Prog 6. Develop a program to find the shortest path between vertices using the Bellman-Ford and path vector routing algorithm.

Bellman Ford Algorithm

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
// Define maximum number of vertices
#define MAX_VERTICES 1000
// Define infinity for initialization
#define INF INT_MAX
// Define Edge structure
typedef struct {
    int source;
    int destination;
    int weight;
} Edge;
// Define Bellman-Ford function
void bellmanFord(int graph[MAX_VERTICES][MAX_VERTICES],
                 int vertices, int edges, int source)
{
    // Declare distance array
    int distance[MAX_VERTICES];
    // Initialize distances from source to all vertices as
    // infinity
    for (int i = 0; i < vertices; ++i)
        distance[i] = INF;
    // Distance from source to itself is 0
```

```
distance[source] = 0;


// Relax edges V-1 times
for (int i = 0; i < vertices - 1; ++i) {
    // For each edge
    for (int j = 0; j < edges; ++j) {
        // If the edge exists and the new distance is
        // shorter
        if (graph[j][0] != -1
            && distance[graph[j][0]] != INF
            && distance[graph[j][1]]
                > distance[graph[j][0]]
                    + graph[j][2])
            // Update the distance
            distance[graph[j][1]]
                = distance[graph[j][0]] + graph[j][2];
    }
}
// Check for negative cycles
for (int i = 0; i < edges; ++i) {
    // If a shorter path is found, there is a negative
    // cycle
    if (graph[i][0] != -1
        && distance[graph[i][0]] != INF
        && distance[graph[i][1]]
            > distance[graph[i][0]] + graph[i][2]) {
        printf("Negative cycle detected\n");
```

```
            return;
        }
    }
    // Print shortest distances from source to all vertices
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < vertices; ++i)
        printf("%d \t\t %d\n", i, distance[i]);
}
// Define main function
int main()
{
    // Define number of vertices and edges
    int vertices = 6;
    int edges = 8;
    // Define graph as an array of edges
    int graph[MAX_VERTICES][MAX_VERTICES]
        = { { 0, 1, 5 }, { 0, 2, 7 }, { 1, 2, 3 },
            { 1, 3, 4 }, { 1, 4, 6 }, { 3, 4, -1 },
            { 3, 5, 2 }, { 4, 5, -3 } };
    // Call Bellman-Ford function with source vertex as 0
    bellmanFord(graph, vertices, edges, 0);
    return 0;
}
```

Path Vector Routing Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_NODES 10
```

```
// Structure to represent a node in the network
typedef struct Node {
    int id;
    int routing_table[MAX_NODES][3]; // [destination, next_hop, cost]
    int num_neighbors;
    struct Node* neighbors[MAX_NODES];
} Node;
// Function to initialize a node
void initialize_node(Node* node, int id) {
    node->id = id;
    node->num_neighbors = 0;
    for (int i = 0; i < MAX_NODES; i++) {
        for (int j = 0; j < 3; j++) {
            node->routing_table[i][j] = -1;
        }
    }
}
// Function to add a neighbor to a node
void add_neighbor(Node* node, Node* neighbor) {
    if (node->num_neighbors < MAX_NODES) {
        node->neighbors[node->num_neighbors] = neighbor;
        node->num_neighbors++;
    }
}
// Function to update the routing table of a node
void update_routing_table(Node* node) {
    for (int i = 0; i < node->num_neighbors; i++) {
        Node* neighbor = node->neighbors[i];
```

```
        for (int j = 0; j < MAX_NODES; j++) {
            if (neighbor->routing_table[j][0] != -1) {
                int destination = neighbor->routing_table[j][0];
                int cost = neighbor->routing_table[j][2];
                int total_cost = cost + 1; // Assuming a cost of 1 for each hop
                if (node->routing_table[destination][0] == -1 ||
                    total_cost < node->routing_table[destination][2]) {
                    // Update the routing table entry
                    node->routing_table[destination][0] = destination;
                    node->routing_table[destination][1] = neighbor->id;
                    node->routing_table[destination][2] = total_cost;
                }
            }
        }
    }
}
// Function to print the routing table of a node
void print_routing_table(Node* node) {
    printf("Routing table of Node %d:\n", node->id);
    printf("---------------------------\n");
    printf("| Destination | Next Hop | Cost |\n");
    printf("---------------------------\n");
    for (int i = 0; i < MAX_NODES; i++) {
        if (node->routing_table[i][0] != -1) {
            printf("|     %2d      |    %2d    |  %2d  |\n", node->routing_table[i][0],
                   node->routing_table[i][1], node->routing_table[i][2]);
        }
    }
```

```
    printf("-------------------------\n");
}
int main() {
    // Create nodes
    Node nodes[MAX_NODES];
    for (int i = 0; i < MAX_NODES; i++) {
        initialize_node(&nodes[i], i);
    }
    // Add neighbors
    add_neighbor(&nodes[0], &nodes[1]);
    add_neighbor(&nodes[0], &nodes[2]);
    add_neighbor(&nodes[1], &nodes[0]);
    add_neighbor(&nodes[1], &nodes[3]);
    add_neighbor(&nodes[2], &nodes[0]);
    add_neighbor(&nodes[2], &nodes[3]);
    add_neighbor(&nodes[3], &nodes[1]);
    add_neighbor(&nodes[3], &nodes[2]);
    // Run distance vector routing algorithm
    int convergence = 0;
    int iteration = 0;
    while (!convergence) {
        convergence = 1;
        printf("Iteration %d\n", iteration);
        for (int i = 0; i < MAX_NODES; i++) {
            update_routing_table(&nodes[i]);
        }

        // Check for convergence
```

```
        for (int i = 0; i < MAX_NODES; i++) {
            for (int j = 0; j < MAX_NODES; j++) {
                if (nodes[i].routing_table[j][0] != -1 &&
                    nodes[i].routing_table[j][2] != nodes[i].routing_table[j][2]) {
                    convergence = 0;
                    break;
                }
            }
            if (!convergence) {
                break;
            }
        }
        // Print routing tables
        for (int i = 0; i < MAX_NODES; i++) {
            print_routing_table(&nodes[i]);
        }
        iteration++;
        printf("\n");
    }
    return 0;
}
```