Dijkstra’s Algorithm

Dijkstra’s Algorithm is one of the most famous and widely used algorithms for finding the **shortest path** from a source node to all other nodes in a graph—**as long as all edge weights are non-negative**.

What Is Dijkstra’s Algorithm?

Dijkstra’s Algorithm computes the shortest distance from a **single source** to all other vertices in a weighted graph.

Important: It only works correctly if all weights are **non-negative**.

Dijkstra is extremely efficient and often used in:

- GPS navigation systems
- Network routing protocols
- Game development (pathfinding)
- Robotics motion planning

Why Use Dijkstra?

Choose Dijkstra when:

- The graph has **non-negative weights**
- You need fast shortest-path computation
- Performance matters (Dijkstra is much faster than Bellman–Ford)

Dijkstra is great for real-time systems because it is efficient and predictable.

Key Concepts You Need to Know

1. Weighted Graph

Each edge has a positive cost/weight.

2. Priority Queue (Min-Heap)

Dijkstra uses a **priority queue** to always pick the next closest node.

- This is what makes it efficient.

3. Relaxation

Just like Bellman–Ford, Dijkstra updates distances when it finds a shorter path.

4. Distance Array

Stores the shortest known distance to each node.

How Dijkstra Works (Step-by-Step)

1. Set all distances to infinity, except the source which is **0**.
2. Push the source into a **min-priority queue**.
3. While the queue is not empty:
    - Extract the node with the **smallest distance**.
    - For each neighbor:
        - Check if the path through this node is shorter.
        - If yes, update the distance and push the neighbor into the queue.

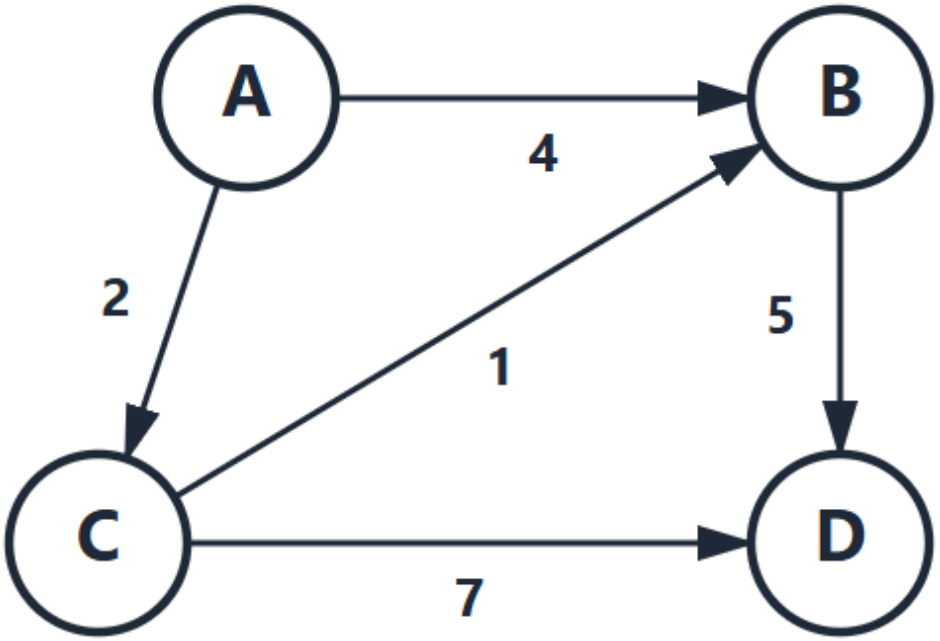
Because weights are non-negative, once a node is processed, its shortest distance is final.

Dijkstra Pseudocode

```
Dijkstra(Graph, source):
   for each vertex v in Graph:
      distance[v] = infinity
   distance[source] = 0

   create a min-priority-queue PQ
   PQ.push((0, source))

   while PQ is not empty:
      (dist, u) = PQ.pop()   // node with smallest distance

      for each neighbor (v, weight) of u:
         if distance[u] + weight < distance[v]:
            distance[v] = distance[u] + weight
            PQ.push((distance[v], v))

   return distance
```

Example (Simple Understanding)

Graph:



Start from **A**:

- Distance[A] = 0
- Relax neighbors → update B and C
- Next take the smallest distance node from the queue (C)
- Relax edges from C → update B and D
- Continue until all nodes processed

Final shortest distances from A:

A = 0
C = 2
B = 3
D = 8

What You Need to Implement Dijkstra

- A graph representation:
    - **Adjacency list** (recommended)
- A **priority queue** (min-heap)
- Distance array
- Source vertex

Time & Space Complexity

Using a Min-Heap

- **Time complexity:** O((V + E) log V)
- **Space complexity:** O(V)

This makes Dijkstra one of the fastest shortest-path algorithms.

When NOT to Use Dijkstra

Do **not** use Dijkstra when:

- The graph has **negative weights** → use Bellman–Ford
- You need all-pairs shortest paths → use Floyd–Warshall

Final Thoughts

Dijkstra's Algorithm is fast, elegant, and extremely practical. It is essential knowledge for anyone studying algorithms, data structures, AI, or pathfinding.

Once you're comfortable with Dijkstra, you may explore:

- A* Search (adds heuristics for faster pathfinding)
- Floyd–Warshall (for all pairs shortest path)
- Johnson's Algorithm (efficient all-pairs for sparse graphs)