ALGORITHM LABORATORY

ASSIGNMENT-5A

PROBLEM STATEMENT: Implement Dijkstra and Bellman-Ford algorithms to find the shortest path from any source node to all nodes as destinations of a weighted graph of n vertices.

ALGORITHM (Dijkstra):

a. Input Handling
    i. Prompt the user to enter the number of vertices.
    ii. Ask the user to either input edges manually or generate a random graph.
b. Graph Generation
    i. For manual input, take pairs of nodes and their respective weights. Reject negative weights immediately.
    ii. For random generation, generate random weights for each edge using rand().
c. Shortest Path Calculation:
    i. Initialize the distance array with INT_MAX, and the source node's distance as 0.
    ii. Use a priority queue (min-heap) to process nodes in ascending order of their current known distances.
    iii. For each node processed, update its neighbouring nodes if a shorter path is found
d. Path Printing:
    i. Use the parent array to backtrack and reconstruct the path for each reachable node from the source.
    ii. Calculate and display the total path cost.

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct Connector {
    int start, end, weight;
};

void displayConnectors(vector<Connector> &links) {
    cout << "\nGraph Connections:\n";
    for (auto &link : links) {
        cout << link.start << " - " << link.end << " : " << link.weight << endl;
    }
}

void showPath(int startNode, vector<int> &route, vector<vector<pair<int, int>>> &network,
int endNode) {
    if (endNode == -1) {
        cout << "No route from " << startNode << " to " << endNode << endl;
        return;
    }
    vector<int> trail;
    for (int v = endNode; v != -1; v = route[v]) {
        trail.push_back(v);
    }
    reverse(trail.begin(), trail.end());

    cout << "Path from " << startNode << " to " << endNode << ": ";
    for (int i : trail) cout << i << " ";

    int totalCost = 0;
    for (int i = 0; i < trail.size() - 1; ++i) {
        for (auto &p : network[trail[i]]) {
            if (p.first == trail[i + 1]) {
                totalCost += p.second;
                break;
            }
        }
    }
    cout << "\nTotal Path Cost: " << totalCost << endl;
}

bool detectNegativeLoop(int size, vector<Connector>& links) {
    vector<int> travelDist(size, INT_MAX);
    travelDist[0] = 0;

    for (int i = 0; i < size - 1; i++) {
        for (auto &l : links) {
            if (travelDist[l.start] != INT_MAX && travelDist[l.start] + l.weight <
travelDist[l.end]) {
                travelDist[l.end] = travelDist[l.start] + l.weight;
```

```
            }
        }
    }

    for (auto &l : links) {
        if (travelDist[l.start] != INT_MAX && travelDist[l.start] + l.weight <
travelDist[l.end]) {
            cout << "Detected Negative Loop!" << endl;
            return true;
        }
    }
    return false;
}

void findShortestPath(int size, vector<vector<pair<int, int>>> &network, int start) {
    vector<int> travelDist(size, INT_MAX);
    vector<int> route(size, -1);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    travelDist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        int dist = pq.top().first;
        int node = pq.top().second;
        pq.pop();

        if (dist > travelDist[node]) continue;

        for (auto &p : network[node]) {
            int neighbor = p.first;
            int cost = p.second;
            if (travelDist[node] + cost < travelDist[neighbor]) {
                travelDist[neighbor] = travelDist[node] + cost;
                route[neighbor] = node;
                pq.push({travelDist[neighbor], neighbor});
            }
        }
    }

    cout << "\nDistances from Node " << start << ":\n";
    for (int i = 0; i < size; i++) {
        cout << "Node " << i << " : " << (travelDist[i] == INT_MAX ? "INF" :
to_string(travelDist[i])) << endl;
    }

    for (int i = 0; i < size; ++i) {
        if (i != start) showPath(start, route, network, i);
    }
}

int main() {
```

```
    int size, option, startPoint;
    cout << "Enter number of nodes: ";
    cin >> size;

    vector<Connector> links;
    vector<vector<pair<int, int>>> network(size);

    cout << "1. Manual Input\n2. Random Graph Generation\nChoose: ";
    cin >> option;

    if (option == 1) {
        int edges;
        cout << "Enter number of connections: ";
        cin >> edges;
        cout << "Enter links (start end weight):\n";
        for (int i = 0; i < edges; i++) {
            int a, b, w;
            cin >> a >> b >> w;
            links.push_back({a, b, w});
            network[a].push_back({b, w});
            network[b].push_back({a, w});
        }
    } else if (option == 2) {
        srand(time(0));
        for (int i = 0; i < size; i++) {
            for (int j = i + 1; j < size; j++) {
                int w = 1 + rand() % 100;
                links.push_back({i, j, w});
                network[i].push_back({j, w});
                network[j].push_back({i, w});
            }
        }
        displayConnectors(links);
    } else {
        cout << "Invalid option!" << endl;
        return 0;
    }

    if (detectNegativeLoop(size, links)) return 0;

    cout << "Enter starting node: ";
    cin >> startPoint;
    findShortestPath(size, network, startPoint);

    return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(E log V)
  - This occurs when the graph is sparse, meaning there are fewer edges compared to vertices. Each node is processed once, and the priority queue operations are minimal.
- Average Case: O(E log V)
- Worst Case: O(E log V)
  - This occurs when each vertex and edge must be processed with the maximum number of updates in the priority queue.

OUTPUT and PLOT:

Enter number of nodes: 4

Graph Connections:

0 - 1 : 10

0 - 2 : 66

0 - 3 : 30

1 - 2 : 38

1 - 3 : 38

2 - 3 : 21

Enter starting node: 2

Distances from Node 2:

Node 0 : 48

Node 1 : 38

Node 2 : 0

Node 3 : 21

Path from 2 to 0: 2 1 0

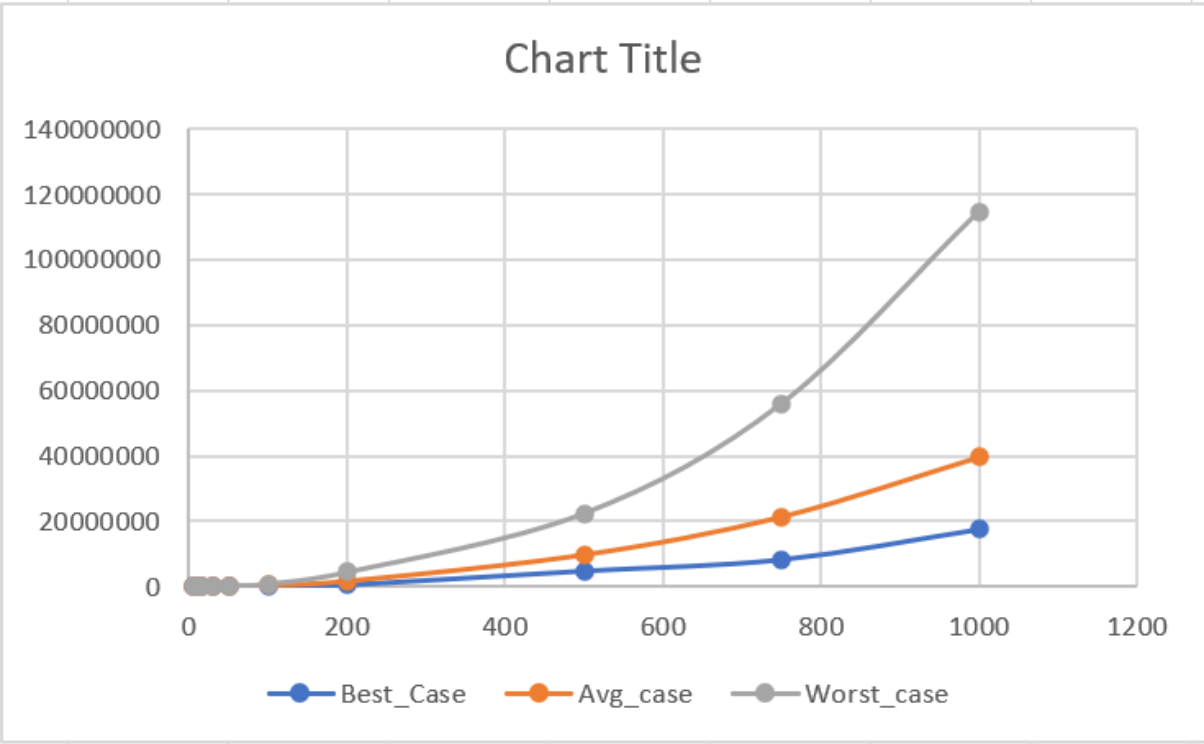
Total Path Cost: 48

Path from 2 to 1: 2 1

Total Path Cost: 38

Path from 2 to 3: 2 3

Total Path Cost: 21



ALGORITHM (Bellman Ford):

a. Input Reading:
    i. Prompt the user for the number of vertices (nodes).
    ii. Prompt the user to choose between manual edge entry or random graph generation.
    iii. If manual entry is chosen, prompt for the number of edges and their details.
    iv. If random graph generation is chosen, create random weights in the range of [-20, 20].
b. Initialize Distance and Parent Arrays:
    i. Create a distance array initialized with INT_MAX.
    ii. Create a parent array initialized with -1.

iii. Set the source node's distance to 0.

c. Relaxation Process:

i. Repeat the following process (nodes - 1) times:

I. For each edge (start, end, weight):

1. If distance[start] is not INT_MAX and distance[start] + weight < distance[end]:

a. Update distance[end].

b. Update parent[end].

II. If no updates occur in the current iteration, break out of the loop (early stopping).

d. Negative Cycle Detection:

i. Iterate over all edges.

ii. If distance[start] + weight < distance[end], a negative cycle is detected.

e. Output:

i. Display the final shortest path distances.

ii. For each node (except the source), reconstruct and display the shortest path using the parent array.

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct Connector {
    int start, end, weight;
};

void displayGraph(const vector<Connector> &links) {
    cout << "\nGraph Connections:\n";
    for (auto &link : links) {
        cout << link.start << " -> " << link.end << " : " << link.weight << endl;
    }
}

void displayDistances(int totalNodes, const vector<int> &range) {
    cout << "\nUpdated Distances:\n";
    for (int idx = 0; idx < totalNodes; idx++) {
        cout << "Node " << idx << " : " << (range[idx] == INT_MAX ? "INF" :
to_string(range[idx])) << endl;
    }
}
```

```
void trackPath(int source, const vector<int> &trace, int target, const vector<Connector>
&links) {
    if (target == -1) {
        cout << "No valid path from " << source << " to " << target << endl;
        return;
    }

    vector<int> route;
    for (int x = target; x != -1; x = trace[x]) {
        route.push_back(x);
    }
    reverse(route.begin(), route.end());

    cout << "Path from " << source << " to " << target << ": ";
    for (int step : route) {
        cout << step << " ";
    }
    cout << "\nTotal Weight: ";

    int totalWeight = 0;
    for (size_t idx = 0; idx < route.size() - 1; ++idx) {
        for (auto &link : links) {
            if (link.start == route[idx] && link.end == route[idx + 1]) {
                totalWeight += link.weight;
                break;
            }
        }
    }
    cout << totalWeight << endl;
}

void optimizedBellmanFord(int totalNodes, vector<Connector> &links, int origin) {
    vector<int> range(totalNodes, INT_MAX);
    vector<int> trace(totalNodes, -1);
    range[origin] = 0;

    for (int i = 1; i <= totalNodes - 1; i++) {
        bool modified = false;
        for (auto &link : links) {
            if (range[link.start] != INT_MAX && range[link.start] + link.weight <
range[link.end]) {
                range[link.end] = range[link.start] + link.weight;
                trace[link.end] = link.start;
                modified = true;
            }
        }
        if (!modified) break;
        displayDistances(totalNodes, range);
    }

    for (auto &link : links) {
```

```
        if (range[link.start] != INT_MAX && range[link.start] + link.weight <
range[link.end]) {
            cout << "Negative cycle detected!\n";
            return;
        }
    }

    cout << "\nFinal Distances from Node " << origin << "\n";
    displayDistances(totalNodes, range);

    for (int idx = 0; idx < totalNodes; ++idx) {
        if (idx != origin) {
            trackPath(origin, trace, idx, links);
        }
    }
}

void buildRandomGraph(int totalNodes, vector<Connector> &links) {
    srand(time(0));
    for (int i = 0; i < totalNodes; i++) {
        for (int j = i + 1; j < totalNodes; j++) {
            int randomWeight = -20 + rand() % 41;
            links.push_back({i, j, randomWeight});
            links.push_back({j, i, randomWeight});
        }
    }
    displayGraph(links);
}

int main() {
    int totalNodes, userChoice, origin;
    cout << "Enter total nodes: ";
    cin >> totalNodes;

    vector<Connector> links;

    cout << "Choose method:\n";
    cout << "1. Manual Entry\n2. Auto-Generated Graph\n";
    cin >> userChoice;

    if (userChoice == 1) {
        int totalLinks;
        cout << "Enter total connections: ";
        cin >> totalLinks;
        cout << "Enter connections (start end weight):\n";
        for (int i = 0; i < totalLinks; i++) {
            int s, e, w;
            cin >> s >> e >> w;
            links.push_back({s, e, w});
        }
    } else if (userChoice == 2) {
        buildRandomGraph(totalNodes, links);
```

```
    } else {
        cout << "Invalid option selected!\n";
        return 0;
    }

    cout << "Enter starting node: ";
    cin >> origin;
    optimizedBellmanFord(totalNodes, links, origin);

    return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(E)
    - Occurs when no updates are needed after the first relaxation step, and the algorithm exits early. This is possible if the optimal distances are achieved in the first iteration itself.
- Average Case: O(V × E)
- Worst Case: O(V × E)
    - Happens when updates are required in every iteration up to the (V-1)th step, and no early termination occurs. This is the theoretical upper limit of the algorithm.

OUTPUT and PLOT:

Enter total nodes: 4

Graph Connections:

0 -> 1 : 15

1 -> 0 : 15

0 -> 2 : -5

2 -> 0 : -5

0 -> 3 : -14

3 -> 0 : -14

1 -> 2 : -9

2 -> 1 : -9

1 -> 3 : 4

3 -> 1 : 4

2 -> 3 : -16

3 -> 2 : -16

Enter starting node: 2

Updated Distances:

Node 0 : -33

Node 1 : -15

Node 2 : -35

Node 3 : -19

Updated Distances:

Node 0 : -71

Node 1 : -53

Node 2 : -73

Node 3 : -57

Updated Distances:

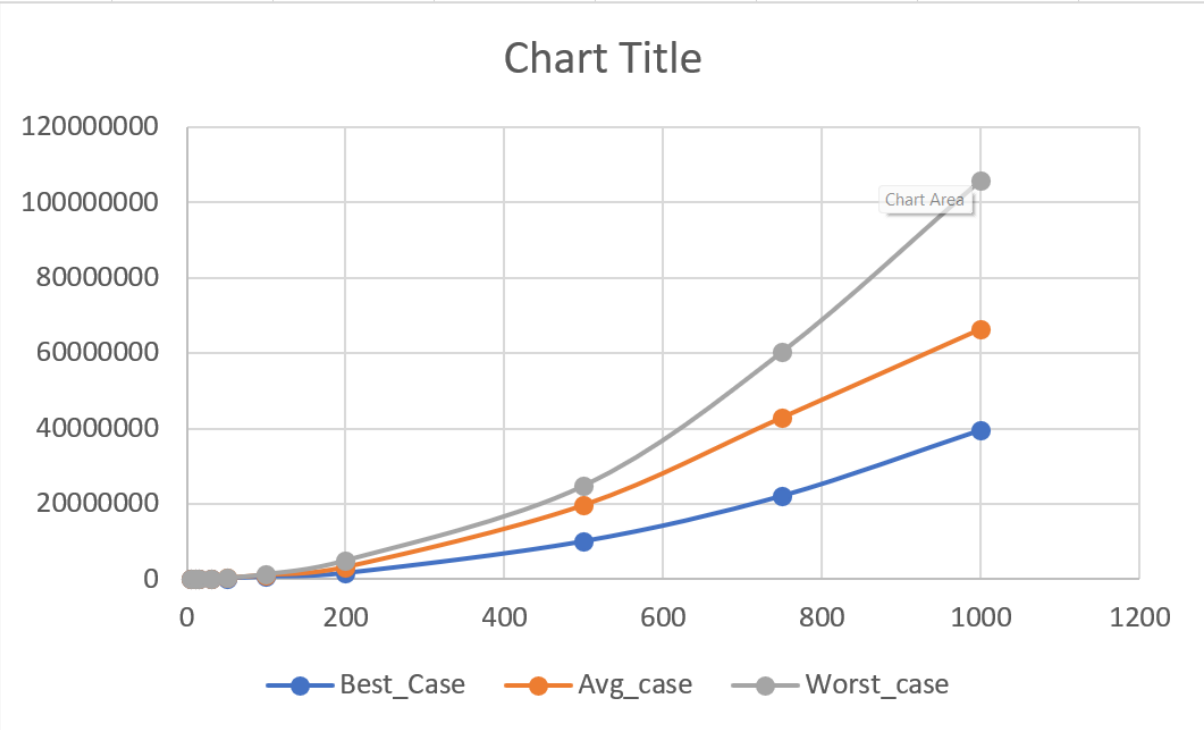
Node 0 : -109

Node 1 : -91

Node 2 : -111

Node 3 : -95

Negative Cycle detected!!



ASSIGNMENT-5B

PROBLEM STATEMENT: Implement BFS and DFS algorithms to traverse any graph of n vertices and print the traversal sequence.

ALGORITHM (DFS):

a. Input Handling
   i. Read the total number of vertices (nodesCount).
   ii. Ask the user whether they want to enter edges manually or generate a random graph.
b. Graph Construction:
   i. If manual entry is chosen, read totalEdges and the corresponding pairs of vertices.
   ii. If random generation is chosen, generate a random undirected graph by iterating through each pair of nodes and adding edges with both directions.

c. Adjacency List Formation:
   i. Create a 2D vector (adjList) to store the graph connections.

ii. Insert each edge pair into the adjacency list.

d. DFS Traversal:
   i. Initialize a visitedNodes array to track visited nodes.
   ii. Use a route vector to store the DFS path.
   iii. Call the depthFirstSearch function, starting from the chosen source vertex.

e. DFS Function (depthFirstSearch):
   i. Mark the current node as visited.
   ii. Add the node to the path.
   iii. For each connected node, recursively call depthFirstSearch if the node is unvisited.

f. Output:
   i. Display the graph structure.
   ii. Print the DFS path.

PROGRAM CODE:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
using namespace std;

struct Link {
    int start, stop;
};

void showLinks(vector<Link> &links) {
    cout << "\nGraph Edges:\n";
    for (auto &link : links) {
        cout << link.start << " -> " << link.stop << endl;
    }
}

void depthFirstSearch(int point, vector<vector<int>> &connections, vector<bool> &seen,
vector<int> &track) {
    seen[point] = true;
    track.push_back(point);
    cout << "Visited Node: " << point << endl;

    for (int link : connections[point]) {
        if (!seen[link]) {
            depthFirstSearch(link, connections, seen, track);
        }
    }
```

```
}

void showTraversalPath(vector<int> &track) {
    cout << "\nDFS Traversal Path: ";
    for (int point : track) {
        cout << point << " ";
    }
    cout << endl;
}

void autoGenerateGraph(int totalPoints, vector<Link> &links) {
    srand(time(0));
    for (int i = 0; i < totalPoints; i++) {
        for (int j = i + 1; j < totalPoints; j++) {
            links.push_back({i, j});
            links.push_back({j, i});
        }
    }
    showLinks(links);
}

int main() {
    int nodesCount, option, beginPoint;
    cout << "Enter total vertices: ";
    cin >> nodesCount;

    vector<Link> links;

    cout << "Choose an option:\n";
    cout << "1. Manual Input\n";
    cout << "2. Auto-Generate Graph\n";
    cin >> option;

    if (option == 1) {
        int totalLinks;
        cout << "Enter number of edges: ";
        cin >> totalLinks;
        cout << "Enter edges (start stop):\n";
        for (int i = 0; i < totalLinks; i++) {
            int start, stop;
            cin >> start >> stop;
            links.push_back({start, stop});
        }
    } else if (option == 2) {
        autoGenerateGraph(nodesCount, links);
    } else {
        cout << "Invalid option selected!\n";
        return 0;
    }

    vector<vector<int>> connections(nodesCount);
    for (auto &link : links) {
```

```
        connections[link.start].push_back(link.stop);
    }

    cout << "Enter starting vertex for DFS: ";
    cin >> beginPoint;

    vector<bool> seen(nodesCount, false);
    vector<int> track;

    depthFirstSearch(beginPoint, connections, seen, track);

    showTraversalPath(track);

    return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(V)
  - Occurs when the graph is highly disconnected, and the DFS only explores one connected component or minimal nodes.
- Average Case: O(E + V)
- Worst Case: O(E + V)
  - Occurs in a densely connected graph (like a complete graph) where every node is connected to every other node.

OUTPUT and PLOT:

Enter total vertices: 3

Graph Edges:

0 -> 1

1 -> 0

0 -> 2

2 -> 0

1 -> 2

2 -> 1

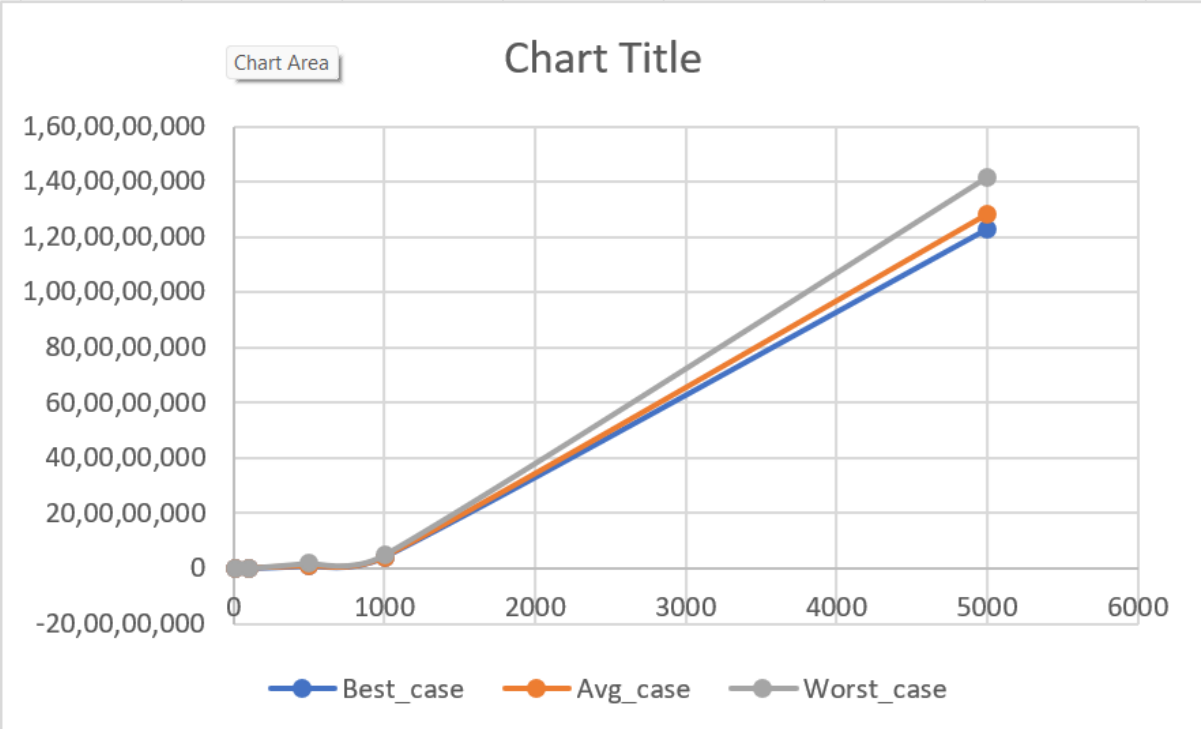
Enter starting vertex for DFS: 0

Visited Node: 0

Visited Node: 1

Visited Node: 2

DFS Traversal Path: 0 1 2



ALGORITHM (BFS):

a. Define the Data Structure:
   i. Create a structure called Link instead of Edge.
   ii. The structure should store start and end points of the link.
b. Print Links Function:
   i. Define a function displayLinks() to print all links in the graph.
c. BFS Function:
   i. Create a BFS function breadthFirstTraversal() to perform BFS.
   ii. Maintain a queue for node traversal.
   iii. Use a boolean vector visitedNodes to track visited nodes.
   iv. Track visited nodes in an array exploredPath.
d. Generate Graph Function:
   i. Create autoGenerateGraph() to generate random links.

e. Main Function:
   i. Input the number of vertices and choice for manual/random graph generation.
   ii. For manual input, take the number of links and their connections.
   iii. For random graph generation, use rand() to generate random links.
   iv. Create an adjacency list graphLinks to store the connections.

   v. Ask the user for the starting node for BFS.
   vi. Call the BFS function and print the resulting path.

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct Link {
    int start, end;
};

void displayLinks(vector<Link> &connections) {
    cout << "\nConnections in the Network:\n";
    for (auto &link : connections) {
        cout << link.start << " - " << link.end << endl;
    }
}

void breadthFirstTraversal(int startNode, vector<vector<int>> &graphLinks,
                           vector<bool> &visitedNodes, vector<int> &exploredPath) {
    queue<int> nodeQueue;
    visitedNodes[startNode] = true;
    nodeQueue.push(startNode);

    while (!nodeQueue.empty()) {
        int currentNode = nodeQueue.front();
        nodeQueue.pop();
        exploredPath.push_back(currentNode);
        cout << "Visited: " << currentNode << endl;

        for (int neighbor : graphLinks[currentNode]) {
            if (!visitedNodes[neighbor]) {
                visitedNodes[neighbor] = true;
                nodeQueue.push(neighbor);
            }
        }
```

```
    }
}

void displayTraversalPath(vector<int> &exploredPath) {
    cout << "\nTraversal Route: ";
    for (int node : exploredPath) {
        cout << node << " ";
    }
    cout << endl;
}

void autoGenerateGraph(int totalNodes, vector<Link> &connections) {
    srand(time(0));
    for (int i = 0; i < totalNodes; i++) {
        for (int j = i + 1; j < totalNodes; j++) {
            connections.push_back({i, j});
            connections.push_back({j, i});
        }
    }
    displayLinks(connections);
}

int main() {
    int totalNodes, userChoice, startingNode;
    cout << "Enter number of nodes in the network: ";
    cin >> totalNodes;

    vector<Link> connections;

    cout << "Choose an option:\n";
    cout << "1. Enter connections manually\n";
    cout << "2. Generate a random network\n";
    cin >> userChoice;

    if (userChoice == 1) {
        int totalConnections;
        cout << "Enter number of connections: ";
        cin >> totalConnections;
        cout << "Enter connections (start end):\n";
        for (int i = 0; i < totalConnections; i++) {
            int start, end;
            cin >> start >> end;
            connections.push_back({start, end});
        }
    } else if (userChoice == 2) {
        autoGenerateGraph(totalNodes, connections);
    } else {
        cout << "Invalid option selected!\n";
        return 0;
    }

    vector<vector<int>> graphLinks(totalNodes);
```

```
    for (auto &link : connections) {
        graphLinks[link.start].push_back(link.end);
    }

    cout << "Enter the starting node for traversal: ";
    cin >> startingNode;

    vector<bool> visitedNodes(totalNodes, false);
    vector<int> exploredPath;

    breadthFirstTraversal(startingNode, graphLinks, visitedNodes, exploredPath);

    displayTraversalPath(exploredPath);

    return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(V)
  - Occurs when the starting node has no outgoing edges or the graph is minimally connected.
- Average Case: O(V + E)
- Worst Case: O(V + E)
  - Occurs in the densest graph possible, where each node is connected to every other node (complete graph).

OUTPUT and PLOT:

Enter number of nodes in the network: 3

Connections in the Network:

0 - 1

1 - 0

0 - 2

2 - 0

1 - 2

2 - 1

Enter the starting node for traversal: 2

Visited: 2

Visited: 0

Visited: 1

Traversal Route: 2 0 1