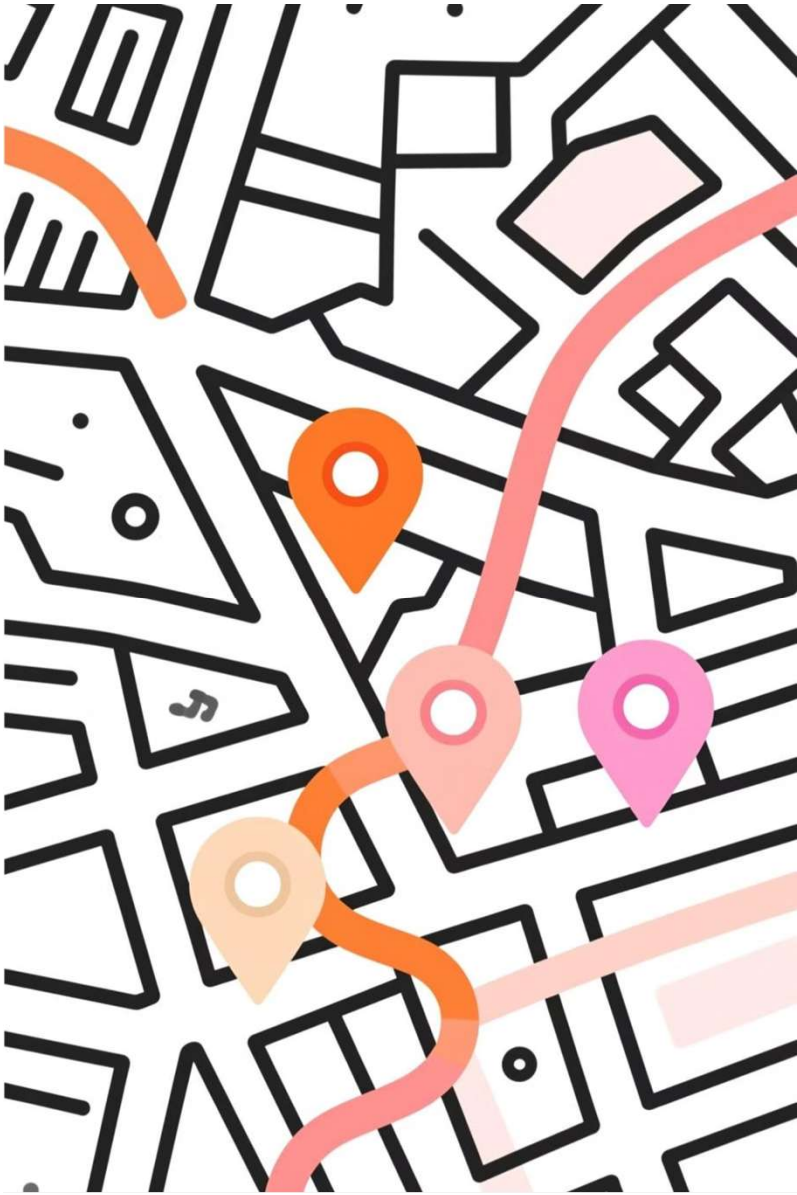
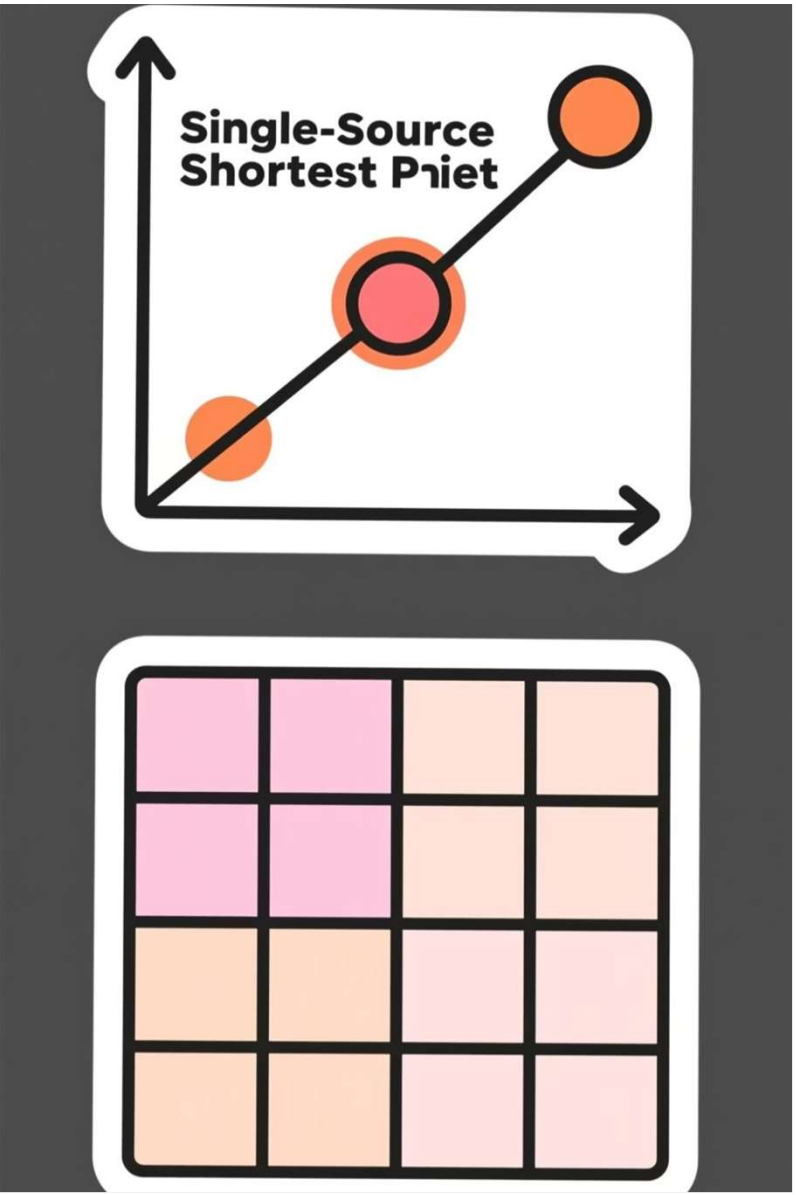
Shortest Paths: Dijkstra vs. Floyd–Warshall

This presentation compares two foundational algorithms for shortest paths in graphs: Dijkstra (single-source shortest paths, SSSP) and Floyd–Warshall (all-pairs shortest paths, APSP). We'll cover core ideas, algorithmic steps, complexity trade-offs, practical examples, and a performance comparison to guide when to use each method.





Problem definitions: SSSP vs. APSP

Single-Source Shortest Path (SSSP)

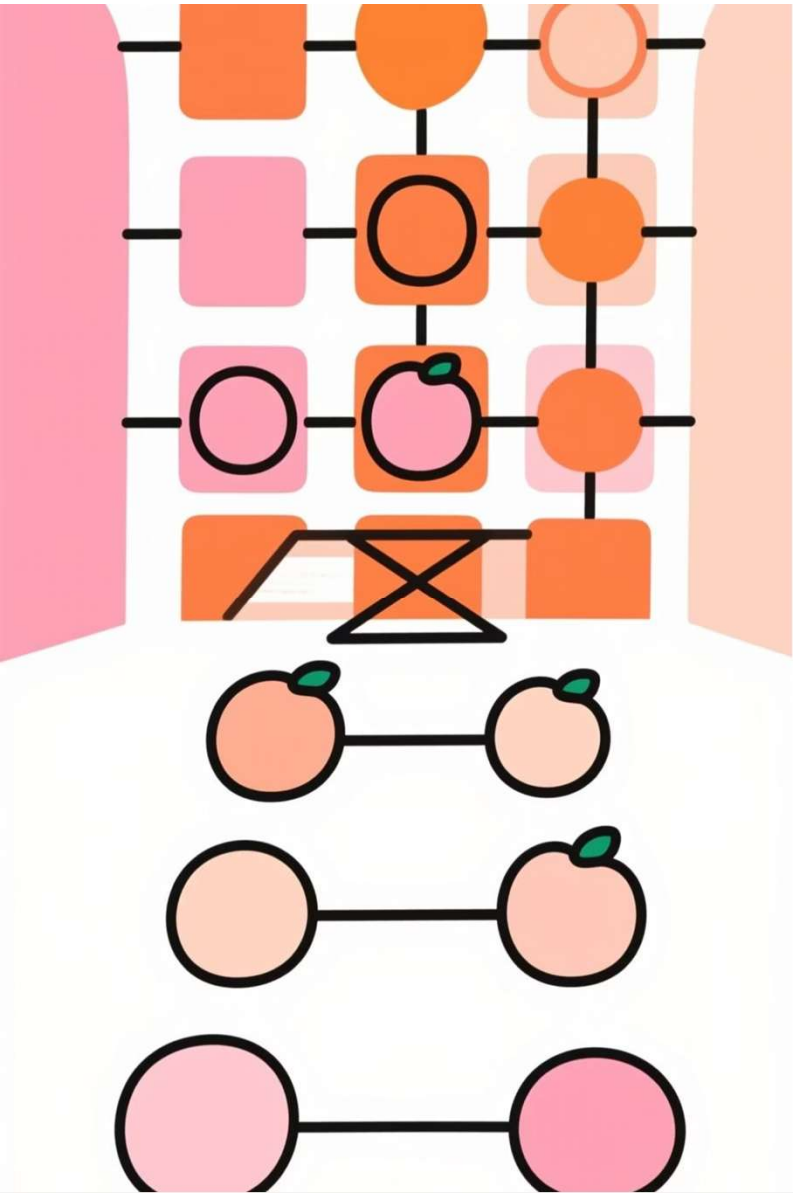
Given a weighted graph G = (V, E) and source s, compute shortest path distances from s to every other node. Typical use: routing from one origin to many destinations.

- One source, many targets
- Often uses Dijkstra (non-negative weights)
- Complexity depends on IVI, IEI and data structure

All-Pairs Shortest Paths (APSP)

Compute shortest path distances between every pair of nodes (u, v) in the graph. Useful when queries are many or when you need global distance information.

- Every source → every destination
- Floyd–Warshall is a classical solution
- Costs scale with IVI^3 for dense approaches



How Dijkstra's Algorithm Works

Dijkstra is a greedy algorithm for SSSP on graphs with non-negative edge weights. Initialize distances (dist[s]=0, others = ∞). Repeatedly extract the vertex with smallest tentative distance from a min-priority queue, then relax its outgoing edges (update neighbor distances if a shorter path is found). Once extracted, a vertex's distance is finalized.

Initialize

Set dist[s]=0, push s into the priority queue.

Extract-min

Pop vertex u with smallest dist[u]; u is now settled.

Relax edges

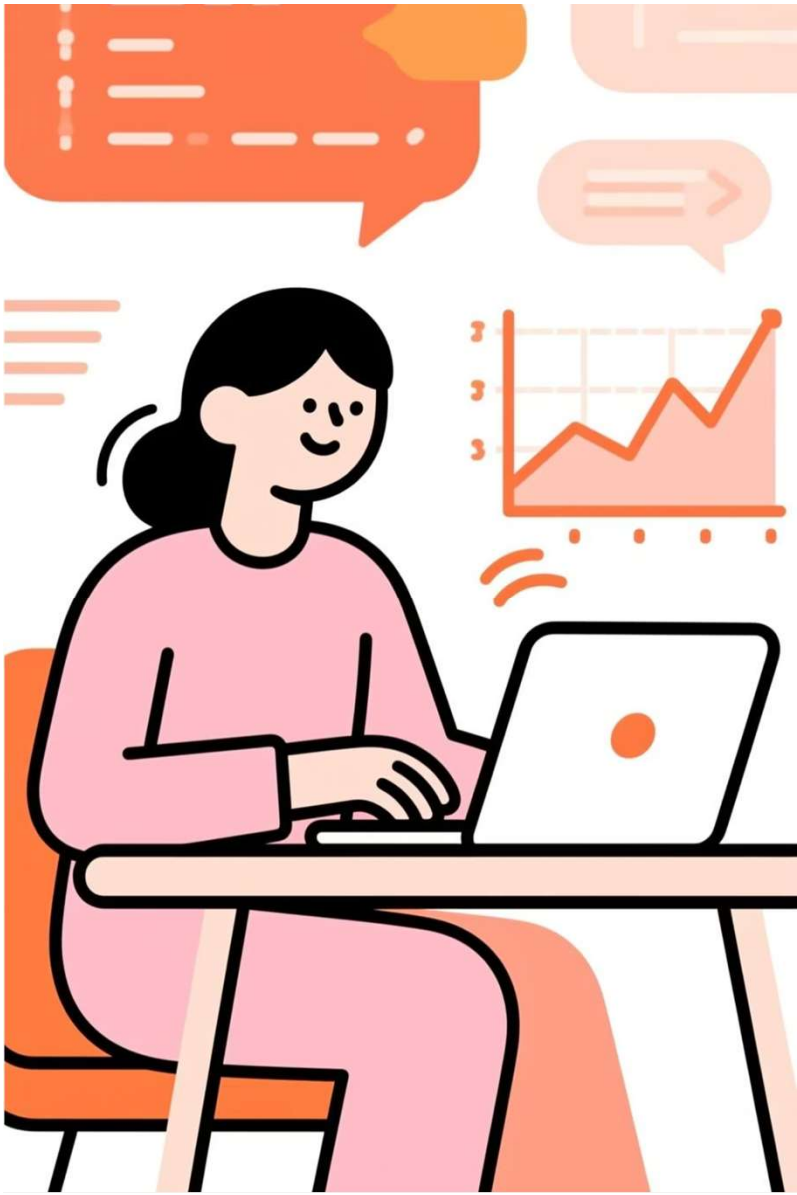
For each (u,v,w): if dist[u]+w < dist[v], update dist[v] and decrease-key.

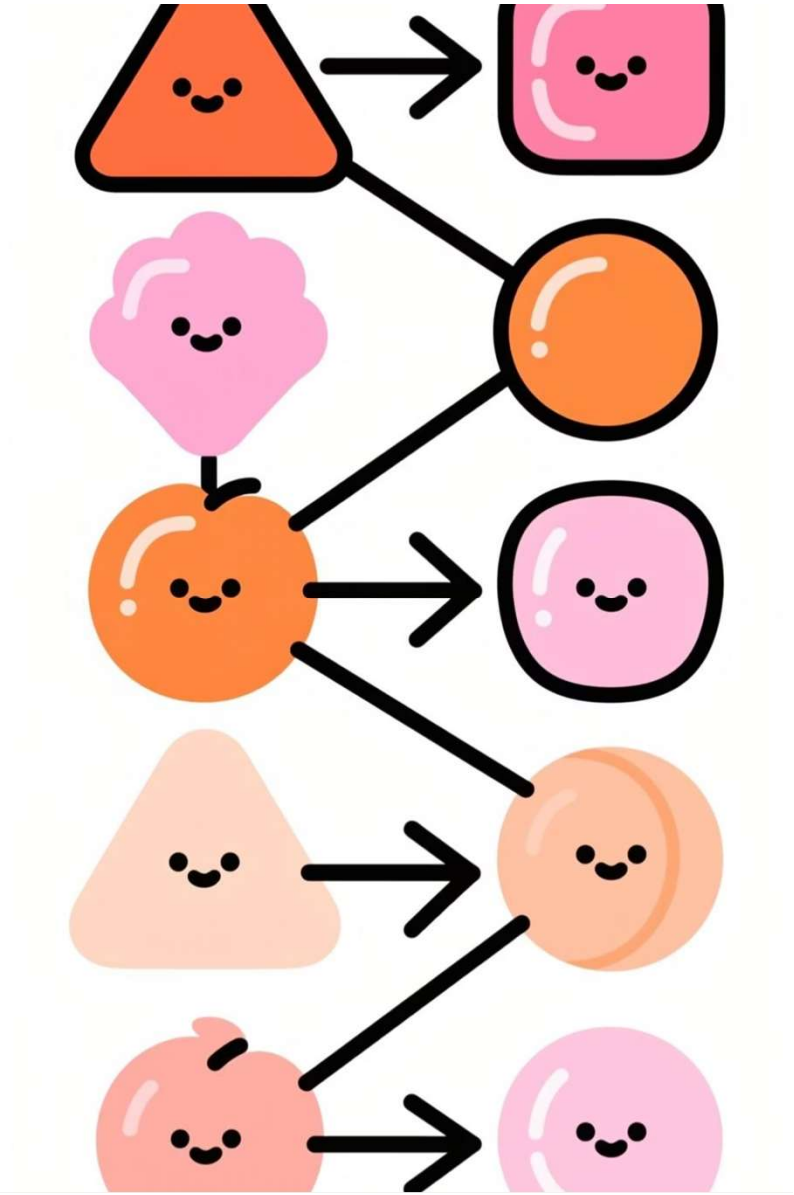
Dijkstra: Complexity & Limitations

Time complexity depends on implementation: using a binary heap: O((|V|+|E|) log |V|) ≈ O(|E| log |V|) for sparse graphs. using a Fibonacci heap: O(|E| + |V| log |V|). Space complexity: O(|V|) for distances and O(|V|+|E|) for graph storage.

- Fast for sparse graphs with single-source queries
- Requires non-negative edge weights
- Repeated runs needed for many different sources

Note: Dijkstra cannot handle negative-weight edges — Bellman–Ford or Johnson's algorithm are alternatives when negative weights exist.





How Floyd–Warshall Works

Floyd–Warshall computes APSP using dynamic programming on an adjacency matrix. Let dist[i][j] be current best distance. Iterate k from 1..n and allow paths that can use only vertices {1..k} as intermediates. Update rule: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]). After n iterations, dist contains shortest distances for all pairs.

Initialize matrix

dist[i][j] = weight(i,j) or ∞ if no edge; dist[i][i]=0.

Iterate k

For each pair (i,j) check if path through k improves dist[i][j].

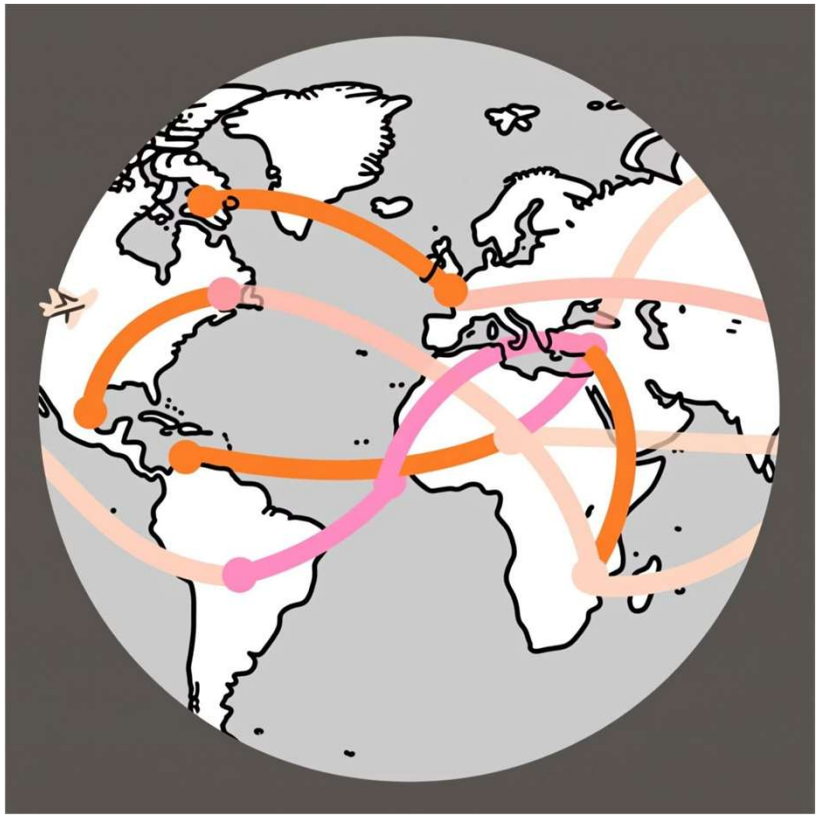
Result

Matrix holds shortest distances for every (i,j).

Floyd–Warshall: Complexity & Benefits

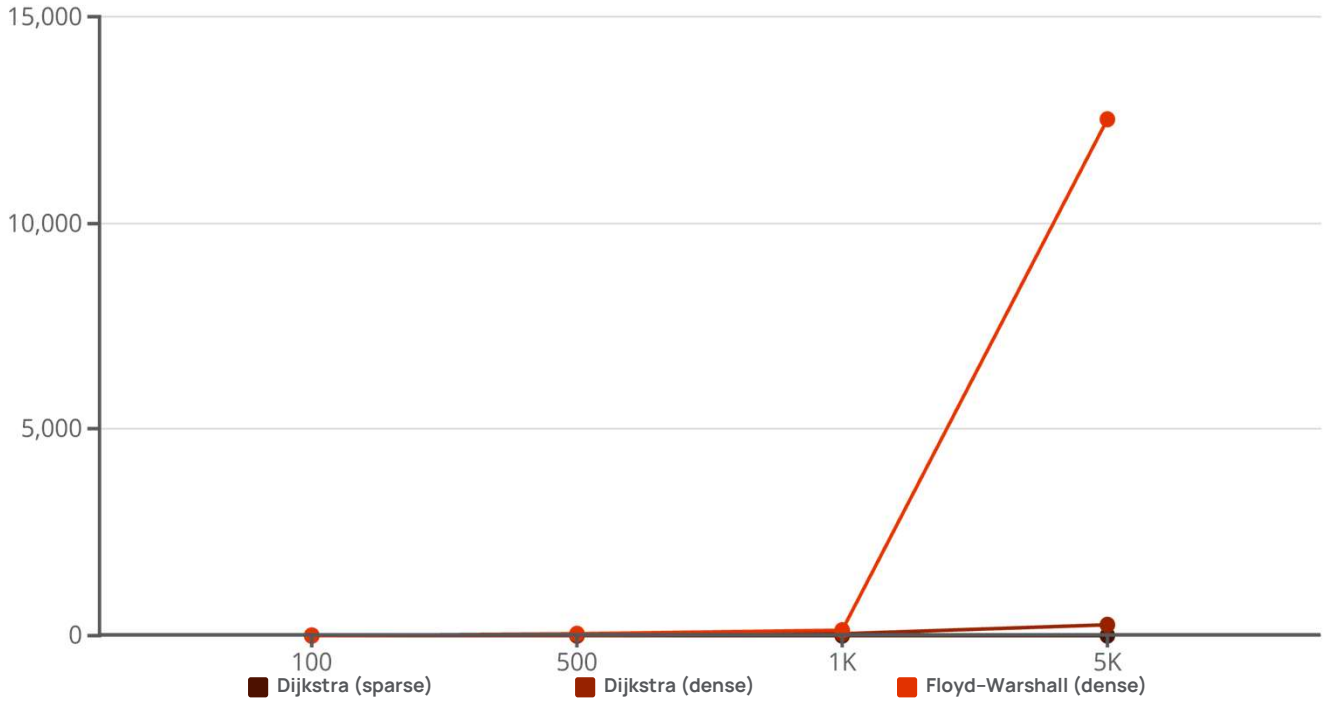
Time complexity: O(|V|^3) regardless of sparsity — predictable but expensive for large graphs. Space complexity: O(|V|^2) to store the distance matrix (and optionally next-hop reconstruction table). Strengths: handles negative edge weights (detects negative cycles), simple matrix-oriented implementation, natural fit for dense graphs and problems needing distances between all pairs.

- APSP computed in one run
- Detects negative cycles via dist[i][i] < 0
- Easy to implement and parallelize (matrix operations)



Performance Comparison: Time vs. Vertices

Comparison setup: synthetic undirected graphs with varying |V| and two regimes: - Sparse: |E| ≈ c|V| (c small) - Dense: |E| ≈ |V|(|V|-1)/2 We'll compare single-run runtimes for Dijkstra (binary heap) and Floyd–Warshall.



Interpretation: Dijkstra scales well for sparse graphs and single-source queries. Floyd–Warshall is competitive only for small-to-moderate vertex counts or when you require APSP and the graph is dense.

Real-world Example: Google Maps & Dijkstra

Google Maps uses variants of SSSP (A*, Dijkstra-like heuristics) to compute fastest route from current location to a destination. Characteristics that make Dijkstra-based approaches suitable:

Single-source queries

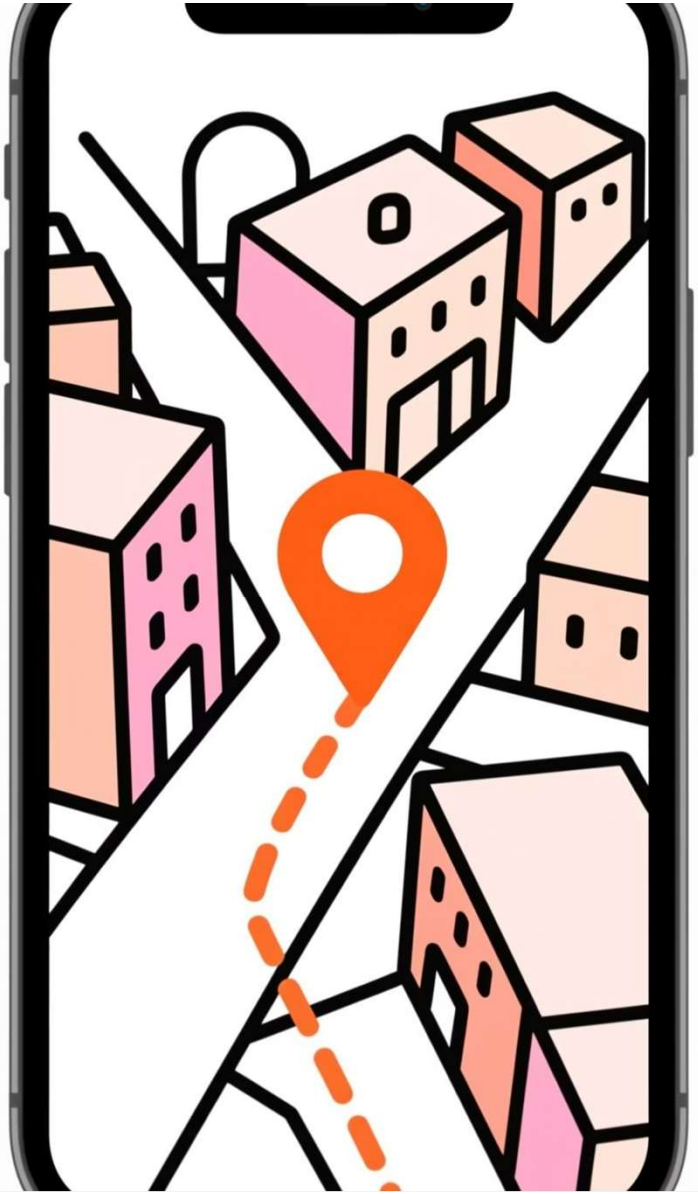
Each navigation request has one origin and one (or a few) destinations.

Sparse road networks

Road graphs are sparse: IEI is O(IVI), making Dijkstra fast with proper heuristics.

Heuristics & constraints

A* adds admissible heuristics (straight-line distance) to speed up search for large maps.

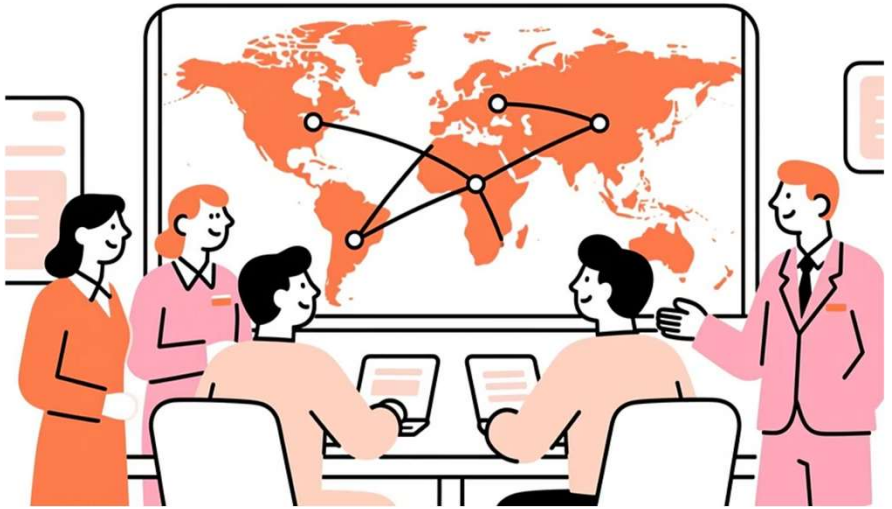


Real-world Example: Airline Route Planning & Floyd–Warshall

Airline route planning often requires global connectivity and frequent APSP queries: e.g., computing minimum hop or minimum-cost connections between any pair of airports, or preparing lookup tables for scheduling and delay impact analysis.

Precomputation advantage

When operations need all-pairs distances readily available, Floyd–Warshall produces a complete distance matrix in one run.



Dense connectivity

Hub-and-spoke networks with dense effective connectivity (many indirect routes) make matrix-based approaches more plausible for moderate IVI.

Choosing Between Dijkstra and Floyd–Warshall

Summary guidance for algorithm selection:

Use Dijkstra (SSSP)

When you have one source (or a few) and a large sparse graph, need fast queries per source, and edges are non-negative. Combine with A* or goal-directed techniques for large spatial graphs.

Use Floyd–Warshall (APSP)

When you need distances between all pairs, graph size is moderate (IVI up to a few thousand depending on resources), or you need negative-weight support and negative-cycle detection.

Hybrid & Practical Tips

For many-sources but not all, consider repeated Dijkstra runs, Johnson's algorithm (reweight + Dijkstra) for sparse graphs with negative edges, or precompute APSP selectively for important nodes (landmarks).

Final takeaway: match the algorithm to the query pattern and graph structure. Use Dijkstra-based approaches for targeted routing at scale; use Floyd–Warshall for complete global distance tables or when negative weights and simplicity matter.