

# Dijkstra

---

## ➤ What is Dijkstra?

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.

## ➤ Why Do We 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)

## ➤ Key Points:

### ✓ **Weighted Graph**

Each edge has a positive cost/weight.

### ✓ **Priority Queue (Min-Heap)**

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

### ✓ **Relaxation**

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

### ✓ **Distance Array**

Stores the shortest known distance to each node.

## ➤ How Dijkstra Works

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.

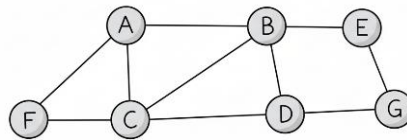
Because weights are non-negative, once a node is processed, its shortest distance is final.

## ➤ Pseudocode:

```
Dijkstra(G, source) {  
  for each vertex u in G {  
    dist[u] = infinity  
    parent[u] = null  
  }  
  dist[source] = 0  
  PQ = priority_queue()  
  PQ.push( (0, source) )  
  while (PQ is not empty) {  
    (du, u) = PQ.pop()
```

```
    for each v in Adj[u] {  
      w = weight(u, v)  
      if (dist[u] + w < dist[v]) {  
        dist[v] = dist[u] + w  
        parent[v] = u  
        PQ.push( (dist[v], v) )  
      }  
    }  
  }  
  for each vertex u in G {  
    print dist[u]  
  }  
}
```

## ➤ Example Graph



### Dijkstra's Algorithm from node A:

Assuming all edges have weight 1

✓ **Initial:**

Distance:  $A=0$ , others= $\infty$

Unvisited: {A, B, C, D, E, F, G}

✓ **Step 1:** Visit A (distance=0)

Update:  $B=1$ ,  $C=1$ ,  $F=1$

Unvisited: {B, C, D, E, F, G}

✓ **Step 2:** Visit B (distance=1)

Update:  $D=2$ ,  $E=2$

Unvisited: {C, D, E, F, G}

✓ **Step 3:** Visit C (distance=1)

Update:  $D=2$  (already),  $F=2$  (already)

Unvisited: {D, E, F, G}

✓ **Step 4:** Visit F (distance=1)

No new updates

Unvisited: {D, E, G}

✓ **Step 5:** Visit D (distance=2)

Update:  $G=3$

Unvisited: {E, G}

✓ **Step 6:** Visit E (distance=2)

Update:  $G=3$  (already)

Unvisited: {G}

✓ **Step 7:** Visit G (distance=3)

### Shortest distances from A:

$A \rightarrow A: 0$

$A \rightarrow B: 1$

$A \rightarrow C: 1$

$A \rightarrow F: 1$

$A \rightarrow D: 2$

$A \rightarrow E: 2$

$A \rightarrow G: 3$

## ➤ 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.

## ➤ What You Need to Implement Dijkstra

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

## ➤ 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, useful, and essential for learning pathfinding and graph algorithms. After mastering it, you can explore:

- **A\* Search** — faster pathfinding with heuristics
- **Floyd–Warshall** — all-pairs shortest paths
- **Johnson's Algorithm** — efficient all-pairs for sparse graphs