Dijkstra's Algorithm

Authors:

Sultan Al Nuaimi

&

Abdulla Al Nuaimi

April 1, 2026

Contents

1 Understanding Dijkstra's Algorithm **1**

2 Python Implementation **2**

3 Visual Logic and Constraints **3**

1 Understanding Dijkstra's Algorithm

Dijkstra's algorithm is a **pathfinding algorithm** used to determine the shortest path between nodes in a graph. It is uniquely **greedy**, meaning at each step, it chooses the path that looks best at that moment, eventually building the globally shortest path.

Relaxation

The core mathematical operation in Dijkstra's is Relaxation. Suppose we have an edge from node u to node v with a weight $w(u, v)$. We maintain a distance array $dist[]$ where $dist[v]$ is the current shortest known distance to v.

If we find that going through u provides a shorter path to v than what we previously recorded, we update $dist[v]$:

$$\text{If } dist[u] + w(u, v) < dist[v], \text{ then } dist[v] = dist[u] + w(u, v) \tag{1}$$

Dijkstra's Optimality

The algorithm maintains a set of visited nodes. The invariant is that for every visited node u, $dist[u]$ is the **absolute shortest path**. This holds true as long as all weights are non-negative (≥ 0), ensuring that no future divergence could possibly reduce the path length once a node is finalized.

Steps of the Algorithm

1. **Initialization:** Set the distance to the source node to **0** and all other nodes to **infinity** (∞).
2. **Selection:** Pick the unvisited node with the smallest current distance.
3. **Exploration:** For the current node, calculate the distance to all its unvisited neighbors.
4. **Relaxation:** Update the neighbor's distance if the new calculated path is shorter than the previously known distance.
5. **Finalization:** Mark the current node as visited. A visited node will never be checked again.
6. **Termination:** Repeat until all nodes are visited or the target is reached.

2 Python Implementation

The following implementation utilizes the heapq library. Using a Min-Heap ensures that extracting the node with the minimum distance takes $O(\log V)$ time.

```python
import heapq

def dijkstra(graph, start):
    # Initialize distances with infinity
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Priority Queue: (distance, current node)
    pq = [(0, start)]

    while pq:
        current_dist, u = heapq.heappop(pq)

        # Skip if we found a better path already
        if current_dist > distances[u]:
            continue

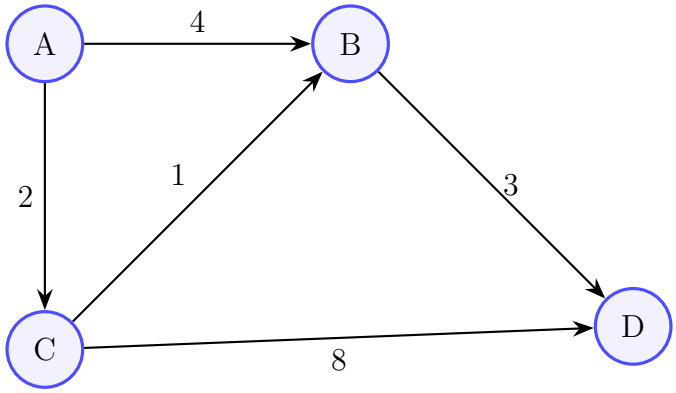
        for v, weight in graph[u]:
            distance = current_dist + weight

            # Relaxation
            if distance < distances[v]:
                distances[v] = distance
                heapq.heappush(pq, (distance, v))

    return distances
```

3 Visual Logic and Constraints

Dijkstra's processes nodes in a **radial fashion**, prioritized by cumulative weight.



Negative Weight Constraint

It is critical to note that Dijkstra **fails** on graphs with negative edges. If a negative edge exists, a node that was previously marked "visited" could have its distance reduced later, breaking the greedy logic. For such cases, use **Bellman-Ford's Algo**.

Complexities

Implementation	**Complexity**
Priority Queue (Binary Heap)	$O((E+V)\log V)$
Unordered List (Brute Force)	$O(V^2)$
Fibonacci Heap (Theoretical)	$O(E+V\log V)$

Conclusion: For Python devs, using **heapq** with an **adjacency list** is the common approach for optimal pathfinding.