Addis Ababa University
አዲስ፡አበባ፡ዩኒቨርሲቲ
SEEK WISDOM, ELEVATE YOUR INTELLECT AND SERVE HUMANITY !

ADDIS ABABA INSTITUE OF TECHNOLOGY

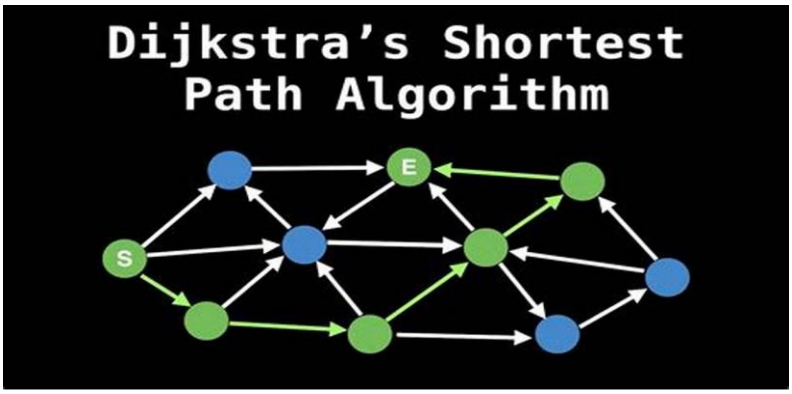
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

COURSE TITLE - DATA STRUCTURE

COURSE CODE - ECEG - 4192

GROUP PROJECT

PROJECT TITLE – IMPLEMENTATION OF DIJKSTRA'S ALGORITHM



GROUP MEMBERS **ID NUMBER**

- **EMNET MAMO ……. UGR/O545/12**
- **MATEWOS TEGETE …… UGR/6649/12**

- **SUBMITTED TO - Dr. Menore T**
- **SUBMISSION DATE – 29 May 2023**

TABLE OF CONTENT

IMPLEMENTION OF DIJKSTRA'S ALGORITHM

1. INTRODUCTION

In this project we have used c++ programming language to implemented a weighted graph data structure and utilized the **Dijkstra's algorithm** to calculate the shortest distance between two vertices in the graph. A weighted graph is a collection of vertices connected by edges, where each edge has a weight assigned to it. Our code provides functionalities to create a weighted graph, insert edges with weights, and retrieve information such as the degree of a vertex and the total number of edges. The `distance` function uses Dijkstra's algorithm to find the shortest path between two vertices by iteratively exploring neighboring vertices and updating the distances. This code serves as a foundation for modeling and analyzing various real-world scenarios, such as in networks routing algorithm , transportation networks and resource allocation problems, where the weight on the edges represents distances, costs, or any other relevant metric.

2. OVERALL CONCEPUTAL DESCRIPTION OF THE CODE IMPLEMETED

Our code implements a weighted graph data structure and calculates the shortest distance between two vertices in the graph using Dijkstra's algorithm.

- The `Weighted_graph` class represents a weighted graph and contains the following **private member** variables:
  - **num_vertices** : An integer representing the number of vertices in the graph.
  - **adjacency_matrix** : A 2D vector of doubles representing the adjacency matrix of the graph. Each entry `adjacency_matrix[i][j]` stores the weight of the edge between vertices `i` and `j`.

The **public member** functions of the `Weighted_graph` class are as follows:

- **Weighted_graph(int n = 50)** : The constructor initializes the graph with a given number of vertices (default is 50). It checks if the number of vertices is valid, and then resizes the adjacency matrix to have `n` rows and columns, with all entries set to infinity.
- **~Weighted_graph()** : The destructor clears the adjacency matrix.
- **int degree(int n) const :** Returns the degree of a given vertex `n`. It counts the number of edges connected to vertex `n` and returns the count.

- **int edge_count() const** : Returns the total number of edges in the graph. It iterates over all pairs of vertices in the upper triangular part of the adjacency matrix and counts the edges whose weight is not infinity.
- **double adjacent(int m, int n) const** : Returns the weight of the edge between vertices `m` and `n`. It checks if the vertex indices are valid and returns the corresponding entry in the adjacency matrix.
- **void insert(int m, int n, double w)** : Inserts an edge between vertices `m` and `n` with a given weight `w`. It checks if the vertex indices are valid and the weight is positive and finite. It updates the adjacency matrix accordingly by setting the corresponding entry to `w`.
- **double distance(int m, int n)** : Calculates the shortest distance between vertices `m` and `n` using Dijkstra's algorithm. It checks if the vertex indices are valid. It initializes a distance vector with all entries set to infinity, except the distance of the source vertex (`m`) set to 0. It iteratively selects the vertex with the smallest distance among the unvisited vertices and updates the distances of its adjacent vertices if a shorter path is found. Finally, it returns the shortest distance between the source and target vertices.

❖ The `main` function demonstrates the usage of the `Weighted_graph` class. It creates a weighted graph with 5 vertices and inserts edges with weights. Then, it prompts the user to enter two vertices and calculates the shortest distance between them using the `distance` function. If an error occurs during the distance calculation, it catches the exception and prints the error message.

❖ Overall, the code provides a basic implementation of a weighted graph and demonstrates how to use it to calculate the shortest distance between two vertices.

3. FURTHER DESCRIPTION OF EACH FUCTION WITH PSEUDOCODES

Member Functions

❖ Constructors

❖ Weighted_graph (int n = 50) :

- ✓ The constructor for the `Weighted_graph` class takes an optional parameter `n` (default value is 50) to specify the number of vertices in the graph.
- ✓ If the value of `n` is less than or equal to 0, it is set to 1 to ensure a valid number of vertices.
- ✓ The `num_vertices` variable is set to the value of `n`.

- ✓ The `adjacency_matrix` is resized to have `n` rows and `n` columns, with all entries initially set to infinity.

PSEUDOCODE LOOKS LIKE:

```
Constructor Weighted_graph(n):
        if n is less than or equal to 0:
            Set n to 1
        Set num_vertices to n
    Create an adjacency_matrix of size n x n filled with
                                infinity
```

❖ Destructor

❖ ~Weighted_graph() :

- ✓ The destructor for the `Weighted_graph` class clears the `adjacency_matrix`, effectively freeing the memory occupied by the adjacency matrix.

PSEUDOCODE LOOKS LIKE:

```
Destructor ~Weighted_graph():
    Clear the adjacency_matrix
```

❖ Accessors

- The class `Weighted_graph` has three accessors:

❖ int degree(int n) const :

- ✓ The `degree` function takes a vertex index `n` as input and returns the degree of that vertex, which is the count of edges connected to it.
- ✓ If the vertex index `n` is less than 0 or greater than or equal to `num_vertices`, an `out_of_range` exception is thrown.
- ✓ The function initializes a `degree` variable to 0.
- ✓ It iterates over all vertices from 0 to `num_vertices - 1` and checks if the current vertex is not the same as `n` and if there is an edge between them (edge weight is less than infinity). If both conditions are true, it increments the `degree` variable.
- ✓ Finally, the function returns the value of `degree`.

PSEUDOCODE LOOKS LIKE:

```
Function degree(n):
   if n is less than 0 or greater than or equal to num_vertices:
      Throw an "out_of_range" exception with the message "Invalid vertex index"

   Initialize degree to 0

   For each vertex i from 0 to num_vertices - 1:
      If i is not equal to n and adjacency_matrix[n][i] is less than infinity:
         Increment degree by 1

                                Return degree
```

❖ int edge_count() const :

- ✓ The `edge_count` function returns the total number of edges in the graph.
- ✓ It initializes a `count` variable to 0.
- ✓ It iterates over all pairs of vertices in the upper triangular part of the adjacency matrix (excluding the diagonal).
- ✓ For each pair of vertices (`i`, `j`), if there is an edge between them (edge weight is less than infinity), it increments the `count` variable.
- ✓ Finally, the function returns the value of `count`.

PSEUDOCODE LOOKS LIKE:

```
Function edge_count():
            Initialize count to 0

            For each vertex i from 0 to num_vertices - 1:
               For each vertex j from i + 1 to num_vertices - 1:
                  If adjacency_matrix[i][j] is less than infinity:
                     Increment count by 1

                        Return count
```

❖ double adjacent(int m, int n) const :

- ✓ The `adjacent` function takes two vertex indices `m` and `n` as input and returns the weight of the edge between them.
- ✓ If either `m` or `n` is less than 0 or greater than or equal to `num_vertices`, an `out_of_range` exception is thrown.
- ✓ If `m` and `n` are the same, indicating the same vertex, the function returns 0.0.
- ✓ Otherwise, it retrieves the value stored in the adjacency matrix at the position `m` and `n`, representing the weight of the edge between the vertices, and returns it.

PSEUDOCODE LOOKS LIKE:

```
Function adjacent(m, n):
        if m is less than 0 or greater than or equal to num_vertices or n is less
    than 0 or greater than or equal to num_vertices:
        Throw an "out_of_range" exception with the message "Invalid vertex
    index"

      If m is equal to n:
        Return 0.0

  Return adjacency_matrix[m][n]
```

❖ Mutators

- The class `Weighted_graph` has two mutators:

❖ void insert(int m, int n, double w) :

- ✓ The `insert` function takes two vertex indices `m` and `n` and a weight `w` as input and inserts an edge between the vertices with the specified weight.
- ✓ If either `m` or `n` is less than 0 or greater than or equal to `num_vertices`, an `out_of_range` exception is thrown.
- ✓ If the weight `w` is less than or equal to 0 or equal to infinity, indicating an invalid weight, an `invalid_argument` exception is thrown.
- ✓ The function updates the adjacency matrix by setting the value at position `m` and `n` as `w`, indicating the edge weight.
- ✓ Since the graph is undirected, it also sets the value at position `n` and `m` as `w`.

PSEUDOCODE LOOKS LIKE:

```
Function insert(m, n, w):
    if m is less than 0 or greater than or equal to num_vertices or n is less than 0
or greater than or equal to num_vertices:
        Throw an "out_of_range" exception with the message "Invalid vertex index"

    If w is less than or equal to 0 or w is equal to infinity:
        Throw an "invalid_argument" exception with the message "Invalid weight"

  Set adjacency_matrix[m][n] to w
  Set adjacency_matrix[n][m] to w
```

❖ double distance(int m, int n) :

- ✓ The `distance`function takes two vertex indices `m` and `n` as input and calculates the shortest distance between them using Dijkstra's algorithm.
- ✓ If either `m` or `n` is less than 0 or greater than or equal to `num_vertices`, an `out_of_range` exception is thrown.
- ✓ If `m` and `n` are the same, indicating the same vertex, the function returns 0.0.
- ✓ The function initializes a vector `dist` of size `num_vertices` with all elements set to infinity, representing the tentative distances from the source vertex.
- ✓ It also initializes a vector `visited` of size `num_vertices` with all elements set to `false`, indicating whether a vertex has been visited or not.
- ✓ The distance of the source vertex (`m`) is set to 0.0 in the `dist` vector.
- ✓ It performs `num_vertices - 1` iterations to process all vertices.
- ✓ In each iteration, it selects the vertex `u` with the minimum distance from the source among the unvisited vertices.
- ✓ It marks the vertex `u` as visited.
- ✓ For each unvisited neighbor vertex `v` of `u` with a finite edge weight, it calculates the new distance `new_dist` by adding the edge weight between `u` and `v` to the distance of `u`.
- ✓ If `new_dist` is smaller than the current distance of `v`, it updates the distance of `v` to `new_dist`.
- ✓ After the iterations, if the distance of the destination vertex (`n`) is still infinity, indicating no path exists between the vertices, a `runtime_error` exception is thrown.
- ✓ Otherwise, it returns the shortest distance between the vertices (`n`) stored in the `dist` vector.]

PSEUDOCODE LOOKS LIKE:

```
Function distance(m, n):
        if m is less than 0 or greater than or equal to num_vertices or n is less
    than 0 or greater than or equal to num_vertices:
            Throw an "out_of_range" exception with the message "Invalid vertex
    index"

        If m is equal to n:
            Return 0.0

        Create a vector dist of size num_vertices filled with infinity
        Create a vector visited of size num_vertices filled with false
        Set dist[m] to 0.0

        For i from 0 to num_vertices - 1:
            Set u to -1
            Set min_dist to infinity

            For each vertex v from 0 to num_vertices - 1:
                If visited[v] is false and dist[v] is less than min_dist:
                    Set u to v
                    Set min_dist to dist[v]

            If u is equal to -1:
                Break the loop

            Set visited[u] to true

            For each vertex v from 0 to num_vertices - 1:
                If visited[v] is false and adjacency_matrix[u][v] is less than infinity:
                    Set new_dist to dist[u] + adjacency_matrix[u][v]

                    If new_dist is less than dist[v]:
                        Set dist[v] to new_dist

        If dist[n] is equal to infinity:
            Throw a "runtime_error" exception with the message "No path exists
    between the vertices"

    Return dist[n]
```

❖ Main function :

- Finally when we come to our main function it serves as the starting point of our program's execution. It demonstrates the functionality of the Weighted_graph class by creating an instance of the class, inserting edges with weights, and calculating the shortest distance between two specified vertices.
  - ✓ It Creates a Weighted_graph object named "graph" with 5 vertices. This initializes the graph with the specified number of vertices, which in this case is 5.
  - ✓ We Inserted edges with weights into the graph using the insert function. This adds edges between vertices with corresponding weights to the adjacency matrix of the graph. In the provided example, six edges with their respective weights are inserted.
  - ✓ It prompts the user to enter the indices of two vertices for which the shortest distance needs to be calculated.
  - ✓ Then reads the values entered by the user into the variables "vertex1" and "vertex2". These values represent the starting and ending vertices for the shortest path calculation.
  - ✓ Finally it attempts to calculate the shortest distance between the given vertices using the distance function of the Weighted_graph class.
    - If the calculation is successful, it assigns the calculated distance to the variable "distance".
    - Then prints the message "Shortest distance between vertices [vertex1] and [vertex2]: [distance]". This displays the calculated shortest distance between the specified vertices.
  - ✓ But if an exception is thrown during the distance calculation, then it executes the corresponding catch block.
    - Print the error message provided by the exception using "e.what()". This displays the specific error message indicating the reason for the exception.
  - ✓ As a final step it ends the main function and returns 0, indicating successful execution of the program.

PSEUDOCODE LOOKS LIKE:

```
Function main():
        Create a Weighted_graph object named "graph" with 5 vertices

        Insert edges with weights into the graph using the insert function:
            Insert an edge between vertex 0 and vertex 1 with weight 4.0
            Insert an edge between vertex 0 and vertex 2 with weight 1.1
            Insert an edge between vertex 1 and vertex 2 with weight 1.9
            Insert an edge between vertex 1 and vertex 3 with weight 3.1
            Insert an edge between vertex 2 and vertex 4 with weight 2.4
            Insert an edge between vertex 3 and vertex 4 with weight 0.5

        Prompt the user to enter the first vertex and store the value in the
    variable "vertex1"
        Prompt the user to enter the second vertex and store the value in the
    variable "vertex2"

        Try to calculate the shortest distance between vertex1 and vertex2 using
    the distance function:
            If successful, assign the calculated distance to the variable "distance"
            Print the message "Shortest distance between vertices [vertex1] and
    [vertex2]: [distance]"
        Catch any exceptions thrown during the distance calculation:
            Print the error message provided by the exception using "e.what()"

    End the main function
```

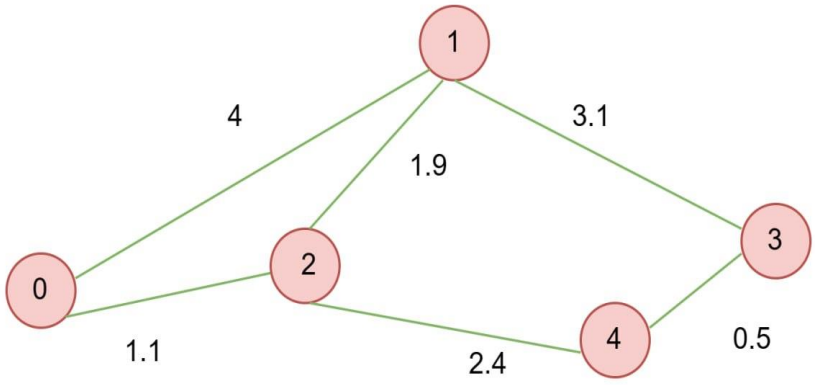
- In summary, the main function demonstrates the usage of the Weighted_graph class by creating a graph, inserting edges with weights, interacting with the user to input vertex indices, and calculating and displaying the shortest distance between the provided vertices using the graph's distance function.

4. CODE IMPLEMETION AND OUTPUT DISPLAY

```
int main() {
    // Create a weighted graph with 5 vertices
    Weighted_graph graph(5);

    // Insert edges with weights
    graph.insert(0, 1, 4.0);
    graph.insert(0, 2, 1.1);
    graph.insert(1, 2, 1.9);
    graph.insert(1, 3, 3.1);
    graph.insert(2, 4, 2.4);
    graph.insert(3, 4, 0.5);
```

- After this insertion the following graph with 5 vertices/nodes/ is created:

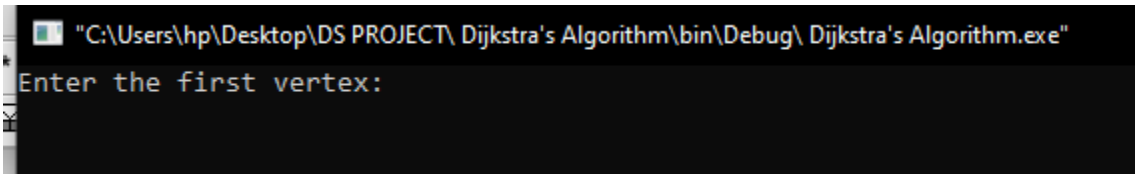


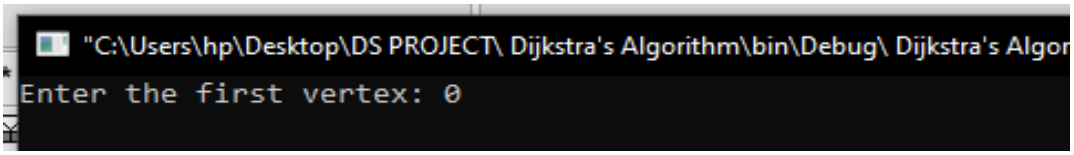
- Then it prompts the user to enter the two vertices for which the distance is to be calculated for with the following lines

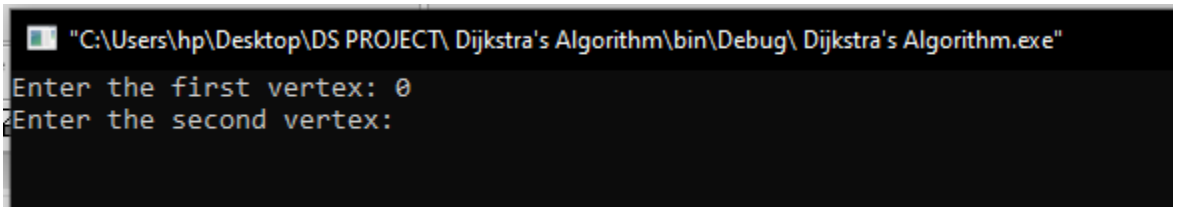
```
// Prompt the user for vertex inputs
int vertex1, vertex2;
cout << "Enter the first vertex: ";
cin >> vertex1;
cout << "Enter the second vertex: ";
cin >> vertex2;
```

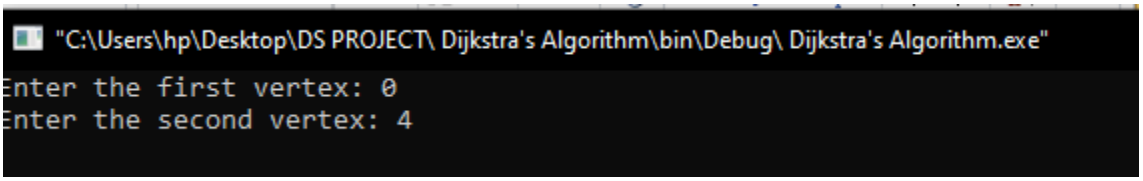
❖ CASE 1 – FOR A VALIED INPUT

- Let's say we want to know the shortest distance between vertex 0 and 4,
  - ✓ After running the code the following console pops up requesting the user to input the first vertex , we enter 0 then 4 as shown below

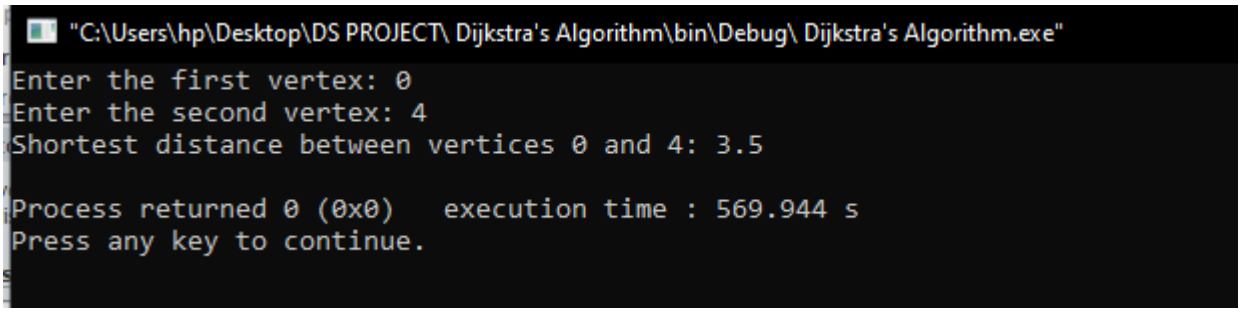








  - ✓ After taking the two input vertices namely (0 and 4) the calculated result is displayed as follows ,



- Hence the shortest distance between vertices 0 and 4 is 3.5

❖ CASE 2 – FOR AN INVALID INPUT

- Let's say a user gave as an input a vertex that doesn't exist say vertex 5, the following output will be displayed.
  - For instance for input vertex 0 and 5 since 5 doesn't exist and error message will be displayed.

```
"C:\Users\hp\Desktop\DS PROJECT\ Dijkstra's Algorithm\bin\Debug\ Dijkstra's Algorithm.exe"
Enter the first vertex: 0
Enter the second vertex: 5
Error: Invalid vertex index

Process returned 0 (0x0)   execution time : 3.833 s
Press any key to continue.
```

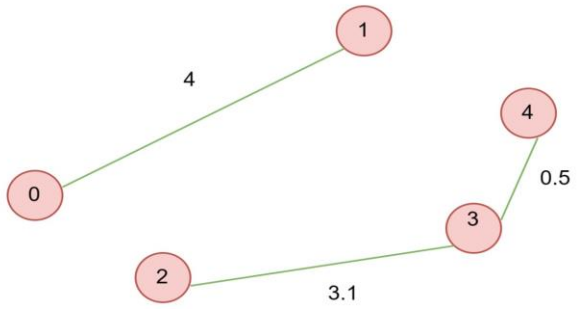
CASE 3 - WHEN THERE IS NO CONNECTION WITH THE VERTICES

➢ Now for another special case where there is no connection path between the vertices let's we made the following insertion and created the following graph

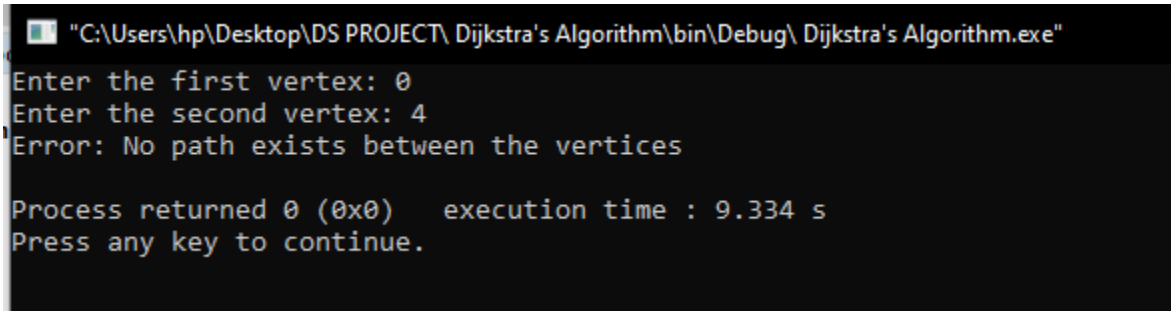
```
int main() {
    // Create a weighted graph with 5 vertices
    Weighted_graph graph(5);

    // Insert edges with weights
    graph.insert(0, 1, 4.0);
    graph.insert(2, 3, 3.1);
    graph.insert(2, 4, 2.4);
```

➢ After this insertion the following graph with 5 vertices/nodes/ is created:



- If we took vertices 0 and 4 as an input an error message saying that no path exists would be displayed as follows:



5. APPENDICES

```
#include <iostream>
#include <vector>
#include <stdexcept>
#include <limits>
   using namespace std;
class Weighted_graph {
private:
   int num_vertices;
   vector<vector<double>> adjacency_matrix;
public:
   Weighted_graph(int n = 50) {
      if (n <= 0)
         n = 1;
      num_vertices = n;
      adjacency_matrix.resize(n,vector<double>(n,numeric_limits<double>::infinity()));
   }

   ~Weighted_graph() {
      adjacency_matrix.clear();
   }

   int degree(int n) const {
      if (n < 0 || n >= num_vertices)
```

```
        throw out_of_range("Invalid vertex index");

    int degree = 0;
    for (int i = 0; i < num_vertices; ++i) {
        if (i != n && adjacency_matrix[n][i] < numeric_limits<double>::infinity())
            degree++;
    }
    return degree;
}

int edge_count() const {
    int count = 0;
    for (int i = 0; i < num_vertices; ++i) {
        for (int j = i + 1; j < num_vertices; ++j) {
            if (adjacency_matrix[i][j] <numeric_limits<double>::infinity())
                count++;
        }
    }
    return count;
}

double adjacent(int m, int n) const {
    if (m < 0 || m >= num_vertices || n < 0 || n >= num_vertices)
        throw out_of_range("Invalid vertex index");
```

```
    if (m == n)
        return 0.0;
    return adjacency_matrix[m][n];
}

void insert(int m, int n, double w) {
    if (m < 0 || m >= num_vertices || n < 0 || n >= num_vertices)
        throw out_of_range("Invalid vertex index");

    if (w <= 0 || w == numeric_limits<double>::infinity())
        throw invalid_argument("Invalid weight");

    adjacency_matrix[m][n] = w;
    adjacency_matrix[n][m] = w;
}

double distance(int m, int n) {
    if (m < 0 || m >= num_vertices || n < 0 || n >= num_vertices)
        throw out_of_range("Invalid vertex index");

    if (m == n)
        return 0.0;
    vector<double> dist(num_vertices, numeric_limits<double>::infinity());
    vector<bool> visited(num_vertices, false);
```

```
dist[m] = 0.0;
for (int i = 0; i < num_vertices - 1; ++i) {
   int u = -1;
   double min_dist = numeric_limits<double>::infinity();
   for (int v = 0; v < num_vertices; ++v) {
      if (!visited[v] && dist[v] < min_dist) {
         u = v;
         min_dist = dist[v];
      }
   }
   if (u == -1)
      break;
   visited[u] = true;
   for (int v = 0; v < num_vertices; ++v) {
      if (!visited[v] && adjacency_matrix[u][v] < numeric_limits<double>::infinity()) {
         double new_dist = dist[u] + adjacency_matrix[u][v];
         if (new_dist < dist[v])
            dist[v] = new_dist;
      }
   }
}
```

```
            // Check if no path exists between vertices m and n
        if (dist[n] == numeric_limits<double>::infinity())
            throw runtime_error("No path exists between the vertices");


        return dist[n];
    }
};
int main() {
    // Create a weighted graph with 5 vertices
    Weighted_graph graph(5);

    // Insert edges with weights
    graph.insert(0, 1, 4.0);
    graph.insert(0, 2, 1.1);
    graph.insert(1, 2, 1.9);
    graph.insert(1, 3, 3.1);
    graph.insert(2, 4, 2.4);
    graph.insert(3, 4, 0.5);
    // Prompt the user for vertex inputs
    int vertex1, vertex2;
    cout << "Enter the first vertex: ";
    cin >> vertex1;
    cout << "Enter the second vertex: ";
    cin >> vertex2;
```

```
    // Calculate the shortest distance between the given vertices
    try {
        double distance = graph.distance(vertex1, vertex2);
        cout << "Shortest distance between vertices " << vertex1 << " and " << vertex2 << ": " <<
distance << endl;
    } catch (const exception& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```