Problem Analysis(Bellman Ford)

1. Definition

The **Bellman–Ford algorithm** finds the **shortest paths** from a single source node to all other nodes in a **weighted graph**, even if some edges have **negative weights**.

- Unlike Dijkstra, it can handle **negative edge weights**, but it **cannot handle negative weight cycles** (it detects them).
- Useful for graphs where edge weights can be negative.

2. Solution Approach (How Bellman–Ford Works)

1. **Initialization**
   - Create a distance array dist[] of size n for all vertices.
   - Set dist[source] = 0 and dist[v] = INF for all other vertices.
2. **Relaxation Process**
   - Repeat for n-1 iterations (where n is the number of vertices):
     - For each edge (u, v) with weight w:
       - If dist[u] + w < dist[v] then update:
       - dist[v] = dist[u] + w
   - After n-1 iterations, all shortest paths **without negative cycles** are finalized.
3. **Negative Cycle Detection**
   - Check all edges (u, v) with weight w:
     - If dist[u] + w < dist[v], a **negative weight cycle** exists.
   - Bellman-Ford reports this because distances can be further decreased beyond n-1 iterations.

3. Pseudocode (C++ Style)

```
struct Edge {
    int u, v, w;
};

bool BellmanFord(int n, int source, vector<Edge>& edges, vector<long long>& dist) {
    dist.assign(n, 1e18);
    dist[source] = 0;

    for (int i = 1; i <= n-1; i++) {
        for (auto edge : edges) {
            int u = edge.u, v = edge.v, w = edge.w;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
            }
        }
    }
    for (auto edge : edges) {
```

```
        int u = edge.u, v = edge.v, w = edge.w;
        if (dist[u] + w < dist[v]) {
            return false; // Negative cycle exists
        }
    }

    return true;
}
```

4. Time Complexity

- **O(n × m)**, where:
  - n = number of vertices
  - m = number of edges
- Reason: Each edge is relaxed for n-1 iterations.
- Negative cycle check takes **O(m)**.

Problem: Warm Holes

1. Problem Statement

- You are given a **directed graph** representing star systems connected by **wormholes**.
- Each wormhole is one-way and changes time by t years:
  - t > 0 → future
  - t < 0 → past
- Travel through a wormhole is **instantaneous**.
- You start from **star system 0**.
- Every star system is guaranteed to be reachable from system 0.

Goal: Determine whether there exists a **cycle reachable from system 0** with a **negative total time change**.

- If such a cycle exists → "possible" (you could theoretically reach the Big Bang).
- Otherwise → "not possible".

2. Hint

- Treat each wormhole as a **directed edge** with weight t (time shift).
- The problem reduces to **detecting a negative-weight cycle** reachable from node 0.
- The **Bellman–Ford algorithm** is specifically designed to detect such negative cycles.

3. Solution Approach

1. **Graph Representation**
    - Each wormhole → directed edge (u → v) with weight t.
2. **Initialization**
    - Distance array dist[] of size N.
    - dist[0] = 0, all other nodes initialized to INF.
3. **Edge Relaxation**
    - Run **N − 1 iterations**:
        - For each edge (u, v, w), check:
        - if dist[u] + w < dist[v]:
        -     dist[v] = dist[u] + w
    - Optional optimization: stop early if no distances are updated in an iteration.
4. **Negative Cycle Detection**
    - On the Nth iteration, if any edge can still be relaxed → **negative cycle exists**.
5. **Output**
    - Negative cycle detected → "possible"
    - Otherwise → "not possible"

4. Pseudocode

```
function WarmHoles(N, edges):
  // edges: list of (u, v, t)

  dist[0..N-1] = INF
  dist[0] = 0

  // Relax edges N-1 times
  for i = 1 to N-1:
    updated = false
    for each edge (u, v, w) in edges:
      if dist[u] + w < dist[v]:
        dist[v] = dist[u] + w
        updated = true
    if not updated:
      break   // Early stopping optimization

  // Check for negative cycle on Nth iteration
  for each edge (u, v, w) in edges:
    if dist[u] + w < dist[v]:
      return "possible"

  return "not possible"
```

5. Time Complexity Analysis

1. **Initialization:** O(N) to set all distances.
2. **Relaxation Passes:**
    - Outer loop: $N - 1$ iterations
    - Inner loop: M edges per iteration
    - Total: $O((N - 1) \times M) = O(N \times M)$
3. **Final Negative Cycle Check:** O(M)
4. **Overall Time Complexity:** $O(N \times M)$

Implementation Link:

algorithm-/bellmanford/warmhole/warmhole.cpp at main · Jannat651/algorithm-