MTech CSE – 1st Semester
Student ID: **A125023**
Student Name: **SRIJITA VERMA**

I I I T
BHUBANESWAR
I I I T Bhubaneswar
Imagine, Innovate, Inspire
(A University established by Government of Odisha)

Question

Explain why Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

Answer

Short Answer (Core Reason)

Dijkstra's algorithm assumes that once a vertex is selected as having the minimum tentative distance, that distance is final. This assumption breaks down in the presence of negative edge weights, because a shorter path to an already processed vertex may be discovered later through a negative-weight edge.

Background: How Dijkstra's Algorithm Works

Dijkstra's algorithm maintains:

- a set of vertices whose shortest-path distances from the source are finalized,
- a tentative distance estimate for each remaining vertex.

At each step:

- the vertex with the smallest tentative distance is selected,
- its distance is permanently fixed,
- its outgoing edges are relaxed.

This strategy relies critically on a key assumption.

Key Assumption of Dijkstra's Algorithm

All edge weights must be non-negative.

With non-negative edges, any path that goes through additional edges can never reduce the distance to an already chosen vertex. Hence, once the minimum-distance vertex is selected, its shortest path is guaranteed to be correct.

Why Negative Edge Weights Break This Assumption

When negative edge weights are present:

- a vertex that appears to have the shortest distance at one step
- may later receive a shorter path via a negative-weight edge.

This violates the greedy choice made by Dijkstra's algorithm.

Formal Proof-Style Explanation

Dijkstra's algorithm is based on the following invariant:

When a vertex u is extracted from the priority queue with minimum tentative distance, the current distance assigned to u is the true shortest path distance from the source.

This invariant holds only when all edge weights are non-negative.

Proof of Failure with Negative Edge Weights

Assume, for contradiction, that Dijkstra's algorithm is correct even when negative edge weights are allowed.

Let u be a vertex whose distance is finalized at some step of the algorithm, with tentative distance $d(u)$. At this point, Dijkstra's algorithm assumes that no shorter path to u exists.

However, suppose there exists another vertex v that has not yet been processed, and an edge (v, u) with negative weight $w(v, u) < 0$. It is possible that:

$$d(v) + w(v, u) < d(u).$$

Since v has not yet been selected, this shorter path to u is discovered only after u has already been finalized. This contradicts the invariant that the distance to u was final and minimal.

Therefore, the greedy choice made by Dijkstra's algorithm is invalid in the presence of negative edge weights. Hence, Dijkstra's algorithm cannot correctly compute shortest paths when negative edges are present.

Illustrative Example

Consider the graph:

- $s \to a$ with weight 2,
- $s \to b$ with weight 5,
- $b \to a$ with weight -4.

Step-by-Step Execution

Initialize distances:

$$\text{dist}(s) = 0, \quad \text{dist}(a) = 2, \quad \text{dist}(b) = 5.$$

Dijkstra's algorithm selects vertex a (distance 2) and finalizes it.

Later, when vertex b is processed, a new path is found:

$$s \to b \to a,$$

with total cost:

$$5 + (-4) = 1.$$

This path is shorter than the previously finalized distance of 2. However, Dijkstra's algorithm does not reconsider finalized vertices and therefore returns an incorrect result.

Fundamental Reason for Failure

This failure is not an implementation issue, but a theoretical limitation. Dijkstra's algorithm relies on the monotonicity of path costs, which holds only when all edge weights are non-negative. Negative weights destroy this property.

Relation to the Greedy Strategy

Dijkstra's algorithm is a greedy algorithm. Greedy algorithms require that the greedy choice be globally optimal.

With negative edge weights:

- a locally optimal choice (smallest current distance)
- may lead to a globally suboptimal result.

Thus, the greedy strategy is invalid in this setting.

What About Negative Cycles?

If a graph contains a negative-weight cycle, no shortest path exists, since distances can be reduced indefinitely. Dijkstra's algorithm cannot detect negative cycles and produces meaningless results in such cases.

Correct Algorithms for Graphs with Negative Weights

When negative edge weights are present, appropriate alternatives include:

- **Bellman–Ford algorithm**: handles negative weights, detects negative cycles, and runs in $O(VE)$ time.
- **Johnson's algorithm**: reweights edges to remove negativity and then uses Dijkstra's algorithm as a subroutine.

Comparison: Dijkstra vs Bellman–Ford

The fundamental difference between Dijkstra's algorithm and the Bellman–Ford algorithm lies in how they handle edge weights and path relaxation.

Dijkstra's Algorithm

- Uses a greedy strategy.
- Assumes all edge weights are non-negative.
- Finalizes distances permanently when a vertex is selected.
- Runs in $O((V + E) \log V)$ time with a priority queue.
- Fails in the presence of negative edge weights.

Bellman–Ford Algorithm

- Uses dynamic programming and repeated relaxation.
- Allows negative edge weights.
- Does not assume distances are final until all relaxations are complete.
- Detects negative-weight cycles.
- Runs in $O(VE)$ time.

Key Distinction

Dijkstra's algorithm relies on the monotonic increase of path costs, which is violated by negative edges. Bellman–Ford avoids this assumption by relaxing all edges multiple times, ensuring correctness even when edge weights are negative.

Final Conclusion

Dijkstra's algorithm cannot be applied to graphs with negative edge weights because:

- it assumes finalized distances never decrease,
- negative edges can produce shorter paths later,
- the greedy choice property no longer holds.

Practical Systems Insight. In real-world routing protocols, negative edge weights are avoided specifically to enable efficient shortest-path algorithms such as Dijkstra's.

Intuition

Dijkstra's algorithm assumes that distances only increase as paths grow. Negative edges allow distances to decrease, invalidating earlier decisions and making the algorithm unreliable.