Student: Damir Rahmani

Course: Performance Analysis of Computer Networks

Professor: prof.dr. Andrej Stefanov

Bellman Ford Algorithm

Introduction

The algorithm is named after its inventors, Richard Bellman and Lester Ford, who first described it in 1958. The Bellman Ford algorithm is a graph algorithm used to find the shortest path between a source node and all other nodes in a weighted graph, even if the graph contains negative edge weights. The algorithm works by iteratively relaxing each edge in the graph, where "relaxing" means updating the distance to a node if a shorter path is found. If the algorithm detects a negative weight cycle in the graph, it indicates that the graph contains no shortest paths, as the distance to a node can decrease infinitively by repeatedly traversing the negative cycle. The algorithm has a time complexity of O(V), where V is the number of vertices in the graph and E is the number of edges.

The Bellman Ford algorithm has a wide range of application in different fields, including:

- Networking – the algorithm is used to compute the shortest path between two nodes in a network. It is commonly used in internet routing protocols, where network topology can change dynamically.
- Transportation – the algorithm is used to find the shortest path between two locations in a transportation network. It can be used in route planning systems, such as GPS navigation, to find the fastest or most fuel-efficient route between two points.
- Computer Science – the algorithm is used in many computer science applications, including compilers, data compression, data transmission. It can be also used to solve various optimization problems.
- Finance – the algorithm is used in finance ton determine the shortest path between two assets in a financial network. This helps investors to optimize their portfolios and minimize risks.
- Robotics – the algorithm is used in robotics to determine the shortest path between two points in a given environment. It can be used to plan robot paths in a complex environment and avoid obstacles.

Overall, the Bellman Ford algorithm is a versatile algorithm that is used in various fields that require optimization or path-finding problems.

Algorithm overview

1. Initialize the distances of the source vertex to 0 and the distance of all vertices to infinity;
2. Iterate over all vertices in the graph V-1 times, where V is the number of vertices in the graph. In each iteration, relax all edges in the graph;
3. Relaxing an edge (u, v, w) means updating the distance to vertex v if a short path to v is found through u. To relax an edge, compare the distance to vertex u plus the weight of the edge (u, v)

with current distance to vertex v. If the distance through u is shorter, update the distance to v to the new distance;

4. After V-1 iterations, if the algorithm detects a negative weight cycle in the graph, it indicates that the graph contains no shortest paths, as the distance to a node can decrease infinitely by repeatedly traversing the negative cycle.
5. If there is no negative cycle, the algorithm returns the shortest path from the source vertex to all other vertices in the graph.

Pseudocode

```
function Bellman-Ford(Graph, source):
    distance[source] = 0
    for each vertex v in Graph:
        if v ≠ source
            distance[v] = infinity
    for i from 1 to |V|-1:
        for each edge (u, v, w) in Graph:
            if distance[u] + w < distance[v]:
                distance[v] = distance[u] + w
                predecessor[v] = u
    for each edge (u, v, w) in Graph:
        if distance[u] + w < distance[v]:
            // The graph contains a negative-weight cycle
            return "Negative cycle detected"
    return distance, predecessor
```

In this pseudocode, Graph is the input graph, source is the starting node, distance is an array that stores the shortest distance from the source to each node, and predecessor is an array that stores the predecessor of each node on the shortest path from the source.

The first step initializes the distance from the source node to itself as 0, and the distance to all other nodes as infinity. The second step sets the predecessor of all nodes to None.

The third step iterates over all edges in the graph |V|-1 times, updating the distance and predecessor for each node if a shorter path is found.

The fourth step checks for the presence of negative cycles in the graph. If a negative cycle is detected, the algorithm returns "Negative cycle detected", indicating that no shortest path exists.

Finally, the algorithm returns the arrays distance and predecessor, which contain the shortest path information for each node in the graph.

Time complexity

The worst case time complexity of the Bellman Ford algorithm is O(VE), where V is the number of vertices in the graph and E is the number of edges. This is because the algorithm iterates over all the edges V-1 times, and for each edge, it checks if relaxing it leads to a shorter path. Since the algorithm needs to check all edges, the time complexity is proportional to the number of edges.

Space complexity

The space complexity of the Bellman Ford algorithm is O(V). This is because the algorithm stores the distance of each vertex from the source vertex in an array of size V. It also stores the predecessor of each vertex on the shortest path, which requires an array of size V.

Note: in practice, the Bellman Ford algorithm can be optimized to have a lower complexity, especially if the graph is sparse. For example, the algorithm can be modified to terminate early if no distance updates occur in an iteration. Additionally, many real-world graphs tend to be sparse, meaning that they have relatively few edges compared to vertices, which can reduce the actual running time of the algorithm.

Specific use case

Routing protocols are used to determine the optimal path for data packets to travel from one network node to another. In IP routing protocols, the nodes are routers that maintain routing tables containing information about the network topology and the shortest path to reach each destination. When a data packet is transmitted, the router consults its routing table to determine the next hop on the path to the destination.

The Bellman Ford algorithm is used in distance-vector routing protocols, such as RIP and BGP, to calculate the shortest path to each destination node. In these protocols, each router broadcasts its routing table to its neighboring routers, which is turn update their own routing tables based on the received information. The routers repeat this process until all routing tables converge to the same values.

The Bellman Ford algorithm is used to update the routing table entries of each router. Specifically, each router maintains a table of distances to all other nodes in the network, where the distance is defined as the number of hops needed to reach the destination node. The algorithm is used to iteratively update these distances until they converge to the correct values. In each iteration, each router sends its routing table to the neighbors, who use the received information to update their own tables. Specifically, the Bellman Ford algorithm is used to calculate the distance from the source router to each destination node based on the received information. If a shorter path to the destination node is found through a neighbor, the distance is updated accordingly. This process is repeated until no more updates are possible, indicating that the routing tables have converged to the correct values.

Overall, the Bellman Ford algorithm is an important component of distance-vector routing protocols in computer networks, which are used to ensure reliable and efficient routing of data packets across the internet.

Variants and extensions

There are several variants and extensions of the Bellman Ford algorithm that are designed to address specific problems or improve its performance. Few examples:

- Shortest path algorithm with negative cycles – in some cases, a graph might contain a negative cycle, which is a cycle whose total weight is negative. In that case, the Bellman Ford algorithm does not terminate, as the distance to the nodes on the cycle can be made arbitrarily negative. One solution to this problem is to identify the negative cycle and report it as an error. Another solution is the Bellman Ford Moore algorithm, which modifies the Bellman Ford algorithm to terminate after a fixed number of iterations and reports a negative cycle if one is found.
- Shortest path algorithm with negative edges – the classic Bellman Ford algorithm assumes that all edge weights are non-negative. However, in some cases, negative weights might be present, and a modified algorithm is needed to find the shortest path. One such algorithm is the Johnson algorithm, which uses the Bellman Ford algorithm to re-weight the edges, making them non-negative, and then applies the Dijkstra's algorithm to find the shortest path.
- K-shortest path algorithm – the Bellman Ford algorithm finds the single shortest path between two nodes. However, in some cases, it may be desirable to find the K-shortest paths, where K is a fixed number. One such algorithm is the Yen algorithm, which uses the Bellman Ford algorithm as a subroutine to find the shortest path and then iteratively finds the next K-1 paths.
- Parallel algorithms – the Bellman Ford algorithm can be parallelized to improve its performance on large graphs. One example is the Delta Stepping algorithm, which divides the vertices into buckets based on their distance from the source node and processes them in parallel. Another example, is the Ligra algorithm, which uses a vertex-centric approach and parallelizes the relaxation step.

Advantages and disadvantages

Advantages:

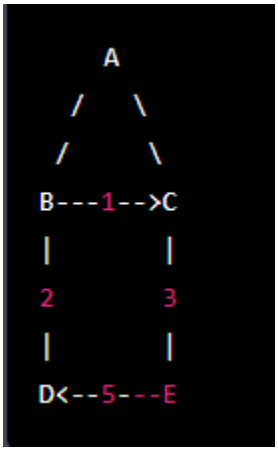
- Can handle graphs with negative edge weights;
- Can detect negative circles;
- Can work on directed graphs;

Disadvantages:

- Time complexity;
- May converge slowly;
- Not optimal for all graph types;

Example and illustration

Let us consider a simple example to illustrate the Bellman Ford algorithm. Suppose we have the following directed graph with weighted edges:



We want to find the shortest path from node A to all other nodes in the graph.

Step 1: Initialization

We start by setting the distance from the source node A to itself as 0, and the distance to all other nodes as infinity. We also set the predecessor of all nodes to None.

```
Distance = { A: 0, B: inf, C: inf, D: inf, E: inf }
Predecessor = { A: None, B: None, C: None, D: None, E: None }
```

Step 2: Relaxation

We then iterate over all edges in the graph, updating the distance and predecessor for each node if a shorter path is found.

```
for i in range(num_nodes-1):
    for edge in edges:
        u, v, weight = edge
        if Distance[u] + weight < Distance[v]:
            Distance[v] = Distance[u] + weight
            Predecessor[v] = u
```

In the first iteration, the distance to node B and C will be updated as follows:

```
Distance = { A: 0, B: 1, C: 4, D: inf, E: inf }
Predecessor = { A: None, B: A, C: B, D: None, E: None }
```

In the second iteration, the distance to node D and E will be updated as follows:

```
Distance = { A: 0, B: 1, C: 4, D: 6, E: 7 }
Predecessor = { A: None, B: A, C: B, D: B, E: D }
```

Step 3: Negative cycle check

In this case, there are no negative cycles in the graph, so we can skip this step.

Step 4: Output

The final output is the shortest distance from node A to each node in the graph:

```
Shortest distance from A to B: 1
Shortest distance from A to C: 4
Shortest distance from A to D: 6
Shortest distance from A to E: 7
```

This illustrates how the Bellman-Ford algorithm can be used to find the shortest path in a directed graph. Note that in this example, the algorithm required two iterations to converge on the shortest path, as there are no negative cycles in the graph. However, for graphs with negative cycles, the algorithm may not converge, and the presence of a negative cycle can be used to detect that no shortest path exists.

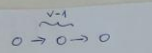
Project

The project will consist of the following four parts:

1. Explanatory part demonstrated with a specific example;
2. C++ implementation
3. Octave implementation
4. Example from part one implemented in steps 2 and 3;

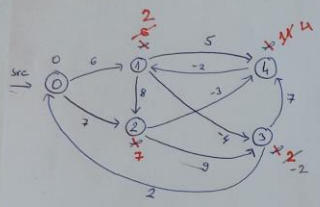
1. Explanatory part demonstrated with specific example.

step 1



step 2

repeated (V-1) times



edges

(0,1) (0,2)
(1,2) (1,3) (1,4)
(2,3) (2,4)
(3,0) (3,4)
(4,1)

	0	1	2	3	4
parent					
cost					

V = 5 V-1 = 4

Now we find distance values for all nodes

0	1	2	3	4	
0	∞	∞	∞	∞	← initialization
0	2	7	2	4	← 1st iteration
0	2	7	-2	4	← 2nd iteration
0	2	7	-2	4	← 3rd iteration

iteration #1

first time relaxation

	0	1	2	3	4
parent	-1	4 ~~0~~	0	1	~~1~~ 2
cost	0	~~6~~ 2	7	2	~~11~~ 4

(1,2) (1,3) (1,4)

→ 6+8 = 14 < 7 ⊥

→ 6+(-4) = 2 < ∞
replace ∞ with 2

→ 6+5 = 11 < ∞
replace ∞ with 11

(4,1)

→ 4+(-2) = 2 < 6
replace 6 with 2

relaxing edges

(0,1) (0,2)

→ 0+6 = 6 < ∞
replace ∞ with 6

→ 0+7 = 7 < ∞
replace ∞ with 7

(2,3) (2,4)

→ 7+9 = 16 < 2 ⊥

→ 7+(-3) = 4 < 11
replace 11 with 4

(3,0) (3,4)

→ 2+2 = 4 < 0 ⊥

→ 2+7 = 9 < 4 ⊥

iteration #2

second time relaxation

	0	1	2	3	4
parent	-1	4	0	1	2
cost	0	2	7	~~2~~ -2	4

(3,0) (3,4)

→ -2+2 = 0 < 0 ⊥

→ -2+7 = 5 < 4 ⊥

(4,1)

→ 4+(-2) = 2 < 2 ⊥

(0,1) (0,2)

→ 0+6 = 6 < 2 ⊥

→ 0+7 = 7 < 7 ⊥

(1,2) (1,3) (1,4)

→ 2+8 = 10 < 7 ⊥

→ 2+(-4) = -2 < 2
replace 2 with -2

→ 2+5 = 7 < 4 ⊥

iteration #3

third time relaxation

	0	1	2	3	4
parent	-1	4	0	1	2
cost	0	2	7	-2	4

(1,2) (1,3) (1,4)

→ 2+8 = 10 ≮ 7 ⊥

→ 2+(-4) = -2 ≮ -2 ⊥

→ 2+5 = 7 ≮ 4 ⊥

(4,1)

→ 4+(-2) = 2 ≮ 2 ⊥

note: max 4 iterations possible [V-1]

(0,1) (0,2)

→ 0+6 = 6 ≮ 2 ⊥

→ 0+7 = 7 ≮ 7 ⊥

(2,3) (2,4)

→ 7+9 = 16 ≮ -2 ⊥

→ 7+(-3) = 4 ≮ 4 ⊥

(3,0) (3,4)

→ -2+2 = 0 ≮ 0 ⊥

→ -2+7 = 5 ≮ 4 ⊥

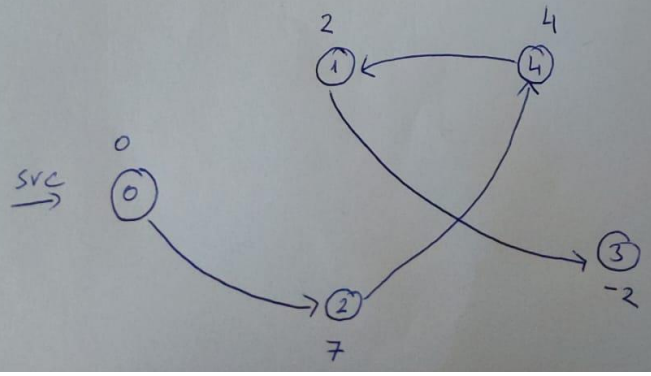
No update → stop → shortest path graph

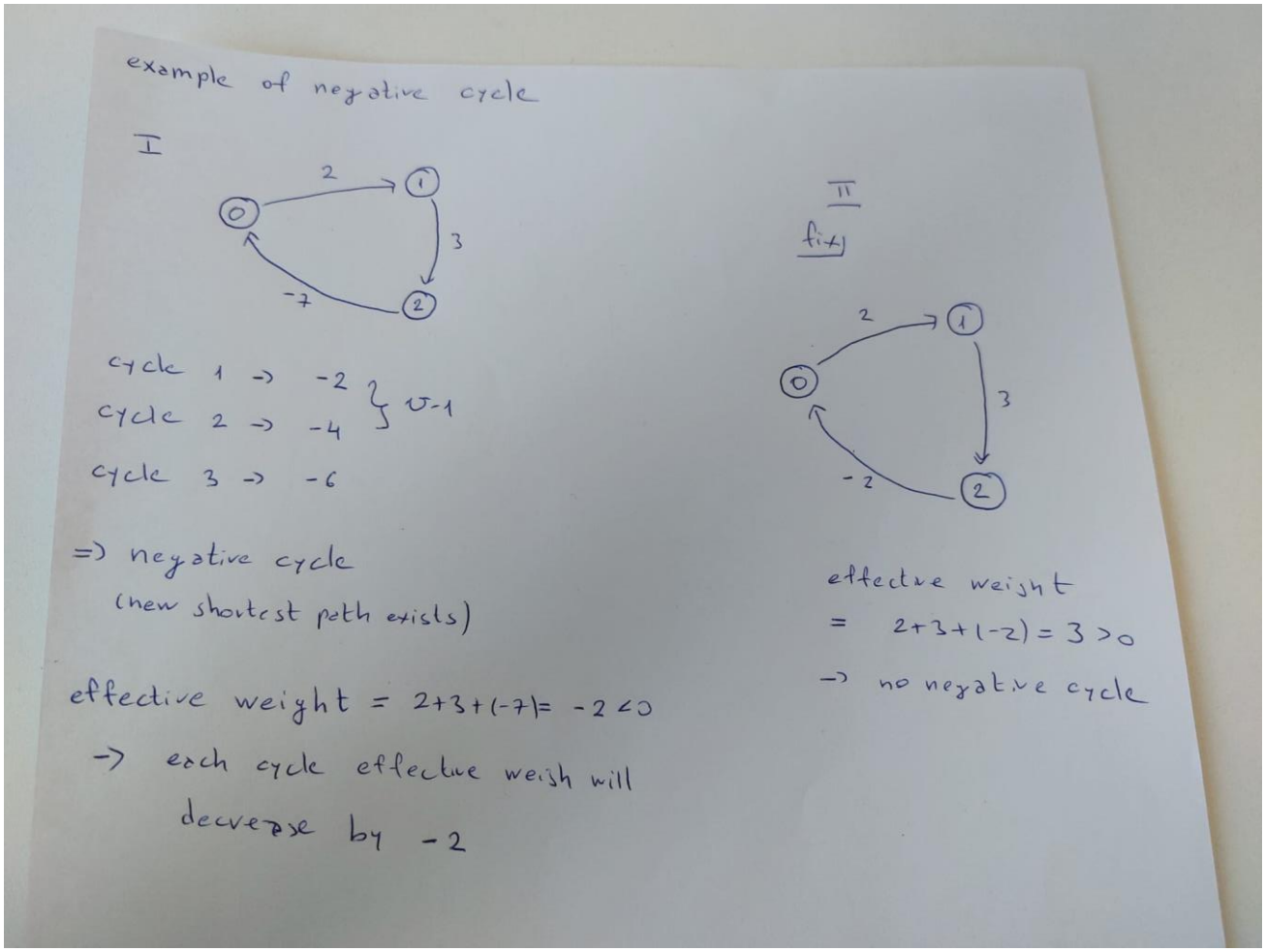
→ no negative cycles

Presentation

→ print single path shortest source graph

	0	1	2	3	4
parent	-1	4	0	1	2
cost	0	2	7	-2	4





2. C++ implementation

Note: the complete code is given in a GitHub repository.

This code is an implementation of the Bellman Ford algorithm for finding the shortest path in a weighted graph. This code is written in C++ and includes a command line menu that allows the user to choose between running the algorithm is C++, creating a weighted matrix and saving it to a file for use in Matlab/Octave, or opening a link to a GitHub repository containing the complete project.

Let's look at the details of the code.

The first few lines of the code include several header files that define various libraries and functions that are used in the code.

```
#include <vector>
#include <array>
#include <iostream>
#include <string>
#include <cstdlib>
#include <Windows.h>
#include <iomanip>
#include <fstream>
#include <shellapi.h>
```

Next, the code defines a few structures that will be used for the graph and the edges of the graph.

```
int V;

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};
```

The “V” variable represents the number of vertices in the graph. The “Edge” struct contains information about each edge, including the source, destination and weight. The “Graph” struct contains information about the graph, including the number of vertices and edges and a pointer to an array of edges. This code then defines a function for creating a graph based on the number of vertices and edges.

```
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}
```

The “createGraph” function allocates memeoty for a new graph, sets the number of vertices and edges, and allocates memory for an array of edges. The main part of the code begins with a “run” function that prompts the user to enter the number of vertices and edges in the graph, and then prompts the user to enter the source, destination, and weight of each edge.

```
int run() {
    int V, E;
    int s, d, w;

    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    struct Graph* graph = createGraph(V, E);

    for (int i = 0; i < E; i++) {
        cout << "Edge " << i;
        cout << endl;
        cout << "Enter source: ";
        cin >> s;
        cout << "Enter destination: ";
        cin >> d;
        cout << "Enter weight: ";
        cin >> w;
        graph->edge[i].src = s;
        graph->edge[i].dest = d;
        graph->edge[i].weight = w;
    }

    cout << endl;
    cout << endl;

    BellmanFord(graph, 0);

    return 0;
}
```

The “BellmanFord” function is then called with the graph that was created and the source vertex. This function implements the algorithm, which computes the shortest path from the source vertex to every other vertex in the graph.

```
void BellmanFord(struct Graph* graph, int src) {
    int const V = graph->V;
    int E = graph->E;
    int dist[5];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
```

3. Matlab/Octave code implementation

In this implementation, the code starts with a source vertex ('src') and initializes the distance to all other vertices as infinity ('dist = inf(1,n)') except for the source vertex, which is set to 0 ('dist(src)=0'). Then, the algorithm iteratively relaxes each edge in the graph by checking if the estimated distance from the source vertex to the destination vertex can be improved by going through the current vertex. If it can, then the estimated distance is updated ('dist(k)=dist(j)+adj(j,k)') and the process repeats until all edges have been relaxed 'n-1'times. Finally, the shortest distances from the source vertex to all other vertices are displayed using the 'disp' command.

Of course the algorithm is stored in a script, and to initialize the process the user enters the matrix of the weights and the number of nodes.

4. Example

Link to video presentation:

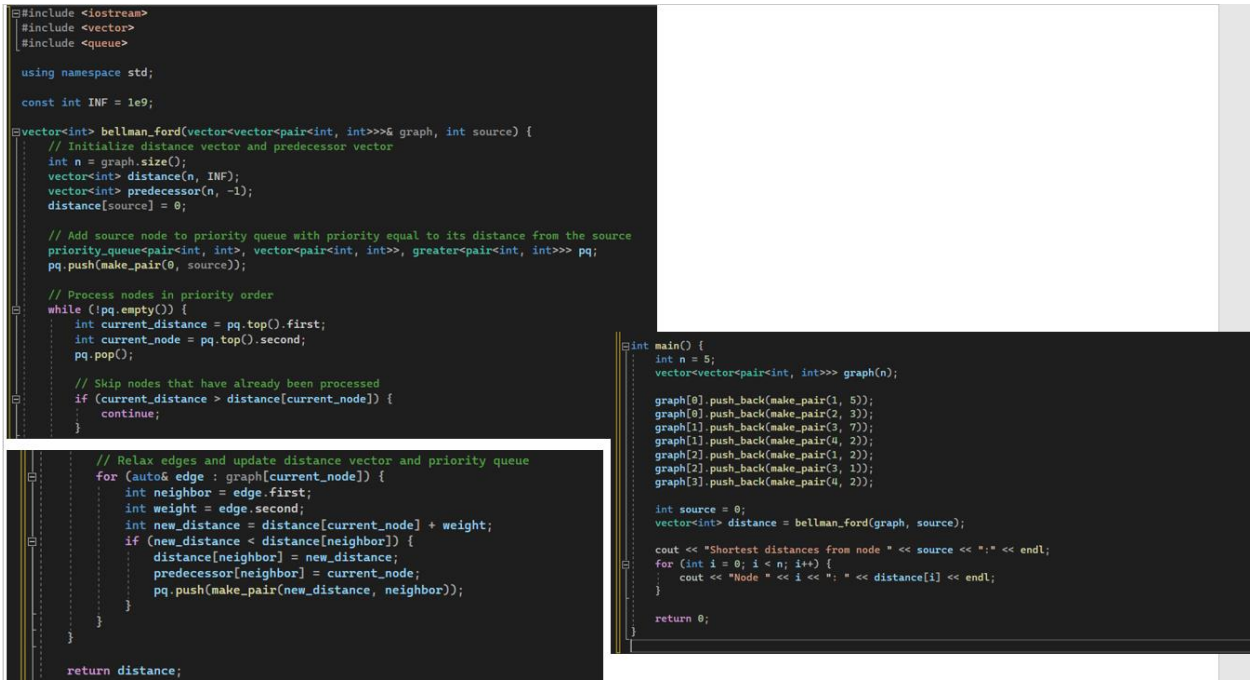
https://youtu.be/YsvuZmazVbE

How to improve the BF algorithm?

The Bellman-Ford algorithm is a fundamental algorithm for finding the shortest path between nodes in a graph, but there are several ways in which it can be improved. Here are a few ideas:

- Using a Priority Queue - instead of processing nodes in a linear order, we can prioritize them based on their distance from the source node. This way, we can process nodes that are closer to the source node first, which can reduce the number of iterations required in some cases.
- Early Stopping – the Bellman-Ford algorithm continues to iterate until it has converged and found the shortest path. However, in many cases, the shortest path can be found earlier. We can add a stopping condition that terminates the algorithm when the shortest path has been found.
- Adaptive Relaxation – the Bellman-Ford algorithm relaxes all edges in each iteration. However, we can use a more intelligent approach and only relax edges that have a chance of improving the shortest path. This can be achieved by keeping track of the nodes that have been relaxed in each iteration and only relaxing their outgoing edges in the next iteration.
- Avoiding Negative Cycles – the Bellman-Ford algorithm can get stuck in an infinite loop if there is a negative cycle in the graph. We can add a check to detect negative cycles and terminate the algorithm in such cases.

The focus will be on the first point.



```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int INF = 1e9;

vector<int> bellman_ford(vector<vector<pair<int, int>>>& graph, int source) {
    // Initialize distance vector and predecessor vector
    int n = graph.size();
    vector<int> distance(n, INF);
    vector<int> predecessor(n, -1);
    distance[source] = 0;

    // Add source node to priority queue with priority equal to its distance from the source
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push(make_pair(0, source));

    // Process nodes in priority order
    while (!pq.empty()) {
        int current_distance = pq.top().first;
        int current_node = pq.top().second;
        pq.pop();

        // Skip nodes that have already been processed
        if (current_distance > distance[current_node]) {
            continue;
        }

        // Relax edges and update distance vector and priority queue
        for (auto& edge : graph[current_node]) {
            int neighbor = edge.first;
            int weight = edge.second;
            int new_distance = distance[current_node] + weight;
            if (new_distance < distance[neighbor]) {
                distance[neighbor] = new_distance;
                predecessor[neighbor] = current_node;
                pq.push(make_pair(new_distance, neighbor));
            }
        }
    }

    return distance;
}

int main() {
    int n = 5;
    vector<vector<pair<int, int>>> graph(n);

    graph[0].push_back(make_pair(1, 5));
    graph[0].push_back(make_pair(2, 3));
    graph[1].push_back(make_pair(3, 7));
    graph[1].push_back(make_pair(4, 2));
    graph[2].push_back(make_pair(1, 2));
    graph[2].push_back(make_pair(3, 1));
    graph[3].push_back(make_pair(4, 2));

    int source = 0;
    vector<int> distance = bellman_ford(graph, source);

    cout << "Shortest distances from node " << source << ":" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Node " << i << ": " << distance[i] << endl;
    }

    return 0;
}
```

In this implementation, we use a ‘priority_queue’ with ‘pair<int, int>’ elements to store nodes ordered by their distance from the source node. We start by initializing the distance and predecessor vectors with infinite distances and ‘-1’ predecessor values. We then add the source node to the priority queue with priority 0.

In the main loop, we pop the node with the smallest distance from the priority queue and relax its outgoing edges. If we find a shorter path to a neighbor node, we update its distance, predecessor, and priority in the queue. We skip nodes that have already been processed by checking if their current distance is less than or equal to the distance in the vector.

Finally, we return the distance vector that contains the shortest path distances for each node in the graph. In the main function, we create a sample graph, run the Bellman-Ford algorithm with node 0 as the source, and print the resulting distances.

References

“Introduction to Algorithms” - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein – Chapter 24

GeeksforGeeks website - https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/