Shortest-path Problems

In Graphs

Definition :

The Shortest Path Problem is a fundamental problem in graph theory.

Given a weighted graph

$G=(V,E,w)$

where

V = set of vertices

E = set of edges

w(e) = weight/cost/length of edge e

The problem is to find the minimum-cost path between two vertices or between all pairs of vertices

Why shortest path?

Because in real-world systems, we want to minimize:

- distance
- time
- cost
- energy
- risk

Mathematical Objective

For vertices sss and ttt, find path:

P=(s=v0,v1,v2,…,vk=t)

that minimizes:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Types Of Shortest Path Problems

1. Single-source shortest path

From one vertex sss to all others.
Algorithms: BFS, Dijkstra, Bellman-Ford.

2. Single-destination shortest path

To one fixed destination.

3. Single-pair shortest path

Between two vertices.

4. All-pairs shortest path

Between every pair.
Algorithm: Floyd–Warshall.

BFS ALGORITHM

(For unweighted graphs or graphs where each edge has equal weight.)

BFS explores vertices in **layers** (levels).

The first time BFS reaches a vertex gives the **shortest distance in terms of number of edges.**

Works because each edge has weight = 1.

BFS ALGORITHM STEPS

- Initialize all distances = ∞
- Set distance[source] = 0
- Put source into a queue
- While queue not empty:
  - take a vertex u from queue
  - visit all neighbors v
    - if v is unvisited:
      - dist[v] = dist[u] + 1
      - parent[v] = u
      - push v into queue
- Continue until all reachable vertices are visited.

Why BFS gives shortest path

Because BFS explores all vertices at distance 1 first, then distance 2, then 3…
It never finds a longer path before exploring all shorter ones.

Dijkstra Algorithem

Dijkstra's Algorithm is a greedy method for finding the shortest paths from a single source to all vertices in a non-negative weighted graph by iteratively picking the nearest unvisited vertex and relaxing its edges.

Dijkstra Algorithem Steps

Steps

1. **Initialize distances to ∞ for all vertices**
2. **dist[source] = 0**
3. **Insert (source, 0) into min-priority queue**
4. **While queue not empty:**
   - **extract vertex u with smallest dist**
   - **for each neighbor v of u: if dist[u] + w(u,v) < dist[v]:**
     - **update dist[v]**
     - **parent[v] = u**
     - **push (v, dist[v]) into queue**
2. **Continue until queue is empty.**

Why Dijkstra works

Because when a vertex is removed from the priority queue, its shortest path is **final** (cannot become smaller later).
This is true only when **weights are non-negative**.

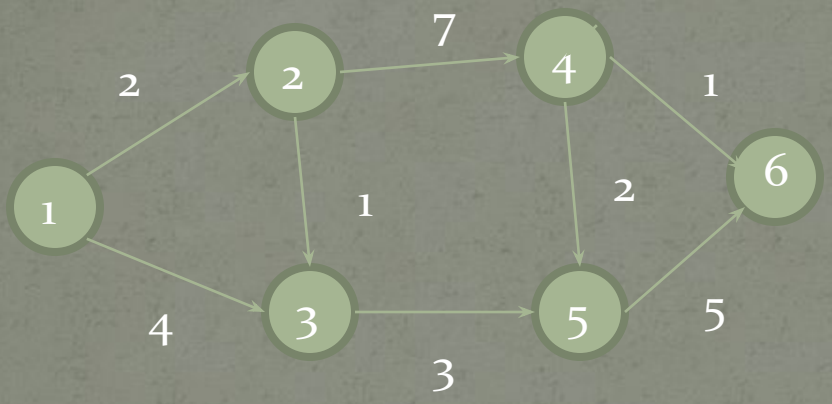
Time Complexity

- With adjacency list + priority queue:

$$O((V+E)\log V)$$

Example: 1

Find the shortest path using Dijkstra algorithem



Sol:

- Step-1:- Initialization (Node 1 is a Source)

  dist[1]=0

  dist[2…6]=∞

  prev[1…6]=null

  visited={} (none yet)

I'll Present Each Iteration as:

- Pick vertex u=unvisited vertex with smallest
- Relax its outgoing edges u-v:
- If dist[u]+w(u,v)<dist[v] then update dist[v] & prev[v]

- Step-2:- **Iteration 1**

  Pick u = 1 (smallest dist = 0).
  Relax outgoing edges from 1:

  Edge 1 → 2 (2): dist[1] + 2 = 0 + 2=2

  dist[2] = 2.

  2 < ∞ → update dist[2] = 2, prev[2] = 1.

  Edge 1 → 3 (4): dist[1] + 4 = 0 + 4 = 4.

  dist[3]=4

  4 < ∞ → update dist[3] = 4, prev[3] = 1.

  Mark 1 visited.

- Step-3:- **Iteration 2**

  Pick u = 2 (smallest unvisited dist = 2).
  Relax outgoing edges from 2:

  - Edge 2 → 3 (1): dist[2] + 1 = 2 + 1 = 3.

    dist[3]=3
    Current dist[3] = 4. Since 3 < 4 → update dist[3] = 3, prev[3] = 2.

  - Edge 2 → 4 (7): dist[2] + 7 = 2 + 7 = 9.

    dist[4]=9
    9 < ∞ → update dist[4] = 9, prev[4] = 2.

  Mark 2 visited.

- Step-4:-**Iteration 3**

  Pick u = 3 (smallest unvisited dist = 3).

  Relax outgoing edges from 3:

  - Edge 3 → 5 (3): dist[3] + 3 = 3 + 3 = 6.
    6 < ∞ → update dist[5] = 6, prev[5] = 3.

No other outgoing from 3. Mark 3 visited.

- Step-5:-Unvisited nodes and dists: 4(9), 5(6), 6(∞).
  Pick u = 5 (smallest = 6).
  Relax outgoing edges from 5:
  - Edge 5 → 4 (2): dist[5] + 2 = 6 + 2 = 8.
    Current dist[4] = 9. Since 8 < 9 → update dist[4] = 8, prev[4] = 5.
  - Edge 5 → 6 (5): dist[5] + 5 = 6 + 5 = 11.
    11 < ∞ → update dist[6] = 11, prev[6] = 5.

Mark 5 visited.

- Step-6:-Unvisited nodes: 4(8), 6(11).
  Pick u = 4 (smallest = 8).
  Relax outgoing edges from 4:
  - Edge 4 → 6 (1): dist[4] + 1 = 8 + 1 = 9.
    Current dist[6] = 11. Since 9 < 11 → update
    dist[6] = 9, prev[6] = 4.

Mark 4 visited.

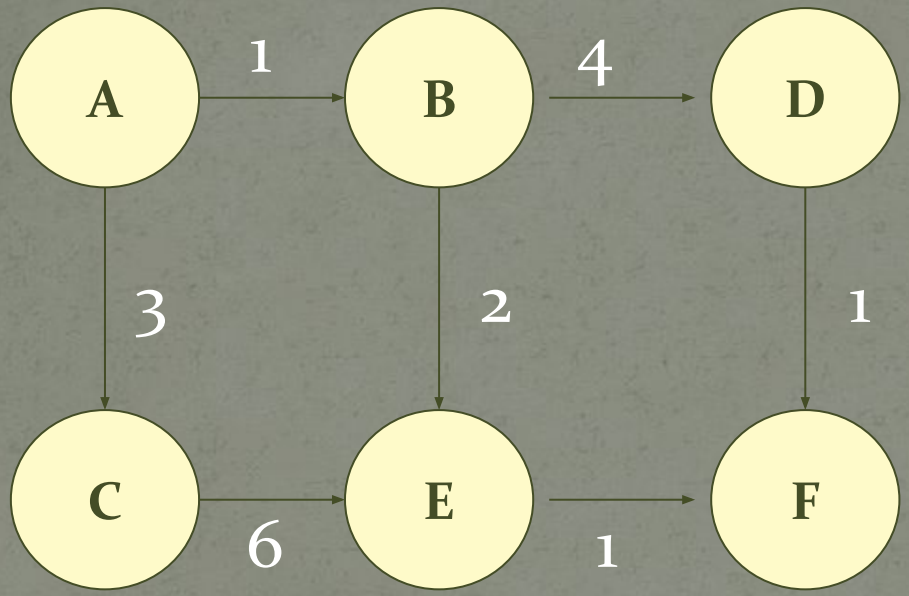
- Step-6:-**Iteration 6**

Only unvisited node: 6 (dist = 9). Pick u = 6. It has no outgoing edges (or no useful ones), so no updates. Mark 6 visited.

Algorithm ends.

	1	2	3	4	5	6
1	0	2	4	∞	∞	∞
2	∞	2	3	9	∞	∞
3	∞	∞	3	9	6	∞
5	∞	∞	∞	∞	6	11
4	∞	∞	∞	8	6	9
6	∞	∞	∞	∞	∞	∞

EX: 2



Picked Node	dist Updates
A(0)	B=1, C=3
B(1)	E=1+2=3
C(3)	E=min(3, 3+6=9)=3
E(3)	D=3+4=7, F=3+1=4
F(4)	Done

Shortest distance to F = 4

Path reconstruction:

A → B → E → F

FLOYD–WARSHALL ALGORITHM

*(All-pairs shortest paths, works with negative weights, but **no negative cycles**.)*

Uses **dynamic programming**.

For each intermediate vertex **k**, update the shortest path between **every pair (i, j)** using k.

Recursive formula

dist[i][j]=min(dist[i][j], dist[i][k]+dist[k][j])

This statement is the heart of the algorithm.

Steps

1. Create an initial matrix dist[i][j]:
   - dist[i][i] = 0
   - dist[i][j] = weight from i→j (or ∞ if no edge)
2. For each vertex k:
   For each vertex i:
   For each vertex j:
   update dist[i][j]
3. After all iterations, dist[i][j] contains shortest distances.

Why Floyd-Warshall works

At iteration k, it considers paths that are allowed to use vertices from the set {1, 2, …, k}.

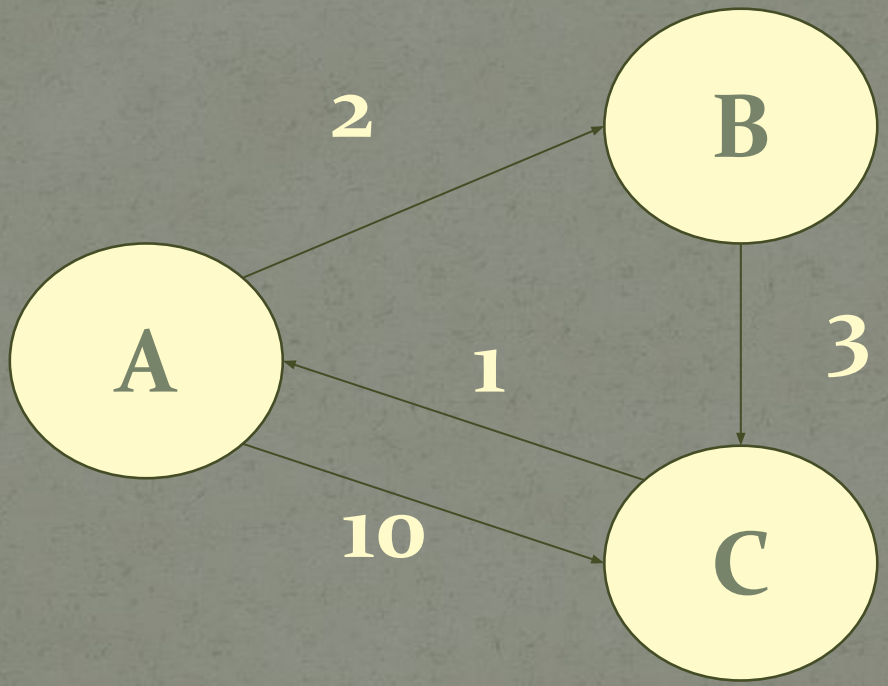
This dynamic programming builds shortest paths in increasing order of allowed intermediate vertices.

Time Complexity

$O(V^3)$

Algorithm	Edge Weights	Negative Weights	Negative Cycles	Type of Problem
BFS	equal/unweighted	not allowed	no	Single-Source
Dijkstra	≥0	not allowed	no	Single-Source
Bellman–Ford	any	allowed	detected	Single-Source
Floyd–Warshall	any	allowed	no(fails)	all-pairs

EX: Floyd-Warshal



	A	B	C
A	0	2	10
B	∞	0	3
C	1	∞	0

Try k = A

No major updates.

Try k = B

A → B → C

2 + 3 = 5 < 10 → update:

A→C = 5

Try k = C

$C \rightarrow A \rightarrow B$

$1 + 2 = 3 < \infty \rightarrow$ update:

$C \rightarrow B = 3$

Final Matrix

	A	B	C
A	0	2	5
B	4	0	3
C	1	3	0