Dijkstra's Algorithm

What is Dijkstra's Algorithm?

It is a popular algorithm for solving single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph.

Problem Statement

Given a graph $G=(V,E)$, where:

- V is the set of vertices,
- E is the set of edges with non-negative weights $w(u,v) \geq 0$

The goal is to compute the minimum distance from a given source node $s \in V$ to every other node $v \in V$.

Working Principle

Dijkstra's Algorithm is based on the **greedy approach**. At each step, it selects the vertex with the **smallest tentative distance** that has not yet been visited and updates the distances of its neighboring vertices.

Algorithm Steps

1. Initialization:

- Set the distance of the **source node** to 0.
- Set the distance of **all other nodes** to infinity.
- Maintain a **priority queue (or min-heap)** to efficiently fetch the next closest unvisited node.

2. Processing Nodes:

- While the priority queue is not empty:
  - Extract the node u with the minimum tentative distance.

- For each unvisited neighbor v of u, compute the new tentative distance:
  `New_distance = distance[u]+ weight(u,v)`
- If the new distance is smaller than the current known distance, update it.

3. Termination:

- Repeat the process until all nodes are visited or the queue is empty.

Example

Consider the following graph:

```
   A
  / \
1/   \4
B     C
 \   /
 2\ /1
   D
```

Steps:

- Start from A: distances → A=0, B=∞, C=∞, D=∞
- Visit A: update B=1, C=4
- Visit B: update D = 1 + 2 = 3
- Visit D: update C = min(4, 3+1) = 4
- Visit C: done

Final shortest distances from A:

- A = 0
- B = 1
- D = 3
- C = 4

Code

Python

```
def dijkstra(graph, start):
    # Initialize distances and visited set
    visited = set()
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
```

```
    while len(visited) < len(graph):
        # Select the unvisited node with the smallest distance
        current_node = None
        for node in graph:
            if node not in visited:
                if current_node is None or distances[node] <
distances[current_node]:
                    current_node = node

        # Mark the node as visited
        visited.add(current_node)

        # Update distances to neighbors
        for neighbor, weight in graph[current_node]:
            if distances[current_node] + weight <
distances[neighbor]:
                distances[neighbor] = distances[current_node] +
weight

    return distances
```

C++

```
#include <iostream>
#include <limits>

using namespace std;

const int INF = 1e9; // A large number representing infinity
const int V = 4;     // Number of vertices

void dijkstra(int graph[V][V], int start) {
    int dist[V];
    bool visited[V];

    // Step 1: Initialize distances and visited array
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        visited[i] = false;
    }
    dist[start] = 0;

    // Step 2: Find shortest path for all vertices
    for (int i = 0; i < V - 1; i++) {
        int minDist = INF, u;

        // Find the unvisited node with the smallest distance
```

```
        for (int j = 0; j < V; j++) {
            if (!visited[j] && dist[j] < minDist) {
                minDist = dist[j];
                u = j;
            }
        }

        visited[u] = true;

        // Update distances of neighbors of u
        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] && dist[u] +
graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // Print shortest distances
    for (int i = 0; i < V; i++) {
        cout << "Distance from " << start << " to " << i << " is "
<< dist[i] << endl;
    }
}
```

Javascript

```
function dijkstra(graph, start) {
  const distances = {};
  const visited = {};

  // Initialize distances
  for (let node in graph) {
    distances[node] = Infinity;
  }
  distances[start] = 0;

  while (Object.keys(visited).length < Object.keys(graph).length)
{
    // Find the unvisited node with the smallest distance
    let closestNode = null;
    for (let node in distances) {
      if (!visited[node]) {
        if (closestNode === null || distances[node] <
distances[closestNode]) {
          closestNode = node;
        }
      }
```

```
    }

    visited[closestNode] = true;

    // Update distances to neighbors
    for (let neighbor of graph[closestNode]) {
      let [nextNode, weight] = neighbor;
      let newDist = distances[closestNode] + weight;
      if (newDist < distances[nextNode]) {
        distances[nextNode] = newDist;
      }
    }
  }

  return distances;
}
```

Time Complexity

Data Structure	Time Complexity	When Used
Array	O(V^2)	Dense graphs
Min-Heap (Binary Heap)	O((V+E)logV)	Sparse graphs

Advantages

- Efficient and accurate for graphs with non-negative weights
- Simple to implement
- Optimized with data structures like heaps and Fibonacci heaps

Limitations

- **Does not support negative edge weights**
- Not the most optimal for very large graphs (in which A* or Bidirectional Dijkstra might be better)
- Needs the whole graph in memory

Bellman ford algorithm

What is it used for?

The **Bellman-Ford algorithm** is a **shortest path algorithm** used to find the **shortest distances from a single source vertex to all other vertices** in a **weighted graph**. Unlike Dijkstra's algorithm, Bellman-Ford **can handle negative weight edges**, which makes it more versatile.

The main idea of the Bellman-Ford algorithm is **relaxation**. It tries to **improve the shortest path** estimates by iteratively updating the distances between connected vertices.

Relaxation Process

If there is an edge from vertex `u` to `v` with weight `w`, and if the distance to `v` can be minimized by going through `u`, then we update the distance to `v`.

```
if dist[u] + w < dist[v]: dist[v] = dist[u] + w
```

This process is repeated **V-1 times**, where `V` is the number of vertices.

In the end, we do **one more iteration** over all edges to check for **negative weight cycles**. If we can still relax any edge, then a negative cycle exists.

Time Complexity

`O(V * E)`
– because we relax all `E` edges `V-1` times.

Space Complexity

`O(V)`
– for the `dist[]` array storing distances from the source.

Code

Python

```
def bellman_ford(graph, V, E, source):
    dist = [float('inf')] * V
    dist[source] = 0

    # Relax all edges V-1 times
    for _ in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Check for negative weight cycle
    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("Graph contains a negative weight cycle")
            return

    print("Vertex Distance from Source")
    for i in range(V):
        print(f"{i}\t{dist[i]}")
```

C++

```
#include <iostream>
using namespace std;

const int MAX = 100;
const int INF = 1e9;

int main() {
    int V, E;
    cin >> V >> E; // number of vertices and edges

    int edges[MAX][3]; // each edge has (u, v, w)

    for (int i = 0; i < E; i++) {
        cin >> edges[i][0] >> edges[i][1] >> edges[i][2];
    }

    int dist[MAX];
    for (int i = 0; i < V; i++) dist[i] = INF;

    int src;
```

```
    cin >> src;
    dist[src] = 0;

    // Relax all edges V-1 times
    for (int i = 0; i < V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j][0];
            int v = edges[j][1];
            int w = edges[j][2];

            if (dist[u] != INF && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
            }
        }
    }

    // Check for negative weight cycle
    for (int j = 0; j < E; j++) {
        int u = edges[j][0];
        int v = edges[j][1];
        int w = edges[j][2];

        if (dist[u] != INF && dist[u] + w < dist[v]) {
            cout << "Negative weight cycle detected\n";
            return 0;
        }
    }

    // Print distances
    for (int i = 0; i < V; i++) {
        if (dist[i] == INF)
            cout << "INF ";
        else
            cout << dist[i] << " ";
    }

    return 0;
}
```

Javascript

```
function bellmanFord(V, edges, source) {
    let dist = Array(V).fill(Infinity);
    dist[source] = 0;

    // Relax all edges V-1 times
    for (let i = 0; i < V - 1; i++) {
```

```
        for (let [u, v, w] of edges) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
            }
        }
    }

    // Check for negative weight cycle
    for (let [u, v, w] of edges) {
        if (dist[u] + w < dist[v]) {
            console.log("Negative weight cycle detected");
            return;
        }
    }

    // Print distances
    console.log("Shortest distances from source " + source + ":");
    for (let i = 0; i < V; i++) {
        console.log("Vertex", i, "Distance:", dist[i]);
    }
}

// Example usage:
const V = 5;
const edges = [
    [0, 1, 6], [0, 2, 7],
    [1, 2, 8], [1, 3, 5], [1, 4, -4],
    [2, 3, -3], [2, 4, 9],
    [3, 1, -2],
    [4, 0, 2], [4, 3, 7]
];

bellmanFord(V, edges, 0);
```

Floyd warshall algorithm

Introduction

The **Floyd-Warshall algorithm** is a classical dynamic programming algorithm used to **find the shortest paths between all pairs of vertices** in a weighted graph. Unlike Dijkstra's or Bellman-Ford algorithms that find the shortest path from a single source, Floyd-Warshall provides a full-pairwise shortest path matrix.

Problem Statement

Given a **directed weighted graph** G=(V,E), where:

- V is the set of vertices (|V| = n),
- E is the set of edges with weights (can be negative but no negative cycles),

Find the **shortest path between every pair of vertices** (i,j) such that the sum of weights along the path from i to j is minimized.

Assumptions

- The graph can have negative weights.
- The graph **should not** contain **negative weight cycles** (if it does, the algorithm can detect them).
- Self-loops (i.e., i→i) are assumed to be 0.

Main Idea

The Floyd-Warshall algorithm uses **Dynamic Programming**. The idea is:

Let:

- D^(k)[i][j] be the shortest distance from vertex i to vertex j using only vertices from the set {1,2,...,k} as intermediate vertices.

Recursive relation:

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j],\ D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

This means:

- Either the shortest path from i to j **does not** go through vertex k, or
- It **does** go through k, and we sum the shortest paths from i→k and k→j.

Algorithm Steps

1. **Initialize** the distance matrix `dist[i][j]` as follows:
    - `dist[i][j] = 0` if i=j

    ○ `dist[i][j] = weight(i, j)` if edge (i,j) exists
    ○ `dist[i][j] = ∞` if $(i,j) \notin E$
2. **Iteratively update** all distances using the recursive relation.

Time and Space Complexity

Aspect	Complexity
Time	O(V^3)
Space	O(V^2)

Code

Python

```
INF = 99999
V = 4

# Graph as adjacency matrix
graph = [
    [0,      3,      INF,    5],
    [2,      0,      INF,    4],
    [INF,    1,      0,      INF],
    [INF,    INF,    2,      0]
]

# Floyd-Warshall algorithm
for k in range(V):
    for i in range(V):
        for j in range(V):
            graph[i][j] = min(graph[i][j], graph[i][k] +
graph[k][j])

# Print the result
for row in graph:
    for val in row:
        print("INF" if val == INF else val, end="\t")
    print()
```

C++

```
#include <iostream>
using namespace std;

#define V 4
#define INF 99999

int main() {
    int graph[V][V] = {
        {0,   3,   INF, 5},
        {2,   0,   INF, 4},
        {INF, 1,   0,   INF},
        {INF, INF, 2,   0}
    };

    // Floyd-Warshall algorithm
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (graph[i][k] + graph[k][j] < graph[i][j])
                    graph[i][j] = graph[i][k] + graph[k][j];
            }
        }
    }

    // Print result
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (graph[i][j] == INF)
                cout << "INF\t";
            else
                cout << graph[i][j] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

Javascript

```
const V = 4;
const INF = 99999;
```

```
let graph = [
  [0,     3,     INF,   5],
  [2,     0,     INF,   4],
  [INF,   1,     0,     INF],
  [INF,   INF,   2,     0]
];

// Floyd-Warshall algorithm
for (let k = 0; k < V; k++) {
  for (let i = 0; i < V; i++) {
    for (let j = 0; j < V; j++) {
      if (graph[i][k] + graph[k][j] < graph[i][j]) {
        graph[i][j] = graph[i][k] + graph[k][j];
      }
    }
  }
}

// Print result
for (let i = 0; i < V; i++) {
  let row = "";
  for (let j = 0; j < V; j++) {
    row += (graph[i][j] === INF ? "INF" : graph[i][j]) + "\t";
  }
  console.log(row);
}
```