"
- "Determine if profitable cycle exists" (arbitrage)
- "Count unreachable pairs"
- "Find bottleneck in every path"
- "Optimize meeting point for multiple parties"

24.2 Common Tricks and Variations

Multi-Graph Layers

Different edge types (road, rail, air) - Create layered graph

Bidirectional Transformation

Different costs for each direction - Use directed edges

Negative Weights Allowed

Explicitly mentions "can have negative weights" - Still use Floyd-Warshall

Path Validation

Additional constraints on valid paths - Modify relaxation condition

25 Complete Problem Classification

25.1 By Algorithm Choice

- **Floyd-Warshall:** Dense graphs, $n \leq 500$, simple queries
- **Johnson's:** Sparse graphs, negative edges allowed
- **Repeated Dijkstra:** Sparse graphs, non-negative weights
- **Matrix Multiplication:** Theoretical interest, special cases

25.2 By Graph Properties

- **Undirected:** Symmetric distance matrix
- **Directed:** Asymmetric distances
- **Weighted:** Standard APSP
- **Unweighted:** BFS from each vertex
- **DAG:** Topological order + DP

25.3 By Optimization Goal

- **Minimize sum:** Classic shortest path
- **Minimize maximum:** Minimax/bottleneck
- **Maximize minimum:** Widest path
- **Maximize product:** Currency (use logs)
- **Multiple criteria:** Pareto-optimal solutions

26 Critical Exam Tips and Edge Cases

26.1 Common Pitfalls to Avoid

- **Self-loops:** Initialize $dist[i][i] = 0$ AFTER reading edges
- **Multiple edges:** Take minimum weight between same vertices
- **Integer overflow:** Use appropriate INF value (10^9 if edge weights $\leq 10^9$)
- **Negative cycles:** Check $dist[i][i] < 0$ after algorithm
- **Unreachable pairs:** Return -1, not INF
- **Path reconstruction:** Need parent matrix, not just distance
- **Loop order:** MUST be k-i-j, not i-j-k or other permutations

26.2 Input/Output Edge Cases

- Single vertex ($n = 1$): All distances to self are 0
- No edges ($m = 0$): Only $dist[i][i] = 0$, rest unreachable
- Complete graph: All pairs reachable
- Disconnected components: Many -1 answers
- Self-loop with negative weight: Indicates negative cycle
- Query from vertex to itself: Always 0 (unless negative cycle)

26.3 Optimization Tricks

- **Space optimization:** Don't need 3D array for Floyd-Warshall
- **Early termination:** If only need one pair, can stop early
- **Symmetry:** In undirected graphs, compute only half matrix
- **Bitset optimization:** For transitive closure (boolean)
- **Path compression:** Store only necessary path information

27 Exam Problem Templates

27.1 Template 1: Basic Query

```
// Given: n cities, m roads, q queries
// Find: shortest distance for each query

const ll INF = 1e18;
vector<vector<ll>> dist(n, vector<ll>(n, INF));

// Initialize
for (int i = 0; i < n; i++)
    dist[i][i] = 0;

// Read edges
for (int i = 0; i < m; i++) {
    int u, v; ll w;
    cin >> u >> v >> w;
    u--; v--;  // 0-indexed
    dist[u][v] = min(dist[u][v], w);
    dist[v][u] = min(dist[v][u], w); // if undirected
}

// Floyd-Warshall
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; 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]);

// Answer queries
for (int i = 0; i < q; i++) {
    int u, v;
    cin >> u >> v;
    u--; v--;
    if (dist[u][v] == INF)
        cout << -1 << "\n";
    else
        cout << dist[u][v] << "\n";
}
```

27.2 Template 2: Arbitrage Detection

```
// Convert rates to -log for cycle detection
map<string, int> id;
int n; // number of currencies
vector<vector<double>> dist(n, vector<double>(n, INF));

// Initialize
for (int i = 0; i < n; i++)
    dist[i][i] = 0;

// Add exchange rates
dist[u][v] = -log(rate);  // convert multiplication to addition

// Floyd-Warshall
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dist[i][j] = min(dist[i][j],
                             dist[i][k] + dist[k][j]);

// Check for negative cycle (arbitrage)
bool arbitrage = false;
for (int i = 0; i < n; i++) {
    if (dist[i][i] < -1e-9) {  // use epsilon for floating point
        arbitrage = true;
        break;
    }
}

cout << (arbitrage ? "Yes" : "No") << "\n";
```

27.3 Template 3: Path Reconstruction

```
vector<vector<ll>> dist(n, vector<ll>(n, INF));
vector<vector<int>> next(n, vector<int>(n, -1));

// Initialize
for (int i = 0; i < n; i++) {
    dist[i][i] = 0;
    next[i][i] = i;
}

// Add edges
dist[u][v] = w;
next[u][v] = v;

// Floyd-Warshall with path reconstruction
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][k] + dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
                next[i][j] = next[i][k];
            }
        }
    }
}
```

```cpp
// Reconstruct path from u to v
vector<int> getPath(int u, int v) {
    if (next[u][v] == -1) return {};  // no path
    vector<int> path = {u};
    while (u != v) {
        u = next[u][v];
        path.push_back(u);
    }
    return path;
}
```

27.4 Template 4: Graph Properties

```cpp
// After Floyd-Warshall:

// 1. Graph Diameter
ll diameter = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        if (dist[i][j] != INF)
            diameter = max(diameter, dist[i][j]);

// 2. Graph Center (minimize max distance)
int center = -1;
ll minMaxDist = INF;
for (int i = 0; i < n; i++) {
    ll maxDist = 0;
    for (int j = 0; j < n; j++)
        if (dist[i][j] != INF)
            maxDist = max(maxDist, dist[i][j]);
    if (maxDist < minMaxDist) {
        minMaxDist = maxDist;
        center = i;
    }
}

// 3. Eccentricity of each vertex
vector<ll> eccentricity(n);
for (int i = 0; i < n; i++) {
    eccentricity[i] = 0;
    for (int j = 0; j < n; j++)
        if (dist[i][j] != INF)
            eccentricity[i] = max(eccentricity[i], dist[i][j]);
}

// 4. Check for negative cycle
bool hasNegativeCycle = false;
for (int i = 0; i < n; i++) {
    if (dist[i][i] < 0) {
        hasNegativeCycle = true;
        break;
    }
}

// 5. Count reachable pairs
int reachablePairs = 0;
for (int i = 0; i < n; i++)
```

```
    for (int j = i + 1; j < n; j++)
        if (dist[i][j] != INF)
            reachablePairs++;
```

28 Final Checklist Before Exam

28.1 Must Know By Heart

1. Floyd-Warshall basic implementation (k-i-j order!)
2. Initialization: $dist[i][i] = 0$, rest to INF
3. Handling bidirectional edges correctly
4. Negative cycle detection: check diagonal
5. Path reconstruction using next/parent matrix
6. Arbitrage: use $-\log(rate)$ transformation

28.2 Common Variations to Prepare

1. Basic APSP with distance queries
2. Arbitrage / currency exchange
3. Graph diameter / center / radius
4. Negative cycle detection
5. Path reconstruction
6. Minimax path (change min to min-max)
7. Widest path (change min to max-min)
8. Count shortest paths
9. Transitive closure (boolean version)

28.3 Time Complexity Quick Reference

- Floyd-Warshall: $O(n^3)$ time, $O(n^2)$ space
- Works for: $n \leq 450$ typically (500 max)
- Query time: $O(1)$ after preprocessing
- Path reconstruction: $O(n)$ per query
- Johnson's: $O(n^2 \log n + nm)$ better for sparse

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

// Add edges
```

```
dist[u][v] = weight;

// Floyd-Warshall
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dist[i][j] = min(dist[i][j],
                             dist[i][k] + dist[k][j]);
```

28.4 Arbitrage Detection Template

```
// Convert to log
for (edge : edges)
    weight[u][v] = -log(rate);

// Run Floyd-Warshall
// Check for negative cycle
for (int i = 0; i < n; i++)
    if (dist[i][i] < 0)
        return true; // Arbitrage exists
```

29 Complexity Analysis

- **Floyd-Warshall:** $O(n^3)$ time, $O(n^2)$ space
- **Johnson's Algorithm:** $O(n^2 \log n + nm)$ time
- **Repeated Dijkstra:** $O(n^2 \log n + nm)$ time
- **With Path Reconstruction:** Same time, $O(n^2)$ extra space

---

This collection covers fundamental to advanced APSP variations.
Practice these problems to master shortest path algorithms!