**SPL-1 Project Report**

**TravelPlanner**

Submitted by

***Redowan Billah Sami***

**BSSE Roll No. : 1409**

**BSSE Session: 2021-2022**

Submitted to

***Toukir Ahammed***

**Lecturer**

**Institute of Information Technology**

**Supervisor’s Approval: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**(Signature)**

****

**Institute of Information Technology**

**University of Dhaka**

[15-12-2023]

# Introduction……………………………………………………………………………………………………………………1

# Background…………………………………………………………………………………………………………………….2

# Description of the Project………………………………………………………………………………………………3

# Implementation and Testing……………………………………………………………………………………………4

4.1 Taking the input of graph…………………………………………………………………………………4

4.1.1 Opening File and Taking inputs……………………………………………………………………4

4.1.2 Calculating Distance and Storing…………………………………………………………………5

4.1.3 Remove edges………………………………………………………………………………………………5

4.2 Viewing the Location……………………………………………………………………………………………………6

4.3 View the Graph……………………………………………………………………………………………………………6

4.3.1 Calculating Coordinates…………………………………………………………………………………6

4.3.2 Bangladesh Map………………………………………………………………………………………………7

4.3.3: Visualizing the Graph……………………………………………………………………………………7

4.3.4 Visualizing the derived path…………………………………………………………………………7

4.3.5 Visualizing the derived edges………………………………………………………………………8

4.4 BFS Algorithm………………………………………………………………………………………………………………9

4.4.1 Creating Queue, initialization and isEmpty function……………………………………9

4.4.2 Enqueue…………………………………………………………………………………………………………9

4.4.3 Dequeue…………………………………………………………………………………………………………9

4.4.4 BFS function…………………………………………………………………………………………………10

4.5 Dijkstra’s Algorithm……………………………………………………………………………………………………11

4.5.1 Initializations…………………………………………………………………………………………………11

4.5.2 Find Minimum Distance ………………………………………………………………………………11

4.5.3 print\_path function………………………………………………………………………………………12

4.6 TSP solution…………………………………………………………………………………………………………………13

4.6.1 Initialization…………………………………………………………………………………………………13

4.6.2 tsp function……………………………………………………………………………………………………13

4.6.3 print\_tsp\_path function………………………………………………………………………………14

4.7 Kruskal’s Algorithm……………………………………………………………………………………………………15

4.7.1 Declaring Structures and lists……………………………………………………………………15

4.7.2 sort\_edge, find and applyUnion function……………………………………………………15

4.7.3 Printing and storing optimal edges……………………………………………………………16

4.7.4 kruskal function……………………………………………………………………………………………16

4.8 Prim’s Algorithm……………………………………………………………………………………………………………17

4.8.1 Initialization…………………………………………………………………………………………………17

4.8.2 Finding Minimum Weight Path……………………………………………………………………18

# User Interface………………………………………………………………………………………………………………19

# 5.1 Load Map…………………………………………………………………………………………………………19

# 5.2 View the Locations…………………………………………………………………………………………20

# 5.3 View the Graph…………………………………………………………………………………………………20

# 5.4 View Path…………………………………………………………………………………………………………21

# 5.5 View the Shortest Path……………………………………………………………………………………21

# 5.6 Minimum Cost Path…………………………………………………………………………………………23

# 5.7 View the route with minimum distance back to starting point………………………23

# 5.8 Paths with Minimum Distance to visit all Locations………………………………………24

# 5.9 Paths with Minimum Distance to visit all Locations(From a start point)………24

# 5.10 Remove Path…………………………………………………………………………………………………25

# 5.11.Exit…………………………………………………………………………………………………………………25

# Challenges Faced…………………………………………………………………………………………………………25

# Conclusion………………………………………………………………………………………………………………………26

# Reference…………………………………………………………………………………………………………………………27

Index of Figures

Figure 1: Opening file………………………………………………………………………………………………………… 4

Figure 2: Storing values from file……………………………………………………………………………………… 5

Figure 3: Calculating distance from coordinates……………………………………………………………… 5

Figure 4: Storing the Distance in graph array…………………………………………………………………. 5

Figure 5: Remove Edges…………………………………………………………………………………………………… 6

Figure 6: View the Locations……………………………………………………………………………………………… 6

Figure 7: Transforming Coordinates…………………………………………………………………………………… 6

Figure 8: Drawing the border……………………………………………………………………………………………… 7

Figure 9: Visualization of the graph…………………………………………………………………………………… 7

Figure 10: Visualizing Source and Destination……………………………………………………………………8

Figure 11: Visualizing the Path…………………………………………………………………………………………… 8

Figure 12: Visualizing the resultant edges………………………………………………………………………… 8

Figure 13: Queue,initQueue,isEmpty…………………………………………………………………………………. 9

Figure 14: Enqueue………………………………………………………………………………………………………………9

Figure 15: Dequeue……………………………………………………………………………………………………………10

Figure 16: BFS Queue and initialization……………………………………………………………………………10

Figure 17: BFS solution and Path………………………………………………………………………………………10

Figure 18: Cost Array Initialization……………………………………………………………………………………11

Figure 19: Initialize dis,prev and visited arrays……………………………………………………………….11

Figure 20: Find Minimum Distance……………………………………………………………………………………12

Figure 21: Print Minimum Values………………………………………………………………………………………12

Figure 22: print\_path function……………………………………………………………………………………………12

Figure 23: TSP initialization……………………………………………………………………………………………….13

Figure 24: TSP function………….…………………………………………………………………………………………14

Figure 25: print\_tsp\_path function……………………………………………………………………………………14

Figure 26: edge, edge\_list structure and list……………………………………………………………………15

Figure 27: sort\_edge, find function……………………………………………………………………………………15

Figure 28: applyUnion function…………………………………………………………………………………………16

Figure 29: Printing and storing edges………………………………………………………………………………16

Figure 30: Kruskal function-1……………………………………………………………………………………………16

Figure 31: Kruskal function-2……………………………………………………………………………………………17

Figure 32: Prim’s Initialization……………………………………………………………………………………………17

Figure 33: Finding Min Weight Path…………………………………………………………………………………..18

Figure 34: Menu…………………………………………………………………………………………………………………19

Figure 35: LoadMap output-1……………………………………………………………………………………………19

Figure 36: LoadMap output-2……………………………………………………………………………………………19

Figure 37: Location Output…………………………………………………………………………………………………20

Figure 38: Bangladesh Map……………………………………………………………………………………………… 20

Figure 39: View Path input…………………………………………………………………………………………………21

Figure 40: View Path output………………………………………………………………………………………………21

Figure 41: Shortest path input-1………………………………………………………………………………………21

Figure 42: Shortest path output-1……………………………………………………………………………………22

Figure 43: Shortest path output-2……………………………………………………………………………………22

Figure 44: Min cost output…………………………………………………………………………………………………23

Figure 45: Output of 5.7……………………………………………………………………………………………………23

Figure 46: Kruskal output map…………………………………………………………………………………………24

Figure 47: Prims output map……………………………………………………………………………………………24

Figure 48: Remove path……………………………………………………………………………………………………25

# 

# Introduction

The necessity for effective and optimized routes is critical in today's increasingly interconnected world, whether for leisure travel, business, or exploration. The objective of this project is to use graph theory to solve a variety of issues related to travel planning and visualize the solution for a better user experience.

Inappropriate travel planning can have far-reaching consequences, ranging from increased travel time to increased logistical expense. One of the major risks of inadequate planning is the potential waste of resource. Inefficient routes and lack of optimization can lead to unnecessary fuel consumption, increased transportation costs, and extended travel times. This not only impacts the financial bottom line but also contributes to environmental concerns through heightened carbon emissions.

People face a variety of difficulties while travelling such as finding the shortest path between locations, determining an optimal route that covers all desired destinations and returns to the starting point, constructing a tree-alike network for efficient connectivity, and identifying cost-efficient paths. The Travel Planner Project addresses these challenges by implementing a suite of graph algorithms (e.g. Dijkstra’s Algorithm has been used to find the shortest path from source to destination.). Through visualizations integrated into the project, users gain a clear and intuitive understanding of the optimal solutions generated by these algorithms. A visual representation of these routes helps the users to foresee potential challenges in their travel plans and make informed decisions.

Key benefits presented by this project are described below:

1. Efficient Planning: This project equips users with a strategic advantage in organizing their travel plans. With the help of advanced algorithm and visualization of the optimal solution, the user can enjoy an efficient plan saving valuable time and resources.
2. Optimized Resource Utilization: With the help of this project, users can minimize unnecessary expenses, such as fuel or tolls, while maximizing the overall efficiency of the travel plan.
3. Increased Capability for Making Decisions: By acquiring insights into the best possible solutions produced by the algorithms, users will be able to make better decisions. The visual representations enable a deeper understanding, aiding users in making well-informed choices.
4. Stress-Free Travel Planning: With the help of advanced algorithms and user-friendly visualizations, users can navigate the planning process with ease. Users can experience a hassle-free journey from conceptualizing routes to making well-informed decisions. This stress-free approach allows users to focus on the excitement of their upcoming travels rather than the complexities of planning, providing a more enjoyable and relaxed experience.

In short, this project will enable the user to enjoy a stress-free travel by using different graph algorithms and visualize the path for a better overview.

# Background

Some key terms should be clarified for a better understanding of the entire project-

1. Geographical Coordinate System: The geographic coordinate system is a spherical or geodetic coordinate system for measuring and communicating positions directly on the Earth as latitude and longitude. Latitude and longitude are coordinates that show the angular distance of any place from the Earth's center. A global address can be given to every location by using Latitude and longitude coordinates.
2. Bitmasking: Bit masking is a technique to manipulate individual bits within a binary number. In this technique, bitwise operations, such as AND, OR, XOR, and NOT, are applied to specific bits in a binary representation. Bit masking is often used in scenarios where individual flags need to be manipulated efficiently.

# Description of the Project

The project begins with a menu which takes the user input and work accordingly. The menu contains 11 options i.e. “1. Load Map, 2. View the Locations, 3. View the Graph, 4. View Path, 5. View the Shortest Path, 6. Minimum Cost Path, 7. View the route with minimum distance back to starting point, 8. Paths with Minimum Distance to visit all Locations, 9. Paths with Minimum Distance to visit all Locations(From a start point), 10. Remove Path, 11. Exit”. After selecting the desired option, the user have to give appropriate inputs to get the optimal solutions.

* Load Map: If the user gives the input 1, Load Map option will be selected. The program will ask the user to input a file name. If the file name is valid, map will be loaded. The distances will be calculated using CalcDistance() function.
* View the Locations: After loading the map, the user can view the locations by choosing this option.
* View the Graph: User can view the map by choosing this option after loading the map.
* View Path: In this option, the prompt will wait for the user input. It will ask the user to give the name of source and destination. The it will perform BFS(Breadth First Search), find a path and print it. The user can also view the graph by giving “Y” in input.
* View the Shortest Path: In this option, the prompt will ask the user to give the name of source and destination. The it will perform Dijkstra’s algorithm, find a path and print it. The user can also view the graph by giving “Y” in input.
* Minimum Cost Path: This will work as the same procedure as the program used to find the Shortest Path with a few additional things. In this option, the cost will be calculated 2taka/km. The minimum cost path will be printed and the user will be able to view the graph with the total expense in each step.
* View the route with minimum distance back to starting point: In this option, the user needs to input a starting location. This problem is same as the TSP(Travelling Salesman Problem). So, Branch and Bound algorithm will be used and output will be generated. The user can also view the graph by giving “Y” in input.
* Paths with Minimum Distance to visit all Locations: In this option, Kruskal’s algorithm will be applied and output will be shown both in console and graph.
* Paths with Minimum Distance to visit all Locations(From a start point): The user will have to choose a starting point. The Prim’s algorithm will be applied and corresponding path will be printed and shown the graph.
* Remove Path: The user can remove any path by choosing this option.(A path can be unusable due to accident, blockage or other problems). The program will ask to input the starting and ending point of the path, then remove it. The user can remove multiple path by choosing “YES” after the console asks for it.
* Exit: This will terminate the program.

# Implementation and Testing

[Insert implementation details and testing details (if applicable). You may add tables/diagrams/ code snippet if needed.]

Different functions and algorithms were applied in this project. So the project can be separated in the following tasks-

* Taking the input of graph
* Viewing the location
* View the graph
* BFS Algorithm
* Dijkstra’s Algorithm
* TSP solution
* Kruskal’s Algorithm
* Prim’s Algorithm
* Remove Edge
* Exit

**4.1 Taking the input of graph**

**4.1.1 Opening File and Taking inputs**

In this part, the graph is initialized by opening and reading a file.

The user will have to provide the appropriate file name. If the name is not appropriate the function will return to menu.

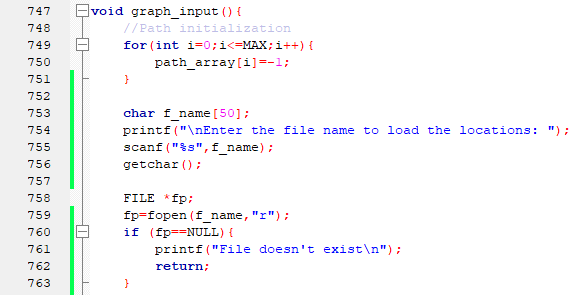


Figure 1: Opening file

If the file is opened successfully, it will read the value of n, place,x\_cor(latitude) and y\_cor(longitude). At last it will print \*\*\*MAP LOADED\*\*\*

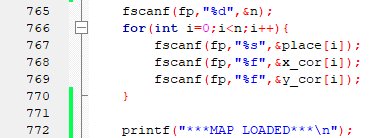


Figure 2: Storing values from file

**4.1.2 Calculating Distance and Storing**

Then the distance between the coordinates will be calculated from the below function-

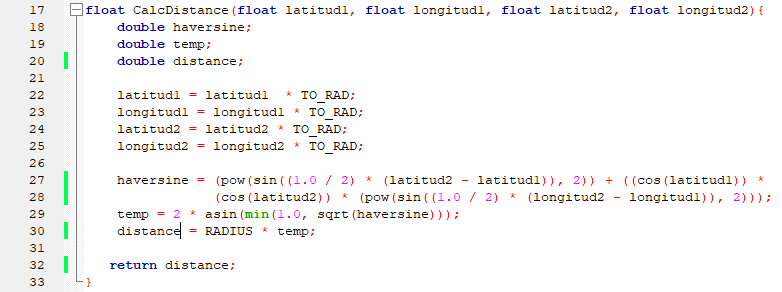


Figure 3: Calculating distance from coordinates

The function above is used to calculate the distance between two Geographical Coordinates.

The value of the distances will be stored in the graph array.

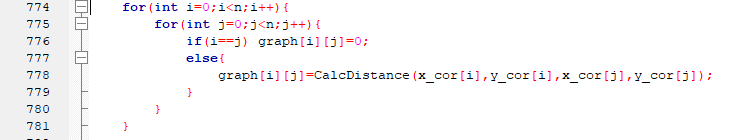


Figure 4: Storing the Distance in graph array

**4.1.3 Remove edges**

The paths that cannot be connected directly because of international border will be removed by replacing them with a larger value.



Figure 5: Remove Edges

**4.2 Viewing the Location:**



Figure 6: View the Locations

Locations along with their index will be printed in this option

**4.3 View the Graph:**

**4.3.1 Calculating Coordinates:**

This code snippet takes a set of coordinates, normalizes them by subtracting the minimum values and scaling, and then performs a coordinate swap and vertical flip to prepare them for rendering in a graphics window. This transformation is required as we change Cartesian coordinates into graphics screen suitable coordinates.

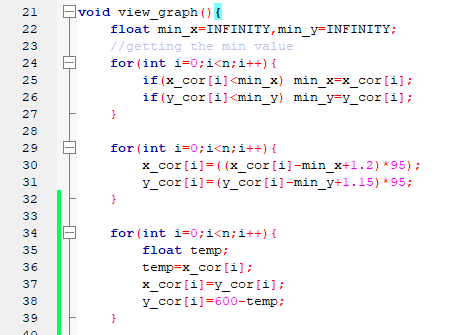


Figure 7: Transforming Coordinates

**4.3.2 Bangladesh Map:**

After that we open the graphics window. Border of Bangladesh is drawn manually with the help of lines.

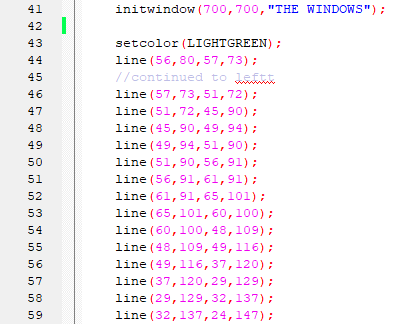


Figure 8: Drawing the border

**4.3.3: Visualizing the Graph**

Circles and lines are drawn to represent the locations and their connecting path. Name of the location is also printed.

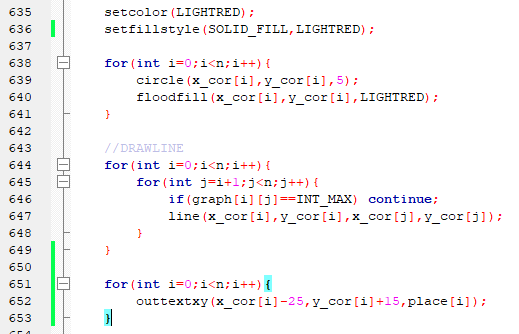


Figure 9: Visualization of the graph

**4.3.4 Visualizing the derived path**

This code snippet marks the source and destination locations if any path exists. It also prints STARTED FROM and the location from where the path started. This path is used in BFS, Dijkstra’s Algorithm and TSP solution.

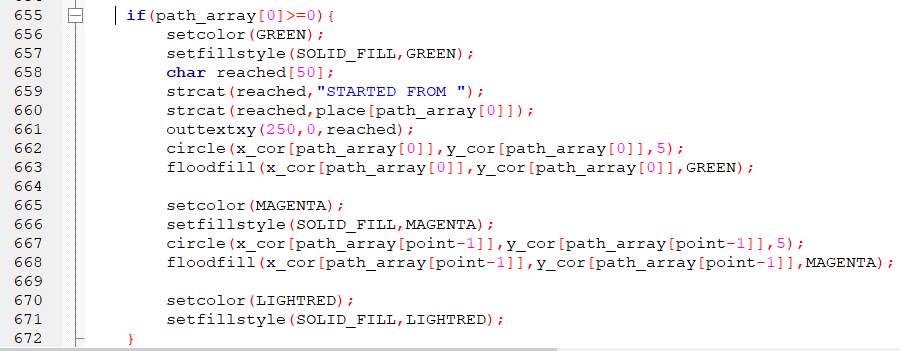


Figure 10: Visualizing Source and Destination

This code snippet prints the entire path from source to destination and represents it graphically. It also prints the total cost if Minimum Cost Path option is selected.

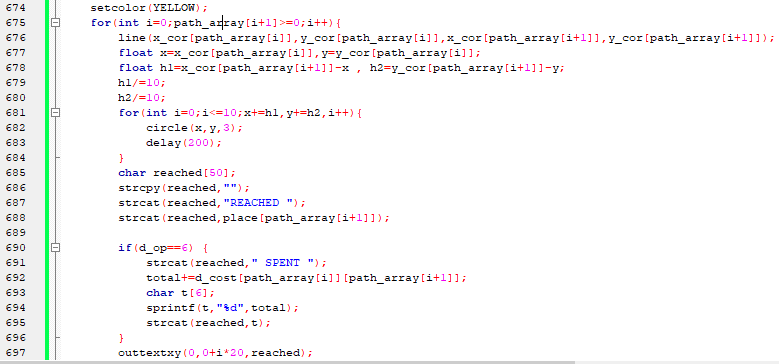


Figure 11: Visualizing the Path

**4.3.5 Visualizing the derived edges**

If there are any edges that need to be printed, this portion of code performs that task. Finally the graph is closed with closegraph();

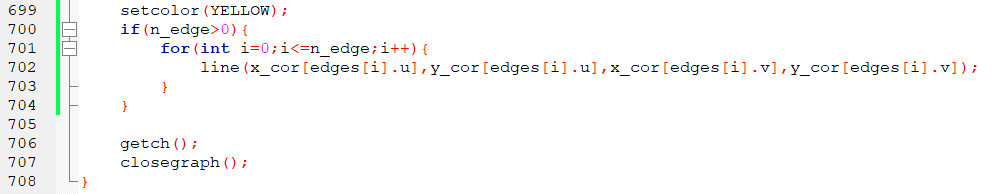


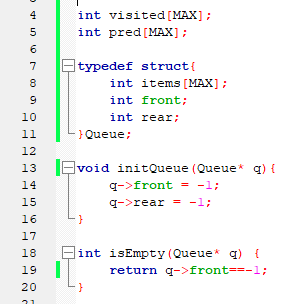
Figure 12: Visualizing the resultant edges

**4.4 BFS Algorithm**

BFS is a traversing algorithm where we start traversing from a source node and traverse the graph layer-wise thus exploring the neighbor nodes (nodes which are directly connected to source node). Then we move towards the next-level neighbor nodes until the destination node is found.

**4.4.1 Creating Queue, initialization and isEmpty function**

First we create two array of MAX size, visited to mark which node is visited and pred array to indicate the parent of a specific node. We create a structure of Queue, which contains front, rear and array of nodes. initQueue function is used to initialize the queue. isEmpty function checks whether the queue is empty.

****Figure 13: Queue,initQueue,isEmpty

**4.4.2 Enqueue**

Enqueue function is used to push a node in the queue. If the queue is full it returns, otherwise it adds the node in the queue.

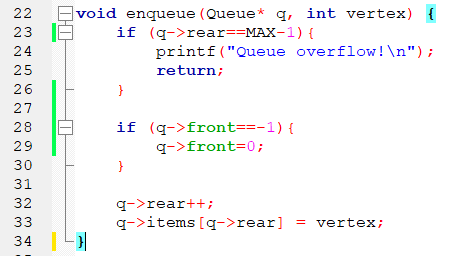
****

Figure 14: Enqueue

**4.4.3 Dequeue**

Dequeue function removes the node in the front and increases the value of front. If the queue is empty it returns -1. If the front value exceeds rear value(queue is empty), front and rear are both set to -1.

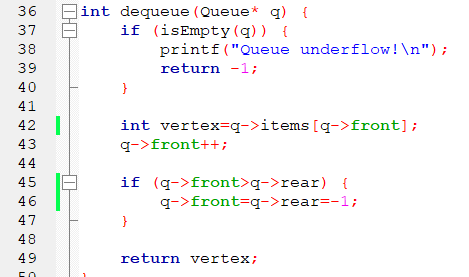
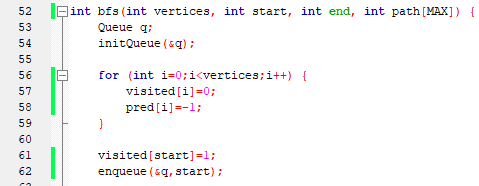


Figure 15: Dequeue

**4.4.4 BFS function**

In the bfs function, a queue is created and initialized. A loop is used to initialize the values of visited and pred array. Then we mark the starting node as visited and enqueue it.

  
Figure 16: BFS Queue and initialization

Then we start a loop which runs until the queue is empty. The value at the front of the queue is stored in u. Then we explore the neighbors of u in the following for loop. if v is an unvisited neighbor of u, it is marked as visited, u is stored as its predecessor and v is enqueued. If the v is the destination node, we reconstruct the path using the values of pred array. After the source node is reached, we reverse the path to get the output format,(source to destination). Then the function returns the path length.

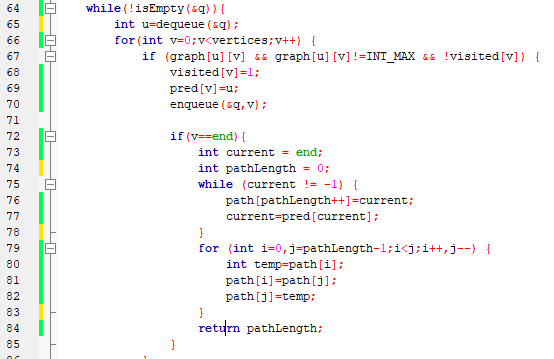


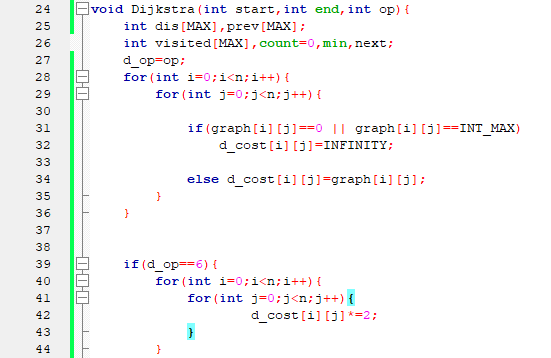
Figure 17: BFS solution and Path

**4.5 Dijkstra’s Algorithm**

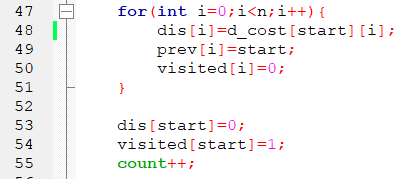
Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a non-negative weighted graph. This algorithm is used in GPS devices to find the shortest path between the current location and the destination. This algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

**4.5.1 Initializations**

In the given function below, we get start node, end node and op as parameter. op determines whether it is the shortest path problem or the minimum budget problem. Dis, prev and visited arrays are declared to calculate distance, store the previous node and mark as visited respectively. Other necessary variables are also declared. A loop is started to store the necessary values in the d\_cost array. If there exists a path and the value of graph is stored in the d\_cost 2D array. Otherwise it stores INFINITY. For minimum cost we double the value as we calculate the cost 2taka/km.

  
Figure 18: Cost Array Initialization

Then we initialize the values of dis,prev and visited array. For starting, we set dis of starting node as 0, mark it as visited, increase the count by one.

  
Figure 19: Initialize dis,prev and visited arrays

**4.5.2 Find Minimum Distance**

A while loop runs until the count is less than n-1, where n are the number of vertices. In an undirected connected graph with n vertices, the shortest path tree will have at most n-1 edges because each vertex, except the source vertex, is connected to the tree by exactly one edge. Therefore, the algorithm ensures that it considers all vertices and includes them in the shortest path tree before terminating.

INFINITY is stored in min variable. A loop is started to find an unvisited vertex with minimum distance. The selected vertex is marked as visited. Then we check the condition (min+d\_cost[next][i] < dis[i]) for any unvisited vertex i. If the condition is satisfied, dis and prev arrays are updated. count is increased.

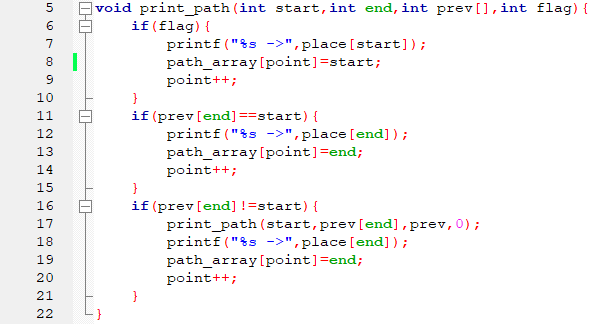
  
Figure 20: Find Minimum Distance

The following code snippet prints the desired output according to the op given in the parameter.

  
Figure 21: Print Minimum Values

**4.5.3 print\_path function**

print\_path function is used to print the path and store the path in the path\_array with the help of point. While calling the function, flag will be 1, so start will be printed and stored in path\_array. If the previous of end is equal end point is printed. Otherwise it will call the function recursively with flag 0.

  
Figure 22: print\_path function

**4.6 TSP solution**

Traveling Salesman Problem (TSP) is solved here using dynamic programming with memoization.

**4.6.1 Initialization**

dp and parent arrays and visited\_ALL variable is declared at first. Then the values are initialized using the tsp\_initials function. “visited\_ALL” is a global variable used as a bitmask to represent all vertices as visited. dp[i][j] is an array used for memoization. It stores the minimum cost of visiting all cities starting from vertex i and with the set of visited vertices represented by the bitmask j. The initialization sets all entries in the dp array to -1, indicating that the corresponding subproblems have not been solved yet.

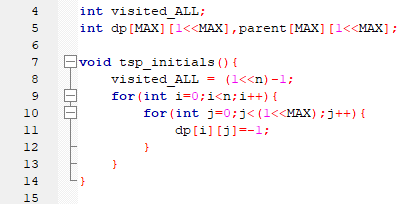


Figure 23: TSP initialization

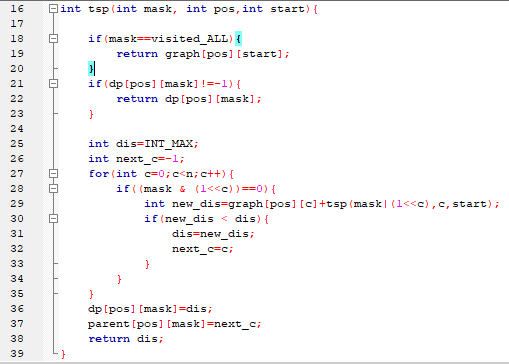
**4.6.2 tsp function**

At first, two base cases are checked, If all cities have been visited (mask == visited\_ALL), it will return the cost of returning from the current city (pos) to the starting city (start). If the result for the current subproblem is already memoized (dp[pos][mask] != -1), return the memoized result.

Then dis and next\_c variables are initialized. The function iterates through all nodes (c) to explore the possible next nodes to visit. The condition (mask & (1 << c)) == 0 checks if the node c has not been visited yet. If the condition is true we calculate new\_dis. new\_dis is the tentative new distance if we visit city c from the current city pos. It is calculated as the sum of the cost of traveling from pos to c (graph[pos][c]) and the result of the recursive call to tsp with the updated bitmask (mask | (1 << c)), indicating that city c is now visited. If the new\_dis<dis, dis is updated to new\_dis, and the index of the next city in the optimal path (next\_c) is set to c.

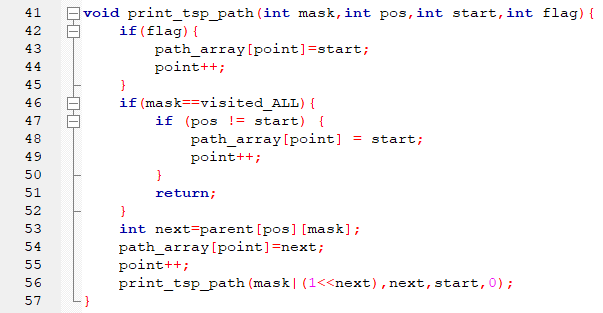
The dynamic programming (DP) table is updated at position pos and with the bitmask mask with the minimum distance. dis represents the minimum distance found for the current subproblem, i.e., the minimum cost of visiting all cities starting from the current city pos and with the set of visited cities represented by the bitmask mask. Then the parent array is updated at position pos and with the bitmask mask. next\_c represents the index of the next city in the optimal path for the current subproblem.

dis is returned.

  
Figure 24: TSP function

**4.6.3 print\_tsp\_path function**

Then the path is printed using the below function. The flag is used to indicate the start city. If flag is true start city is added in the path. The next condition checks if all cities have been visited. Pos!=start ensures the starting city is not repeated at the end of the path. If the does not return in the if codes block, the function proceeds to update the path\_array, with the index of the next city in the optimal path (next). The bitmask is updated by setting the bit corresponding to the next city using (mask | (1 << next)). The function then makes a recursive call with the updated bitmask, the next city as the new position, the starting city, and flag set to 0.

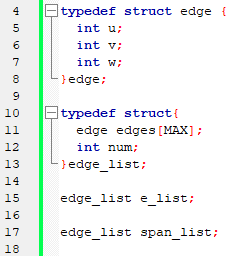
  
Figure 25: print\_tsp\_path function

**4.7 Kruskal’s Algorithm**

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

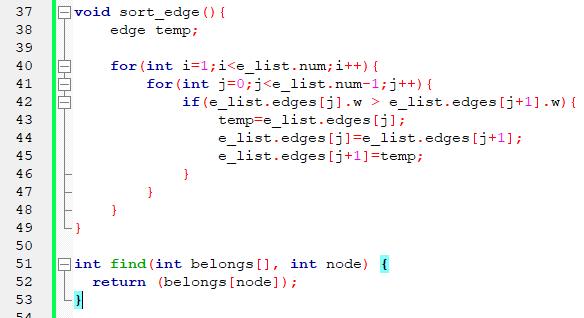
**4.7.1 Declaring Structures and lists**

Structure of edge and edge\_list are declared. Two edge\_list is created named e\_list and span\_list.

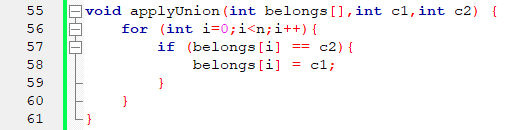
  
Figure 26: edge, edge\_list structure and list

**4.7.2 sort\_edge, find and applyUnion function**

“sort\_edge” function sorts the edges in a non-decreasing order of weight. “find” function is used to find the subset (set to which a node belongs) of a given node in a disjoint-set data structure. The function takes an array belongs[], which represents the disjoint-set data structure, and an integer node representing the node whose subset is to be found. It returns the subset (representative) to which the given node belongs.

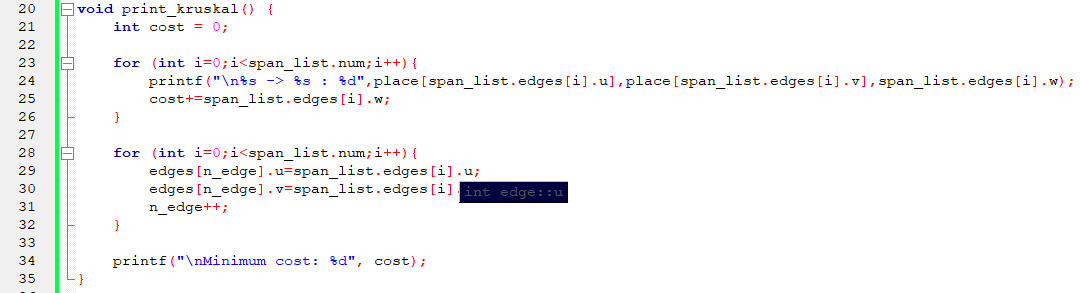
  
Figure 27: sort\_edge, find function

applyUnion function iterates through each element of the belongs array. If an element currently belongs to the subset represented by c2 (which is being merged into c1), it updates that element to belong to the subset represented by c1.

  
Figure 28: applyUnion function

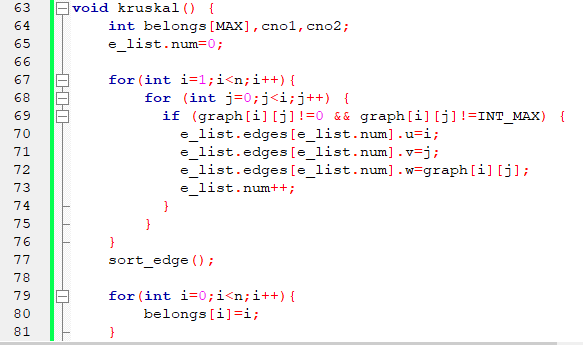
**4.7.3 Printing and storing optimal edges**

The optimal edges are stored and printed using the below function:

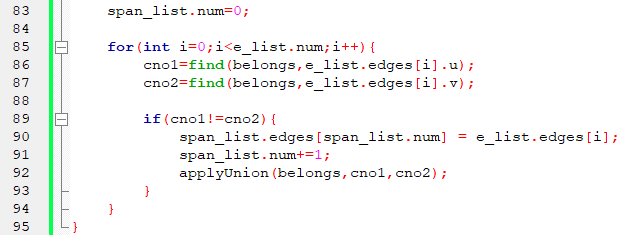
  
Figure 29: Printing and storing edges

**4.7.4 kruskal function**

In the kruskal function, necessary variables are declared at first. All the edges are inserted into e\_list using loop. The edges are then sorted. The next for loop initializes subset of each node such that a node is a member of its own subset

  
Figure 30: Kruskal function-1

The for loop iterates through each edge in the sorted list of edges e\_list. For each edge, it finds the subsets (or components) to which the two endpoints (u and v) of the edge belong. This is done using the find function(cno1 represents the subset to which e\_list.edges[i].u belongs). If the two endpoints do not belong to the same subset (i.e., they are not in the same connected component), it means adding the edge will not create a cycle in the minimum spanning tree. In that case the edge is added to the span\_list. The edge is placed in the span\_list.edges array at index span\_list.num. span\_list.num is then incremented. The applyUnion function is called to merge the subsets to which the two endpoints belong. This ensures that the two connected components are now treated as one connected component in the disjoint-set data structure.

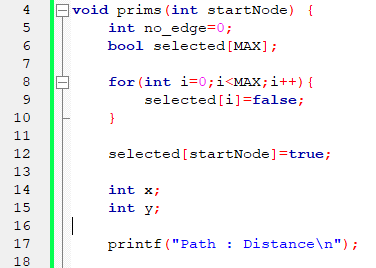
  
Figure 30: Kruskal function-2

**4.8 Prim’s Algorithm**

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. This algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph gets selected.

**4.8.1 Initialization**

The function takes startNode as parameter. no\_edge refers to the number of edges in the MST. selected array keeps the track of selected nodes. The for loop initializes the selected array as false. Startnode is setted as selected. Variable x and y are declared for further use.

  
Figure 32: Prim’s Initialization

**4.8.2 Finding Minimum Weight Path**

The while loop iterates until no\_edges has n-1 edges. The for loop iterates through the selected nodes. The nested for loop iterates through the non-selected nodes, and if there is a path from the selected node to non-selected node. Then the minimum weight edge is searched and endpoint x,y are updated. Then the edge with minimum weight is stored and printed. Mark the endpoint node as selected and increment the no\_edge.

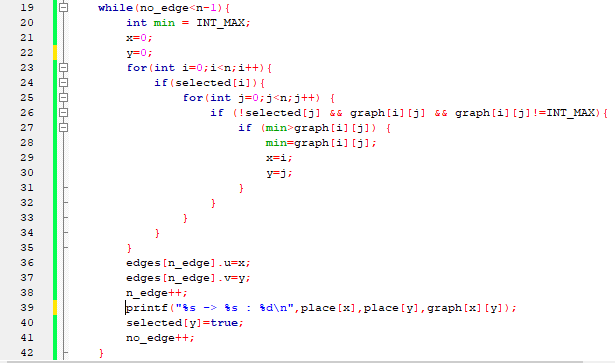


Figure 33: Finding Min Weight Path

# User Interface

# Menu

# The program will show the given menu and ask the user what task to perform.

# Figure 34: Menu

# 5.1 Load Map

# If 1 is given as input, the prompt will ask for a file name to load the map. The user must load a map to perform next tasks.

# Sample Input 1: BD.txt

# Output:

# Figure 35: LoadMap output-1

# Sample Input 2: xyz.txt (invalid filename)

# Output:

# Figure 36: LoadMap output-2

# 5.2 View the Locations:

# After loading the map, the user will be able to see the loaded locations along with their index

# Sample input: 2

# Output:

# Figure 37: Location Output

# 5.3 View the Graph

# After loading the map, the user can see the visual representation of the graph:

# Sample Input: 3

# Output:

# Figure 38: Bangladesh Map

# 5.4 View Path

# If 4th option is selected, the user will have to provide the source and destination. The user can view the graph by giving “Y” when the programs asks for it.

# Sample Input: DHAKA CHITTAGONG Y

# Figure 39: View Path input

# Sample Output:

# Figure 40: View Path output

# 5.5 View the Shortest Path

# If 5th option is selected, the user will have to provide the source and destination. The user can view the graph by giving “Y” when the programs asks for it.

# Sample Input 1: SYLHET CHITTAGONG Y

# Figure 41: Shortest path input-1

# Output 1:

# 

# Figure 42: Shortest path output-1

# Sample Input 2: DHAKA KHULNA N

# Output 2:

# Figure 43: Shortest path output-2

# 5.6 Minimum Cost Path

# If 6th option is selected, the user will have to provide the source and destination. The user can view the graph along with total cost by giving “Y” when the programs asks for it.

# Sample Input: MYMENSHING RANGPUR Y

# Output:

# 

# Figure 44: Min cost output

# 5.7 View the route with minimum distance back to starting point

# If 7th option is selected, the user will have to provide a source. The user can view the graph along with total cost by giving “Y” when the programs asks for it.

# Sample Input: DHAKA Y

# Output:

# Figure 45: Output of 5.7

# 5.8 Paths with Minimum Distance to visit all Locations

# After loading the map if 8 is given as input, Minimum Distance paths will be printed to visit all locations and represented in the graph.

# Sample Input: 8

# Output:

# Figure 46: Kruskal output map

# 5.9 Paths with Minimum Distance to visit all Locations(From a start point)

# After loading the map if 8 is given as input, it will ask for a starting location. Then Minimum Distance paths will be printed to visit all locations and represented in the graph.

# Sample Input: 9 SYLHET

# Sample Output:

# Figure 47: Prims output map

# 5.10 Remove Path

# The user can remove paths by selecting this option. The user can continue to remove path by giving YES as input when the program asks for it.

# Sample Input: 10 DHAKA SYLHET YES DHAKA CHITTAGONG NO

# Output:

# Figure 48: Remove path

# 5.11. Exit

# The program will be terminated.

# Challenges Faced

# While working on this project, I had to face numerous challenges.

* **Using Geographic coordinate system to create the graph:** I had to use the latitude and longitude value to create the graph. For this I had to transform the coordinates into a useable format. This was a difficult task as I also had to change the Cartesian coordinates into graphics screen suitable coordinates.
* **Using Visualization:** Visualizing the graph was a challenging task. I had to learn a lot about graphics.h to represent the graph perfectly. Also showing the paths from start to end was difficult.
* **Using Different Algorithms in a single project:** In this project, I had to use a lot of algorithms on a specific graph. Handling them properly was a challenging task to perform.

# Conclusion

This Travel Planner Project enables the user to enjoy a stress-free travel planning procedure. By providing optimized solutions, valuable resources like time and money can be saved.

The Travel Planner Project stands as a pivotal solution in addressing the challenges associated with travel planning. With the help of different graph algorithm, these challenges were solved and the best solution was provided.

Visualization added a different dimension in this project. With its help, ssers can easily grasp and understand optimal solutions, routes, and connections on a visual map.

In conclusion, this project played a crucial role by seamlessly integrating advanced algorithms with intuitive visualizations providing an enjoyable travel planning process to the users.

**Extensions:**

* This project can be extended by adding more locations like divisions and cities. This will increase the appropriateness and reliability of the project.
* Adding different transportation system(via Bus,Train,Airplane etc.) along with their cost and time will increase the domain of this project.
* Including Real-time data will extend this project at a great level.

**Reference**

1. <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/> , Geeksforgeeks, 02/09/2023
2. <https://developerinsider.co/graphics-graphics-h-c-programming/> , developerinsider, 05/09/2023
3. <https://www.javatpoint.com/bfs-program-in-c> , javatpoint, 13/10/2023
4. <https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/> , Geeksforgeeks, 24/10/2023
5. <https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/> , Geeksforgeeks, 12/11/23