Graph Theory

Graph theory was first proposed by `Leonhard Euler`.

The result from koenigsberg bridge problem returns that,
" A graph that has nodes with an odd number of degrees more than 2. Then the graph will not support a path that travels all nodes using each edge once"

Some words for graph

- Node : The points where edges are connected
- Edge: The connection between nodes
- Complete Graph: All nodes are related to each other with unique edge
- Cycle: last node is connected to the first node making a graph cycle
- Wheel: if a node is connected to each node of a cycle then this is a wheel
- Bipartite: If nodes can be aligned in 2 groups that do not have any edge in the same group. Then this is a bipartite graph.

Graph Representation

We can represent graphs in 2 ways.

a. Adjacency Matrix
b. Adjacency List

Adjacency Matrix

A matrix that shows connection / edge between 2 nodes is the adjacency matrix. If I have n nodes in graph I have to take an $n \times n$ matrix to represent the graph.

```
1 —------------------ 2 —--------------- 3
|                     |                  |
|                     |                  |
4—--------------------5                  6
```

If here we create adjacency matrix,

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	1	0	1	0
3	0	1	0	0	0	1
4	1	0	0	0	1	0
5	0	1	0	1	0	0
6	0	0	1	0	0	0

starting with 1 indexing each row and col reads out its number. The (x,y) coordinate in the matrix if - 1 represents x and y has an edge. But if 0 then it is supposed that x,y coordinates do not share an edge.

Problems of using Adjacency Matrix

- Takes a lot of memory. For a graph of n nodes, it takes n^2 space.
- There is no benefit if the number of edges is less or more.
- Can not show multi-edge in any way

Adjacency List

In the adjacency list each node represents a list and the list stores the information of nodes they are connected to. For the graph above, the adjacency list is,

1 → 2 , 4
2 → 1 , 5
3 → 2 , 6
4 → 1 , 5
5 → 2 , 4
6 → 3

Now the constraint is O(E) the number of edges. Now edges can be as much as n^2. In that case the matrix and list will give the same result time… The less amount of edge means less amount of memory will be in use.

Code for representing graph [using matrix]

```
const int INF = 1e9;

class GraphMat{
    public:
        int V;
        bool isDirected;
        int **adj;
        GraphMat(int V, bool dir){
            this->V = V;
            isDirected = dir;
            adj = new int*[V];
            for(int i=0;i<V;i++){
                adj[i] = new int[V];
                for(int j=0;j<V;j++){
                    adj[i][j] = INF;
                }
            }

            for(int i=0;i<V;i++){
                adj[i][i] = 0;
            }
        }

        void addEdge(int u, int v, int cost){
            adj[u][v] = cost;
            if(!isDirected){
                adj[v][u] = cost;
            }
        }

        void printMatrix(){
            for(int i=0;i<V;i++){
                for(int j=0;j<V;j++){
                    cout<<adj[i][j]<<" ";
                }
                cout<<endl;
            }
        }

};
```

Code for representing graph [using list]

```
class graph{
    public:
    int v;
    vector<int> *adj;
    graph(int v){
        this→v = v;
        adj = new vector<int>[v];
    }

    void addEdge(int u, int v){
        adj[u].push_back(v);
        adj[v].push_back(u); //for undirected graph
    }

    void printGraph(){
        for(int i=0; i<v; i++){
            cout << i << " → ";
            for(int j=0; j<adj[i].size(); j++){
                cout << adj[i][j] << " ";
            }
            cout << endl;
        }
    }
};
```

Breadth First Search (BFS)

This technique is used to traverse through the graph and find the shortest path between 2 nodes. **Only works on unweighted graphs.** We can use BFS to find an element in a graph. In BFS we search first a node and then push all its adjacent nodes to a queue and then popping one by one from the queue we look at everyone's adj nodes. We keep track of visited nodes by pushing to an array.

```
bool bfs(graph g, int source, int destination){
    queue<int> q;
    int level[100];

    init(level, 100, -1);

    q.push(source);
    while(!q.empty()){
        int u = q.front();
        q.pop();

        for(int i=0; i < g.adj[u].size(); i++){
            int v = g.adj[u][i];

            if(v == destination){
                return true;
            }

            if(level[v] == -1){
                level[v] = 1;
                q.push(v);
            }
        }
    }

    return false;
}
```

Finding the Shortest Path using BFS

Finding the shortest path using BFS is a very easy task as it may differ only a line or two from the search purpose BFS. Using BFS we can,

- Find the minimum level distance from a node to another
- Shortest path finding from any node to another node.
- Bi-coloring a graph

Minimum Level Distance

```
int shortestPath(int source, int destination, graph g){
    int level[100];
    init(level, 100, -1);
    queue<int> q;

    q.push(source);
    level[source] = 0;

    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(level[v] == -1){
                level[v] = level[u] + 1;
                q.push(v);
            }
        }
    }
    return level[destination];
}
```

Shortest Path

To find the shortest path I have to first do 2 things.

1. Find the shortest path by finding levels.
2. Labeling the node parents with a `previous array`

```
int previous[100];

void ShortestPath(graph g, int source, int destination){
    queue<int> q;
    int level[100];

    init(level, 100, -1);

    q.push(source);

    while(!q.empty()){
        int u = q.front();
        q.pop();

        for(int i=0; i < g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(level[v] == -1){
                level[v] = level[u] + 1;
                previous[v] = u;
                q.push(v);
            }
        }
    }
    cout << level[destination] << endl;
}
```

Now after finding the shortest path, we can code the recursive function to print the path.

```
void printPath(int source, int destination){
    if(source == destination){
        cout << source;
    }
    else{
        printPath(source, previous[destination]);
        cout << " -> " << destination;
    }
}
```

Bi-Coloring

If we can color each node of a graph from 2 colors that no adjacent nodes share the same color then we can say " *This Graph is Bi-Colorable* "

```
bool isBicolorable(graph g, int source){
    int color[100];
    init(color, 100, -1);
    queue<int> q;

    q.push(source);
    color[source] = 1;

    while(!q.empty()){
        int u = q.front();
        q.pop();

        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(color[v] == -1){
                color[v] = 1 - color[u];
                q.push(v);
            }
            else if(color[v] == color[u]){
                return false;
            }
        }
    }
    return true;
}
```

Depth First Search (DFS)

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It’s similar to the Depth First Traversal of a tree, but unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, DFS uses a boolean-visited array.

DFS can be used in,
- Detecting Cycle/loop in a graph
- Finding if a path exists or not
- If the path exists, show the path
- Test for Bipartite Graph
- Finding strongly connected components

Finding Path using DFS

```
void dfs(graph g, int source, int destination){
    int visited[100];
    init(visited, 100, 0);
    stack<int> s;

    s.push(source);
    visited[source] = 1;

    while(!s.empty()){
        int u = s.top();
        s.pop();

        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(visited[v] == 0){
                visited[v] = 1;
                s.push(v);
            }
        }
    }
    if(visited[destination] == 1){
        cout << "Path exists" << endl;
    }
    else{
        cout << "Path does not exist" << endl;
    }
}
```

Printing the path

Now with a little of modification just like in BFS we did, we can make a recursive function that prints the parent of a node. By that, the path will be printed. We will again be needing a parent array.

```cpp
int parent[100];

void dfs(graph g, int source, int destination){
    int visited[100];
    init(visited, 100, 0);
    stack<int> s;
    s.push(source);
    visited[source] = 1;
    while(!s.empty()){
        int u = s.top();
        s.pop();
        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(visited[v] == 0){
                visited[v] = 1;
                s.push(v);
                parent[v] = u;
            }
        }
    }

    if(visited[destination] == 1){
        cout << "Path exists" << endl;
    }
    else{
        cout << "Path does not exist" << endl;
    }
}
void printPath(int source, int destination){
    if(source == destination){
        cout << source;
    }
    else{
        printPath(source, parent[destination]);
        cout << " -> " << destination;
    }
}
```

Detecting Loop/Cycle in a graph

For an undirected graph, we can check for a cycle like this,

```
int parent[100];
bool findCycle(graph g, int source){
    int visited[100];
    init(visited, 100, 0);
    stack<int> s;

    s.push(source);
    visited[source] = 1;

    while(!s.empty()){
        int u = s.top();
        s.pop();
        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(visited[v] == 0){
                visited[v] = 1;
                s.push(v);
                parent[v] = u;
            }
            else if(visited[v] == 1 && parent[u] != v){
                return true;
            }
        }
    }

    return false;
}
```

For a directed graph, the process is a little different. Here we have to keep track of nodes that are fully processed.

```
bool findCycleDG(graph g, int source) {
    int visited[100];
    int parent[100];
    init(visited, 100, 0);
    init(parent, 100, -1);

    stack<int> s;
    s.push(source);

    while (!s.empty()) {
        int u = s.top();
        s.pop();
        visited[u] = 1;

        for (int i = 0; i < g.adj[u].size(); i++) {
            int v = g.adj[u][i];
            if (visited[v] == 0) {
                s.push(v);
                parent[v] = u;
            } else if (visited[v] == 1) {
                return true;
            }
        }

        visited[u] = 2;
    }

    return false;
}
```

Detecting Bi-Partite Graph

Basically, the graphs which are bi-colorable are bipartite. So we can have the code like,

```
bool bicolorableDFS(graph g, int source){
    int color[100];
    init(color, 100, -1);
    stack<int> s;

    s.push(source);
    color[source] = 1;

    while(!s.empty()){
        int u = s.top();
        s.pop();

        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i];
            if(color[v] == -1){
                color[v] = 1 - color[u];
                s.push(v);
            }
            else if(color[v] == color[u]){
                return false;
            }
        }
    }

    return true;
}
```

Weighted Graph

A weighted graph is a graph in which a number (the weight) is assigned to each edge. These weights can represent things like distance, cost, or any arbitrary value.

```
A -1- B -2- D
|     |
3     4
|     |
C -5- E
```

Here red alphabets are nodes. And numbers are cost of that path. Here from node A to node E there are 2 paths,

```
A → B → E
A → C → E
```

But the total cost in the first path is : $1 + 4 = 5$
The total cost in the second path is : $3 + 5 = 8$

So the shortest path here is,

```
A → B → E
```

In Adjacency List we can store edges between nodes with their weight as pairs. Like this

```
A: [(B, 1), (C, 3)]
B: [(A, 1), (D, 2), (E, 4)]
C: [(A, 3), (E, 5)]
D: [(B, 2)]
E: [(B, 4), (C, 5)]
```

The code to create a weighted graph is,

```
class weightedGraph{
    public:
    int n;
    vector<pair<int, int>> *adj;
    weightedGraph(int n){
        this->n = n;
        adj = new vector<pair<int, int>>[n];
    }
    void addEdge(int u, int v, int w){
        adj[u].push_back(make_pair(v, w));
        adj[v].push_back(make_pair(u, w));
    }
};

void readGraph(weightedGraph &g, bool isDirected){
    int u, v, w;
    while(cin >> u >> v >> w){
        g.adj[u].push_back(make_pair(v, w));
        if(!isDirected){
            g.adj[v].push_back(make_pair(u, w));
        }
    }
}
```

Shortest Path in Weighted Graphs

In the realm of graph theory and computer science, the concept of finding the shortest path in a weighted graph is a fundamental and widely applicable problem. This problem involves finding the shortest or least costly path between two nodes in a graph, where each edge has an associated weight or cost.

Shortest Path Problem

The shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. This is a classic optimization problem and has been studied in depth due to its importance in fields like network routing, transportation, and logistics.

Algorithms for Shortest Path

There are several well-known algorithms to solve the shortest path problem:

1. **Dijkstra’s Algorithm**: This algorithm solves the shortest path problem for a graph with non-negative edge path costs, producing the shortest path tree.

2. **Bellman-Ford Algorithm**: Unlike Dijkstra's algorithm, the Bellman-Ford algorithm can handle graphs with negative edge weights, as long as there are no negative cycles.
3. **Floyd-Warshall Algorithm**: This algorithm finds the shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights (but no negative cycles). It's a dynamic programming algorithm that breaks the problem down into smaller subproblems and then combines the answers to those subproblems to solve the bigger, initial problem.
4. **A Search Algorithm**: Used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes.

Each of these algorithms has its own strengths and weaknesses and is suited to different types of problems and scenarios. Understanding these algorithms and knowing when to apply each one is a key skill in computer science and operations research.

In conclusion, the shortest path problem is a classic and important problem in graph theory and computer science. The ability to efficiently find the shortest or least costly path in a weighted graph has wide-ranging applications in many fields, from network design to transportation planning to logistics.

Path Relaxation

Path relaxation is a fundamental concept in graph algorithms, particularly those that deal with finding the shortest path between nodes in a graph. It's a process that iteratively improves the estimate of the shortest path between nodes.

In the context of graph theory, relaxation is the process of shortening the preliminary value of the shortest path from one node to another. Initially, we set the distance from the source node to all other nodes as infinity, except for the source node itself, which is zero.

During the relaxation process, we traverse the graph and continually update the shortest path estimates. If we find a shorter path to a node than the one previously recorded, we "relax" the path, meaning we replace the old path length with the new, shorter one.

Consider an edge in the graph from node `u` to node `v` with a weight of `w(u, v)`. If we have a shorter path from the source node to `v` that goes through `u`, we update the shortest path value of `v`. In other words, if `d(v) > d(u) + w(u, v)`, then we update `d(v) = d(u) + w(u, v)`. Here, `d(x)` represents the current shortest path estimate from the source to node `x`.

This process is repeated until no more updates are possible, meaning we've found the shortest paths to all nodes reachable from the source.

Path relaxation is crucial for the functioning of several shortest-path algorithms, including Dijkstra's algorithm and the Bellman-Ford algorithm. These algorithms use relaxation to iteratively refine the estimate of the shortest path until they arrive at the optimal solution.

In Dijkstra's algorithm, for example, relaxation is used in conjunction with a priority queue. The algorithm continually selects the node with the smallest shortest path estimate, relaxes all of its outgoing edges, and repeats this process until it has found the shortest path to every node.Here the weighted graph is being used as a graph. The code is provided above for the weighted graph.

```
const int inf = 1e9;
int dist[1000];

int PathRelaxation(weightedGraph g, int source, int destination){
    queue <int> q;
    init(dist, g.n, inf);
    q.push(source);
    dist[source] = 0;

    while(!q.empty()){
        int u = q.front();
        q.pop();

        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i].first;
            int w = g.adj[u][i].second;
            if(dist[v] > dist[u] + w){
                dist[v] = dist[u] + w;
                q.push(v);
            }
        }
    }
    return dist[destination];
}
```

This code is implementing a path relaxation algorithm for a weighted graph.

1. **Initialization**: The `inf` constant is set to a large number, representing infinity. The `dist` array will hold the shortest path estimates from the source to each node.
2. **PathRelaxation Function**: This function performs the path relaxation process on a given graph `g`, from a source node to a destination node. It initializes a queue and adds the source node to it, setting its distance to 0. Then, it enters a loop that continues until the queue is empty.
    a. In each iteration of the loop, it dequeues a node `u` and iterates over all of `u`'s adjacent nodes. For each adjacent node `v` with an edge weight `w`, it checks if the current shortest path to `v` can be improved by going through `u`. If so, it updates `v`'s shortest path estimate and adds `v` to the queue.

The `PathRelaxation` function returns the shortest path length from the source to the destination after performing the path relaxation process. If there's no path from the source to the destination, it will return `inf`, representing infinity.

Dijkstra Algorithm

Dijkstra's algorithm is a popular method used in computing and graph theory to find the shortest path between two nodes in a graph. Named after its creator, Dutch computer scientist Edsger W. Dijkstra, the algorithm works by iteratively selecting the vertex with the smallest tentative distance from the start, and then examining its neighbors. The algorithm keeps track of the accumulated shortest distance from the start node to all other nodes in the graph. It continues until it has processed all nodes, resulting in a shortest path tree. However, it's important to note that Dijkstra's algorithm only works with graphs with non-negative edge weights.

- GPS Navigation: Dijkstra's algorithm for real-time route optimization in GPS.
- Network Routing Protocols: Used in OSPF for data packet routing in computer networks.
- Telephone and cable TV Networks: Applied in network switching and routing.
- Social Networking Apps: Finds connections and suggests friends based on shortest paths.
- Google Maps: Provides optimal routes considering real-time factors like traffic.

```
void dijkastra(weightedGraph g, int source){
    int distance[100];
    init(distance, 100, INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int> >,
greater<pair<int, int> > > pq;

    pq.push(make_pair(0, source));
    distance[source] = 0;

    while(!pq.empty()){
        int u = pq.top().second;
        pq.pop();

        for(int i=0; i<g.adj[u].size(); i++){
            int v = g.adj[u][i].first;
            int w = g.adj[u][i].second;

            if(distance[v] > distance[u] + w){
                distance[v] = distance[u] + w;
                pq.push(make_pair(distance[v], v));
            }
        }
    }
    for(int i=0; i<g.n; i++){
        cout << i << " " << distance[i] << endl;
    }
}
```

The `Dijkstra` function is implementing Dijkstra's algorithm, which is used to find the shortest path from a source node to all other nodes in a weighted graph. Here's a step-by-step explanation:

1. **Initialization**: The function starts by initializing an array `distance` of size 100 with a large value (`INT_MAX`). This array will store the shortest distance from the source node to the other node. A priority queue `pq` is also initialized, which will be used to select the next node to visit.
2. **Source Node**: The source node is added to the priority queue with a distance of 0, and its distance is set to 0 in the `distance` array.
3. **Main Loop**: The function enters a loop that continues until the priority queue is empty. In each iteration of the loop, it removes the node `u` with the smallest distance from the priority queue.
4. **Relaxation**: For each neighbor `v` of `u`, it checks if the current shortest distance to `v` can be improved by going through `u`. If so, it updates the shortest distance to `v` and adds `v` to the priority queue.
5. **Output**: After the priority queue is empty, meaning all reachable nodes have been visited, it prints the shortest distance from the source node to each node.

This function effectively implements Dijkstra's algorithm. It's important to note that Dijkstra's algorithm assumes that all edge weights are non-negative, as negative weights can lead to incorrect results. Also, the graph is represented as an adjacency list, where `g.adj[u]` is a vector of pairs representing the neighbors of `u` and their corresponding edge weights.

Floyd Warshall Algorithm

The **Floyd-Warshall algorithm** is a classic computer science algorithm used for finding the shortest paths in a weighted graph with positive or negative edge weights (but no negative cycles). It's a dynamic programming algorithm that works by breaking the problem down into smaller subproblems and then combining the answers to those subproblems to solve the bigger, initial problem.

The algorithm operates on a 2D matrix, where each cell `[i][j]` represents the shortest distance from node `i` to node `j`. Initially, the diagonal of the matrix is filled with zeros (since the distance from a node to itself is zero), and the rest of the matrix is filled with the direct distances between the nodes. The algorithm then iteratively updates the matrix, considering each node as a possible intermediate step in the path between two other nodes.

By the end of the algorithm, the matrix contains the shortest possible distances between all pairs of nodes. The Floyd-Warshall algorithm has a time complexity of `O(n^3)`, where `n` is the number of nodes in the graph. This makes it less suitable for large graphs, but it's very useful for smaller graphs or graphs where all pairwise shortest paths are needed.

We will be using a graph that's represented in a matrix.

```
void floydWarshall(GraphMat g){
    for(int i=0; i<g.V; i++){
        for(int j=0; j<g.V; j++){
            for(int k=0; k<g.V; k++){
                if(g.adj[j][i] + g.adj[i][k] < g.adj[j][k]){
                    g.adj[j][k] = g.adj[j][i] + g.adj[i][k];
                }
            }
        }
    }
}
```

The `floydWarshall` function in this context is implementing the Floyd-Warshall algorithm on a graph represented as an adjacency matrix.

1. The function takes a `GraphMat` object `g` as an argument. This object represents a graph with `g.V` vertices, and its adjacency matrix is stored in `g.adj`.
2. The function consists of three nested loops, each iterating over the graph's vertices. The outer loop variable `i` represents the intermediate vertex considered for the shortest path between every pair of vertices.
3. The middle and inner loops, with variables `j` and `k`, represent all possible pairs of vertices in the graph. For each pair `(j, k)`, the algorithm checks if the path from `j` to `k` passing through `i` is shorter than the current shortest path from `j` to `k` (which is initially the direct edge from `j` to `k`, or infinity if there is no direct edge).
4. If the path through `i` is shorter, the algorithm updates the shortest path from `j` to `k` in the adjacency matrix (`g.adj[j][k] = g.adj[j][i] + g.adj[i][k]`).
5. After all pairs of vertices have been considered for each intermediate vertex, the adjacency matrix `g.adj` contains the shortest path distances between all pairs of vertices.

This function effectively implements the Floyd-Warshall algorithm, which is used to find the shortest paths between all pairs of vertices in a weighted graph. It's important to note that this function modifies the adjacency matrix of the graph in place, so after calling this function, `g.adj[j][k]` will give the shortest path distance from vertex `j` to vertex `k`.

What if the i loop or the bridge element is placed inside most of the three loops then what will happen?

```
void floydWarshall(GraphMat g){
    for(int j=0; j<g.V; j++){
        for(int k=0; k<g.V; k++){
            for(int i=0; i<g.V; i++){
                if(g.adj[j][i] + g.adj[i][k] < g.adj[j][k]){
                    g.adj[j][k] = g.adj[j][i] + g.adj[i][k];
                }
            }
        }
    }
}
```

If the `i` loop is placed inside most of the 3 loops, the Floyd-Warshall algorithm will not work correctly. The reason is that the Floyd-Warshall algorithm relies on a principle called “optimal substructure,” which means the shortest path from node `j` to node `k` through an intermediate node `i` depends on the shortest paths from `j` to `i` and from `i` to `k`.

In the correct implementation, the outer loop over `i` ensures that when considering paths that go through `i`, all shorter paths that go through nodes `0` to `i-1` have already been considered. This is because the algorithm assumes that the shortest path from `j` to `k` through `i` has already been found before it moves on to `i+1`.

If `i` is looped inside `j` and `k`, then when trying to update the shortest path from `j` to `k` through `i`, the algorithm might not have found the shortest path from `j` to `i` or from `i` to `k` yet, leading to incorrect results.

So, the order of the loops is crucial for the Floyd-Warshall algorithm to work correctly. The outer loop must be `i`, followed by `j` and `k` as inner loops. This ensures that all shorter paths are considered before longer ones, satisfying the optimal substructure property and leading to the correct shortest paths.

Bellman-Ford Algorithm

The Bellman-Ford algorithm is a graph search algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. This algorithm can handle graphs in which some of the edge weights are negative.

Here’s a brief description of how the algorithm works:

1. **Initialization**: The algorithm initializes the distance from the source vertex to itself as 0 and to all other vertices as infinity.
2. **Edge Relaxation**: The algorithm then repeatedly iterates over all edges in the graph. For each edge `(u, v)` with weight `w`, if the current distance estimate to `v` can be improved

by going through u, the algorithm updates the distance of v to the sum of the estimate of u and the weight of the edge (u, v). This is known as “relaxing” the edge.

3. **Negative Cycle Detection**: After iterating over all edges |V|-1 times, where |V| is the number of vertices in the graph, the algorithm has produced the shortest path tree. If the algorithm can still relax an edge, then there is a negative cycle in the graph. The Bellman-Ford algorithm can detect negative cycles and report their existence.

The Bellman-Ford algorithm is more versatile than Dijkstra’s algorithm, another well-known shortest path algorithm, because it can handle graphs with negative edge weights. However, it is also slower, with a time complexity of O(|V||E|), where |V| is the number of vertices and |E| is the number of edges in the graph. Despite its slower speed, the Bellman-Ford algorithm is still widely used in network routing protocols, most notably in the Internet’s Border Gateway Protocol (BGP).

```
const int inf = 1e9;

int dist[1000];

int bellmanFord(weightedGraph g, int source, int destination){
    init(dist, g.n, inf);
    dist[source] = 0;

    for(int i=0; i<g.n-1; i++){
        for(int u=0; u<g.n; u++){
            for(int j=0; j<g.adj[u].size(); j++){
                int v = g.adj[u][j].first;
                int w = g.adj[u][j].second;
                if(dist[v] > dist[u] + w){
                    dist[v] = dist[u] + w;
                }
            }
        }
    }

    // Check for negative weight cycles
    for(int u=0; u<g.n; u++){
        for(int j=0; j<g.adj[u].size(); j++){
            int v = g.adj[u][j].first;
            int w = g.adj[u][j].second;
            if(dist[v] > dist[u] + w){
                cout << "Graph contains a negative-weight cycle" << endl;
                return -1;
            }
        }
    }

    return dist[destination];
}
```

The Bellman-Ford algorithm is a graph search algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. This algorithm can handle graphs with negative weight edges, unlike Dijkstra's algorithm.

The function `bellmanFord` takes a weighted graph `g`, a source vertex `source`, and a destination vertex `destination` as input. It initializes the distance from the source to all vertices as infinity, except for the source itself, which is initialized with a distance of 0. Then, it relaxes the edges of the graph `g.n-1` times, where `g.n` is the number of vertices in the graph. In each iteration, it checks all edges `(u, v)` and updates the shortest distance to vertex `v` if the current distance to `v` is greater than the distance to `u` plus the weight of the edge `(u, v)`.

After the main loop, the function checks for negative weight cycles in the graph. If it can still relax an edge, it means that edge is part of a negative weight cycle. In such cases, the function prints a warning and returns -1. This is because in a graph with a negative weight cycle, the concept of a “shortest path” becomes undefined. If no negative weight cycle is detected, the function returns the shortest distance from the source to the destination. This implementation of the Bellman-Ford algorithm allows for the handling of graphs with negative weight edges and provides a way to detect negative weight cycles.