DA Project 2

Fernando Oliveira
Ilaha Rahman
Mariana Conde

Classes

ToyGraph

- Functions that have the data type of the csv data files, i.e., the origin node, the destination node, and the distance

FileHandler

- Collection of Functions that allows us to read the csv files and store the data properly in our system

Interface

- Main Class of the Program since its what the user sees when they run it, allowing for the user to interact with it

Node

- Stores the important data relate to a Node, like its Id and possible outgoing edges

Edge

- Stores the Edges found in the files, i.e., the connection between the nodes.

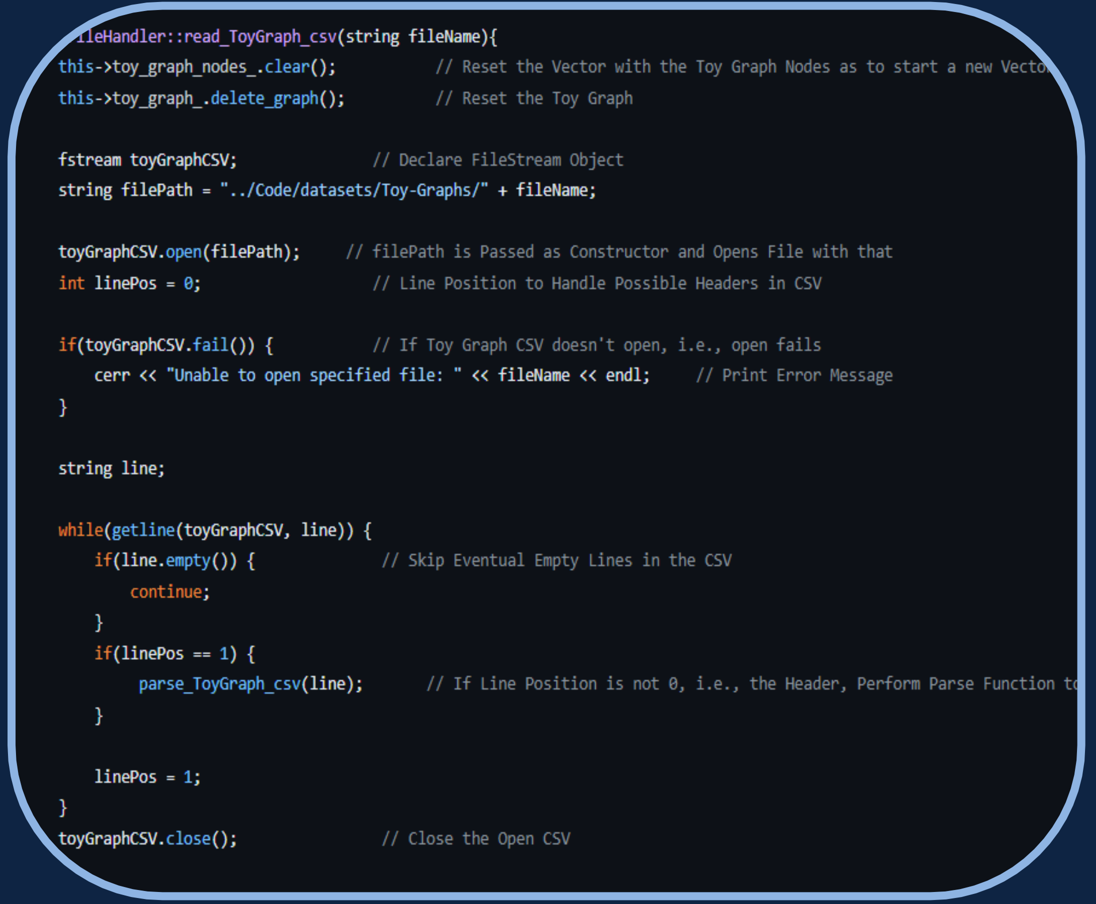
Graph

- A class that puts togethter class Node and class Edge, and forms a graph, along with some functions for proper searching

Utils

- Class that contains all the functions that needed to be used outside the ones from the other classes

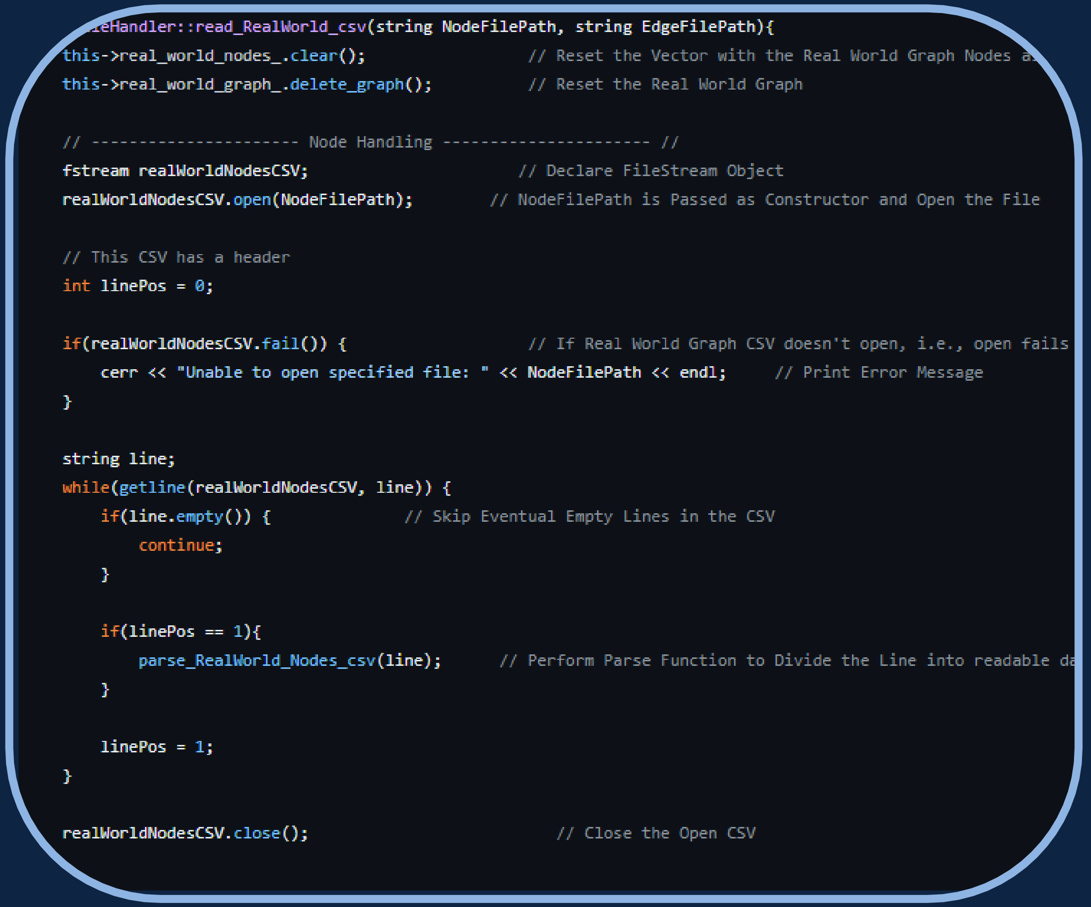
Reading and Importing Datasets



```
ileHandler::read_ToyGraph_csv(string fileName){
this->toy_graph_nodes_.clear();          // Reset the Vector with the Toy Graph Nodes as to start a new Vecto
this->toy_graph_.delete_graph();         // Reset the Toy Graph

fstream toyGraphCSV;                   // Declare FileStream Object
string filePath = "../Code/datasets/Toy-Graphs/" + fileName;

toyGraphCSV.open(filePath);     // filePath is Passed as Constructor and Opens File with that
int linePos = 0;                       // Line Position to Handle Possible Headers in CSV

if(toyGraphCSV.fail()) {               // If Toy Graph CSV doesn't open, i.e., open fails
    cerr << "Unable to open specified file: " << fileName << endl;     // Print Error Message
}

string line;

while(getline(toyGraphCSV, line)) {
    if(line.empty()) {                  // Skip Eventual Empty Lines in the CSV
        continue;
    }
    if(linePos == 1) {
         parse_ToyGraph_csv(line);       // If Line Position is not 0, i.e., the Header, Perform Parse Function to
    }

    linePos = 1;
}
toyGraphCSV.close();                    // Close the Open CSV
```

readToyGraph

- Obtains the data from each of the datasets in ToyGraphs directory like:
  - Origin
  - Destination
  - Distance
  - Possible Node Names

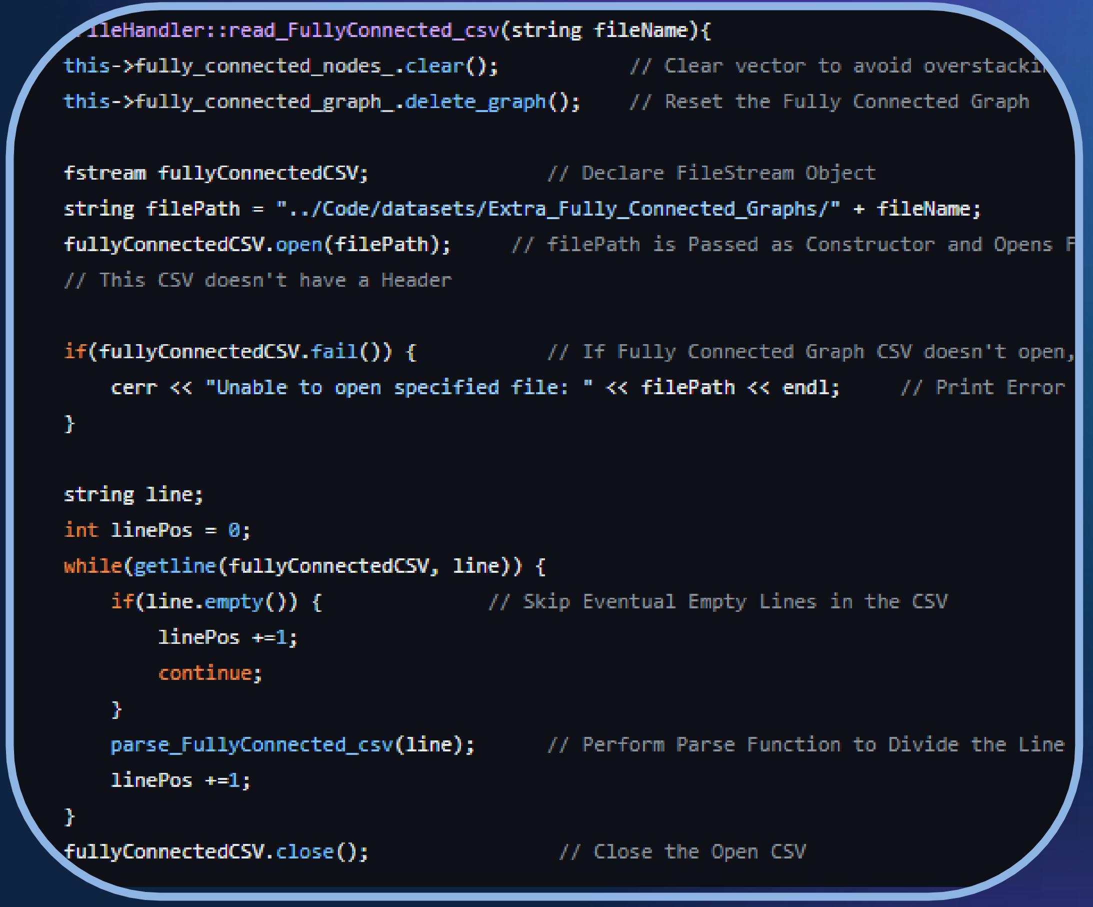


```
ileHandler::read_RealWorld_csv(string NodeFilePath, string EdgeFilePath){
this->real_world_nodes_.clear();                // Reset the Vector with the Real World Graph Nodes a
this->real_world_graph_.delete_graph();         // Reset the Real World Graph

// ---------------------- Node Handling ---------------------- //
fstream realWorldNodesCSV;                     // Declare FileStream Object
realWorldNodesCSV.open(NodeFilePath);        // NodeFilePath is Passed as Constructor and Open the File

// This CSV has a header
int linePos = 0;

if(realWorldNodesCSV.fail()) {                  // If Real World Graph CSV doesn't open, i.e., open fails
    cerr << "Unable to open specified file: " << NodeFilePath << endl;     // Print Error Message
}

string line;
while(getline(realWorldNodesCSV, line)) {
    if(line.empty()) {              // Skip Eventual Empty Lines in the CSV
        continue;
    }

    if(linePos == 1){
        parse_RealWorld_Nodes_csv(line);      // Perform Parse Function to Divide the Line into readable da
    }

    linePos = 1;
}

realWorldNodesCSV.close();                         // Close the Open CSV
```

readRealWorldGraph

- Obtains the data from each of the datasets in Real World Graphs directory like:
  - Nodes
  - Edges

  And joins them to make a Graph



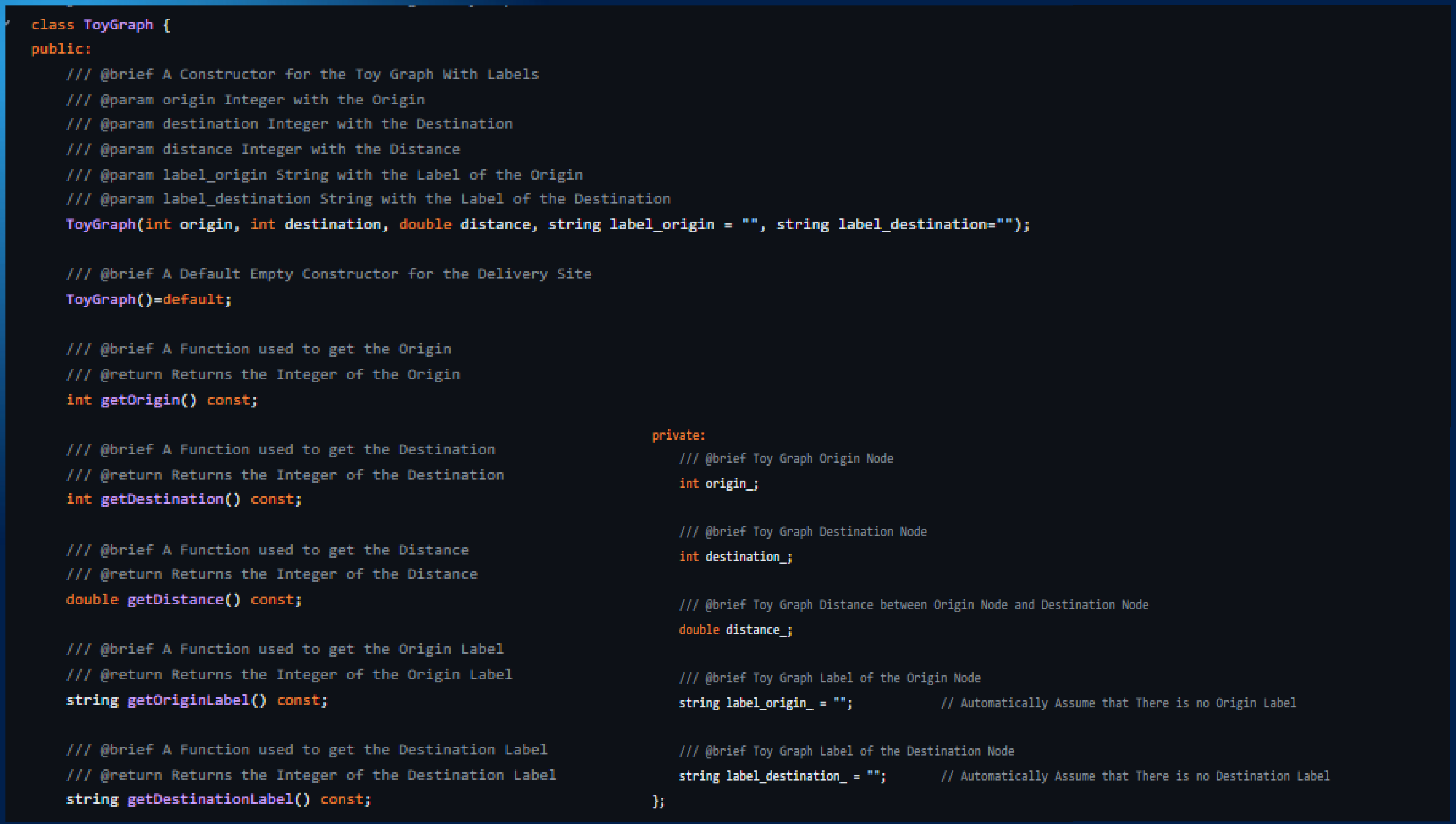
```
ileHandler::read_FullyConnected_csv(string fileName){
this->fully_connected_nodes_.clear();          // Clear vector to avoid overstacki
this->fully_connected_graph_.delete_graph();    // Reset the Fully Connected Graph

fstream fullyConnectedCSV;                 // Declare FileStream Object
string filePath = "../Code/datasets/Extra_Fully_Connected_Graphs/" + fileName;
fullyConnectedCSV.open(filePath);     // filePath is Passed as Constructor and Opens F
// This CSV doesn't have a Header

if(fullyConnectedCSV.fail()) {             // If Fully Connected Graph CSV doesn't open,
    cerr << "Unable to open specified file: " << filePath << endl;     // Print Error
}

string line;
int linePos = 0;
while(getline(fullyConnectedCSV, line)) {
    if(line.empty()) {              // Skip Eventual Empty Lines in the CSV
        linePos +=1;
        continue;
    }
    parse_FullyConnected_csv(line);      // Perform Parse Function to Divide the Line
    linePos +=1;
}
fullyConnectedCSV.close();               // Close the Open CSV
```

readFullyConnectGraph

- Obtains the data from each of the datasets in Extra Connected directory like:
  - Edge Origin
  - Edge Destination
  - Edge Distance

- Using the format of the files of the dataset (csv) for our benefit, since they are "comma-separated values", i.e., looking into each line of said file gives us info about another object in file, this is further explored using the parse functions that show up in the functions above

Toy Graph



```
class ToyGraph {
public:
    /// @brief A Constructor for the Toy Graph With Labels
    /// @param origin Integer with the Origin
    /// @param destination Integer with the Destination
    /// @param distance Integer with the Distance
    /// @param label_origin String with the Label of the Origin
    /// @param label_destination String with the Label of the Destination
    ToyGraph(int origin, int destination, double distance, string label_origin = "", string label_destination="");

    /// @brief A Default Empty Constructor for the Delivery Site
    ToyGraph()=default;

    /// @brief A Function used to get the Origin
    /// @return Returns the Integer of the Origin
    int getOrigin() const;

    /// @brief A Function used to get the Destination
    /// @return Returns the Integer of the Destination
    int getDestination() const;

    /// @brief A Function used to get the Distance
    /// @return Returns the Integer of the Distance
    double getDistance() const;

    /// @brief A Function used to get the Origin Label
    /// @return Returns the Integer of the Origin Label
    string getOriginLabel() const;

    /// @brief A Function used to get the Destination Label
    /// @return Returns the Integer of the Destination Label
    string getDestinationLabel() const;
```

```
private:
    /// @brief Toy Graph Origin Node
    int origin_;

    /// @brief Toy Graph Destination Node
    int destination_;

    /// @brief Toy Graph Distance between Origin Node and Destination Node
    double distance_;

    /// @brief Toy Graph Label of the Origin Node
    string label_origin_ = "";            // Automatically Assume that There is no Origin Label

    /// @brief Toy Graph Label of the Destination Node
    string label_destination_ = "";       // Automatically Assume that There is no Destination Label
};
```

Structures

Source and Destination

Has the information of the graphs edges, admitting that each one has an origin node and a destination node

Distance

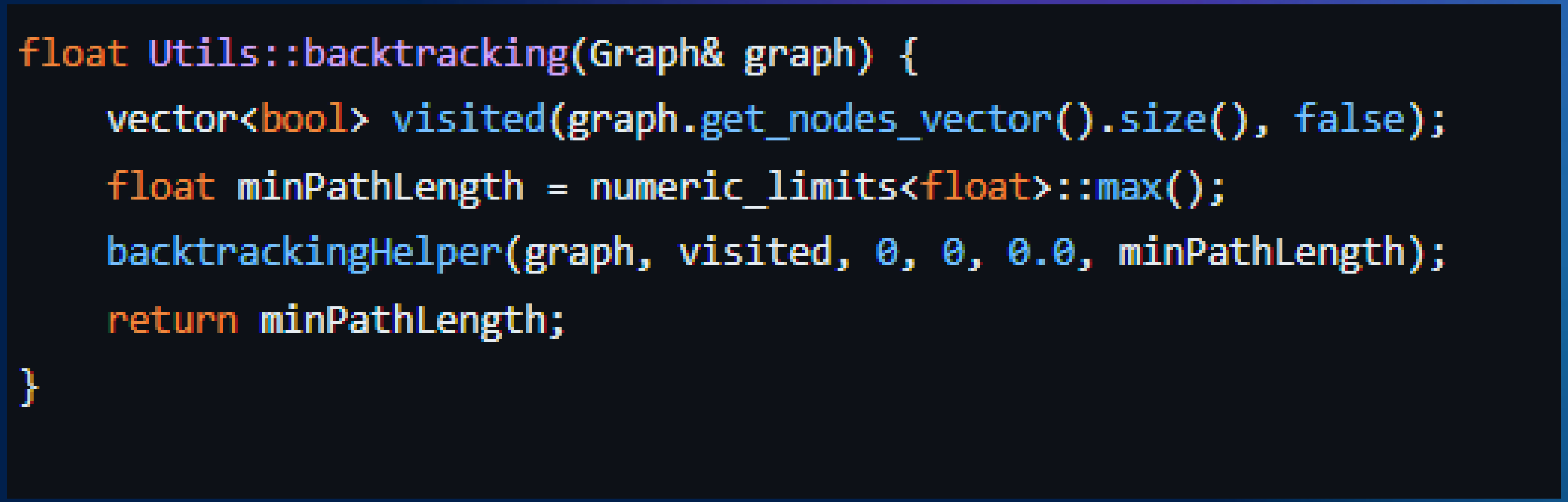
Has the Information related to the distance in between these two nodes

These Structures aid us in dealing with the data type of our datasets, and further getting more information about said data as to make our search of these objects easier. Another argument in the construction are the labels that in some graphs are present and other are not. By giving it a default value, means that this data does not always need to be inputted

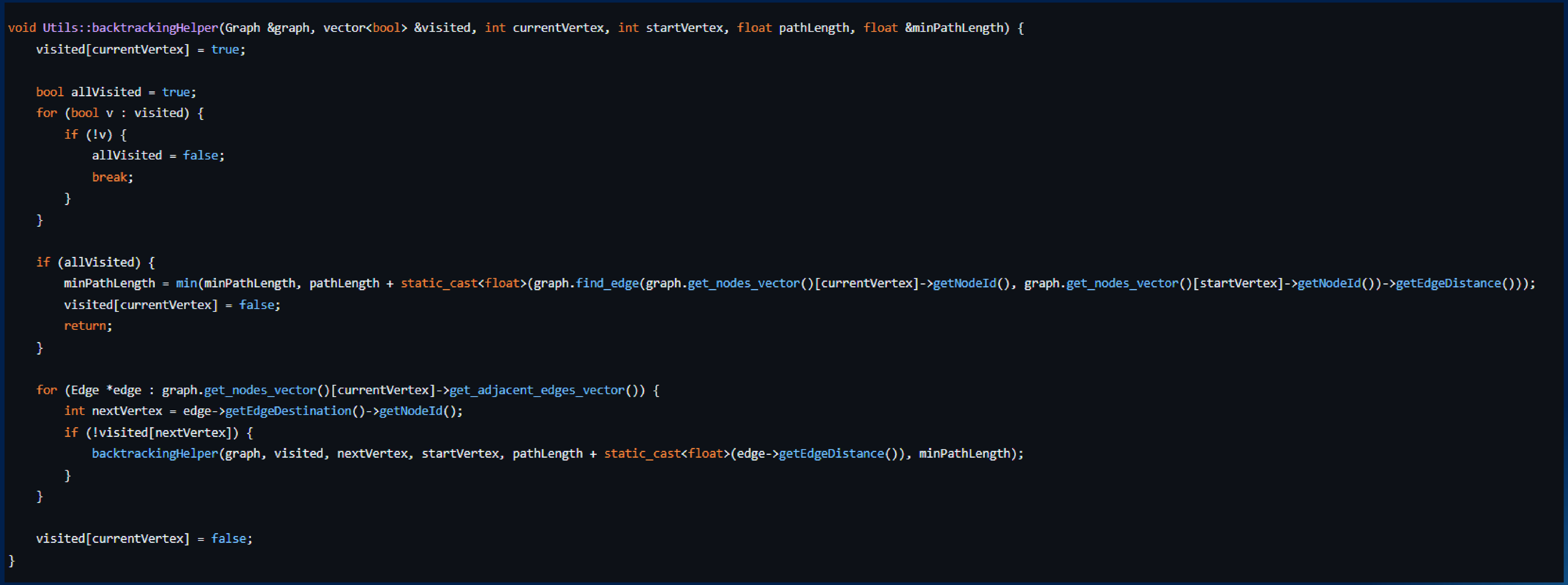
Implemented Functions

- Backtracking algorithm

This is one of the asked functions, that performs a Backtracking algorithm in our Graphs. Along with A function used as auxiliary to the main one



```cpp
float Utils::backtracking(Graph& graph) {
    vector<bool> visited(graph.get_nodes_vector().size(), false);
    float minPathLength = numeric_limits<float>::max();
    backtrackingHelper(graph, visited, 0, 0, 0.0, minPathLength);
    return minPathLength;
}
```

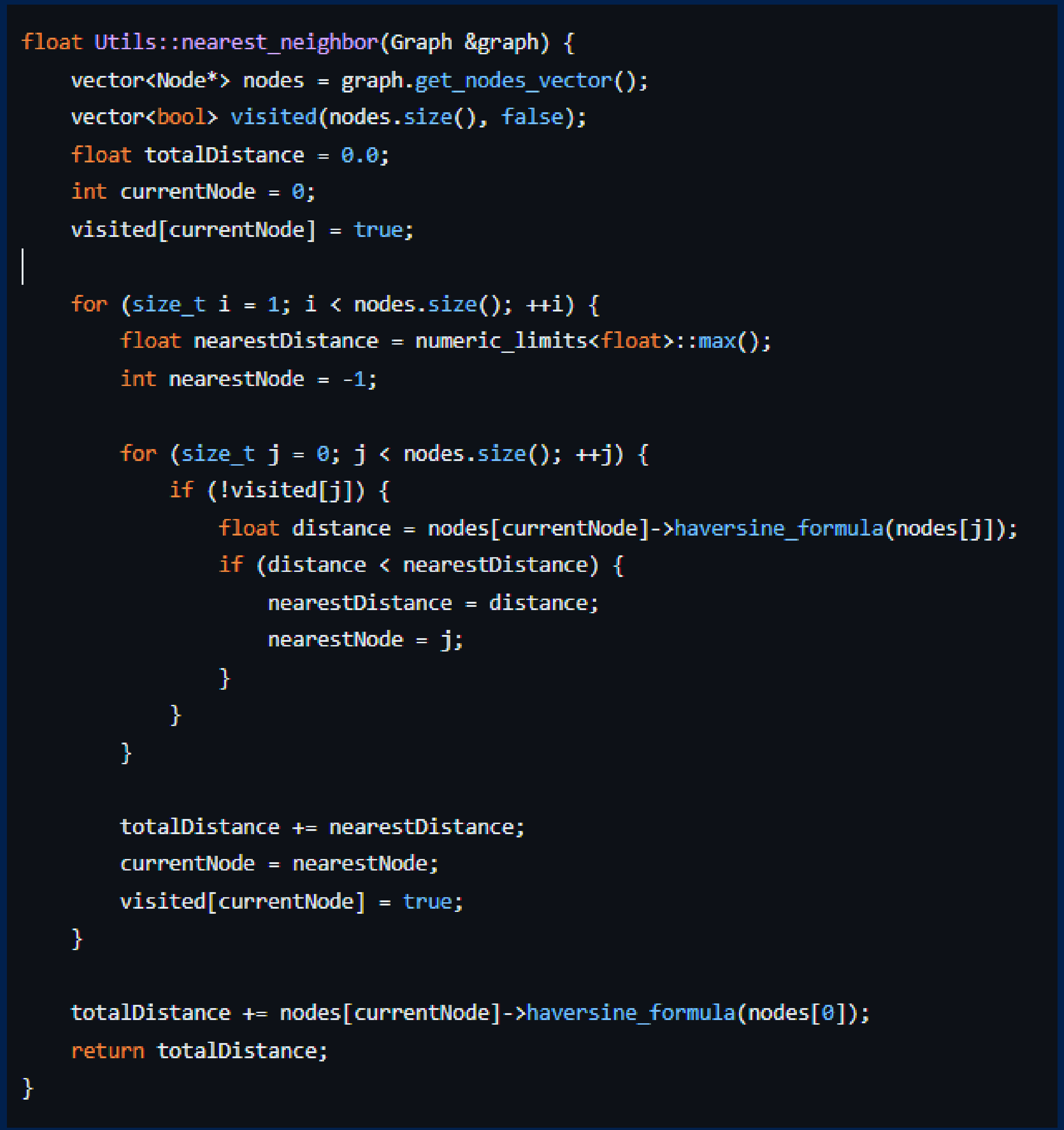


```cpp
void Utils::backtrackingHelper(Graph &graph, vector<bool> &visited, int currentVertex, int startVertex, float pathLength, float &minPathLength) {
    visited[currentVertex] = true;

    bool allVisited = true;
    for (bool v : visited) {
        if (!v) {
            allVisited = false;
            break;
        }
    }

    if (allVisited) {
        minPathLength = min(minPathLength, pathLength + static_cast<float>(graph.find_edge(graph.get_nodes_vector()[currentVertex]->getNodeId(), graph.get_nodes_vector()[startVertex]->getNodeId())->getEdgeDistance()));
        visited[currentVertex] = false;
        return;
    }

    for (Edge *edge : graph.get_nodes_vector()[currentVertex]->get_adjacent_edges_vector()) {
        int nextVertex = edge->getEdgeDestination()->getNodeId();
        if (!visited[nextVertex]) {
            backtrackingHelper(graph, visited, nextVertex, startVertex, pathLength + static_cast<float>(edge->getEdgeDistance()), minPathLength);
        }
    }

    visited[currentVertex] = false;
}
```

Implemented Functions

- Nearest Neighbour

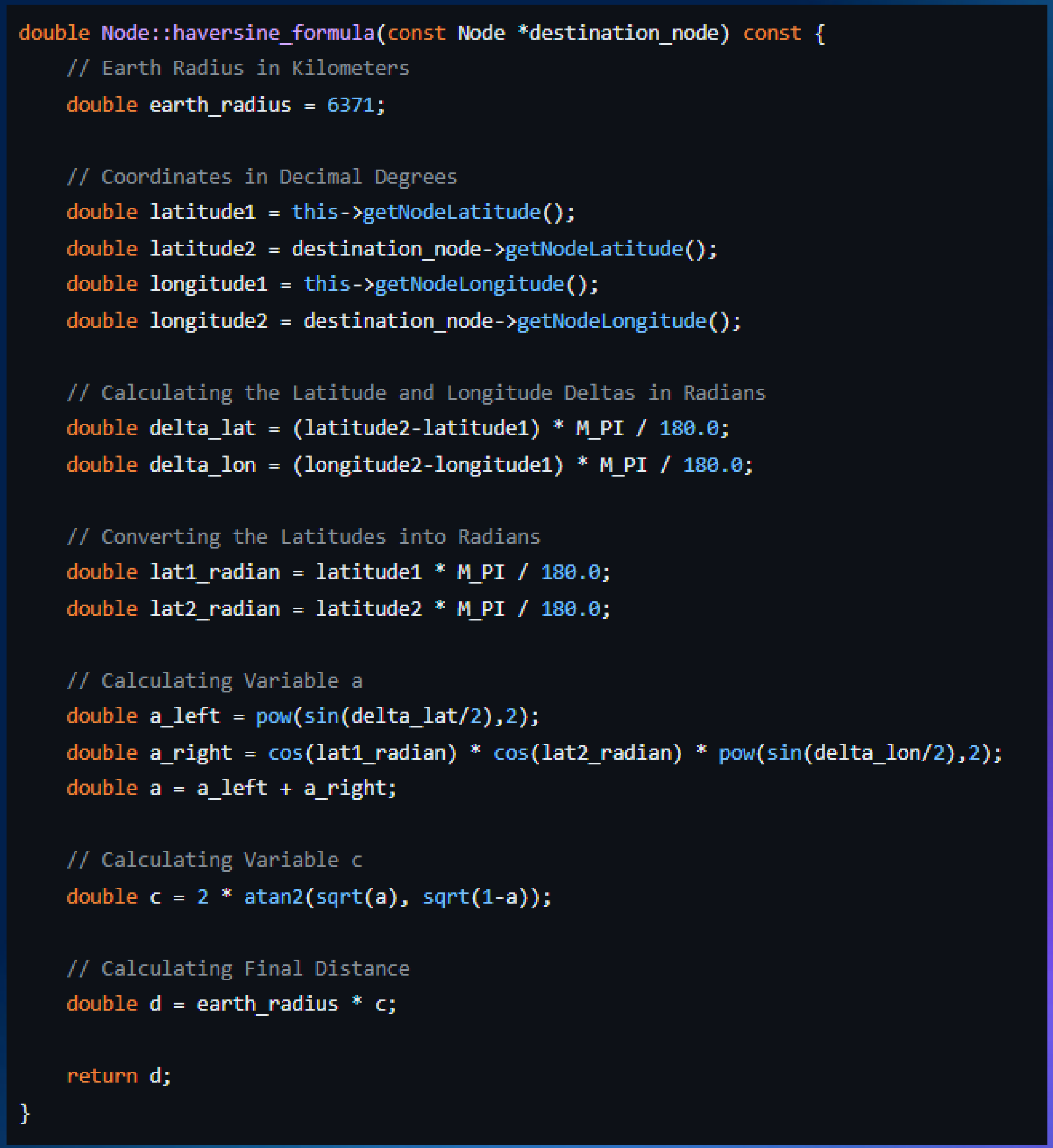
A function used to find the nearest neighbour
In a graph , and returns the distance between them



```
float Utils::nearest_neighbor(Graph &graph) {
    vector<Node*> nodes = graph.get_nodes_vector();
    vector<bool> visited(nodes.size(), false);
    float totalDistance = 0.0;
    int currentNode = 0;
    visited[currentNode] = true;

    for (size_t i = 1; i < nodes.size(); ++i) {
        float nearestDistance = numeric_limits<float>::max();
        int nearestNode = -1;

        for (size_t j = 0; j < nodes.size(); ++j) {
            if (!visited[j]) {
                float distance = nodes[currentNode]->haversine_formula(nodes[j]);
                if (distance < nearestDistance) {
                    nearestDistance = distance;
                    nearestNode = j;
                }
            }
        }

        totalDistance += nearestDistance;
        currentNode = nearestNode;
        visited[currentNode] = true;
    }

    totalDistance += nodes[currentNode]->haversine_formula(nodes[0]);
    return totalDistance;
}
```

- Haversine Formula

A function used to calculate the distance between points, using their latitude and longitude

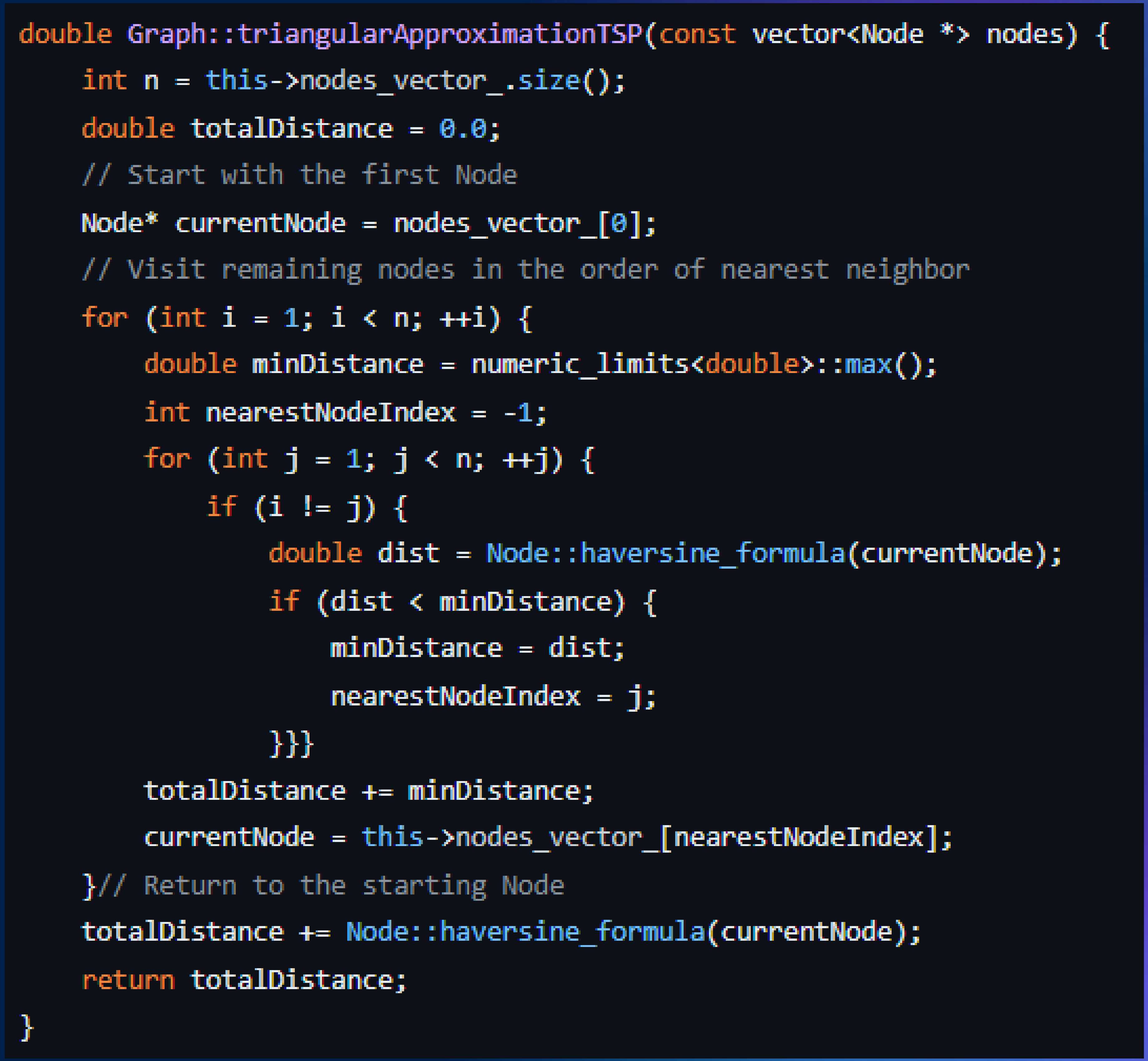


```
double Node::haversine_formula(const Node *destination_node) const {
    // Earth Radius in Kilometers
    double earth_radius = 6371;

    // Coordinates in Decimal Degrees
    double latitude1 = this->getNodeLatitude();
    double latitude2 = destination_node->getNodeLatitude();
    double longitude1 = this->getNodeLongitude();
    double longitude2 = destination_node->getNodeLongitude();

    // Calculating the Latitude and Longitude Deltas in Radians
    double delta_lat = (latitude2-latitude1) * M_PI / 180.0;
    double delta_lon = (longitude2-longitude1) * M_PI / 180.0;

    // Converting the Latitudes into Radians
    double lat1_radian = latitude1 * M_PI / 180.0;
    double lat2_radian = latitude2 * M_PI / 180.0;

    // Calculating Variable a
    double a_left = pow(sin(delta_lat/2),2);
    double a_right = cos(lat1_radian) * cos(lat2_radian) * pow(sin(delta_lon/2),2);
    double a = a_left + a_right;

    // Calculating Variable c
    double c = 2 * atan2(sqrt(a), sqrt(1-a));

    // Calculating Final Distance
    double d = earth_radius * c;

    return d;
}
```

Implemented Functions

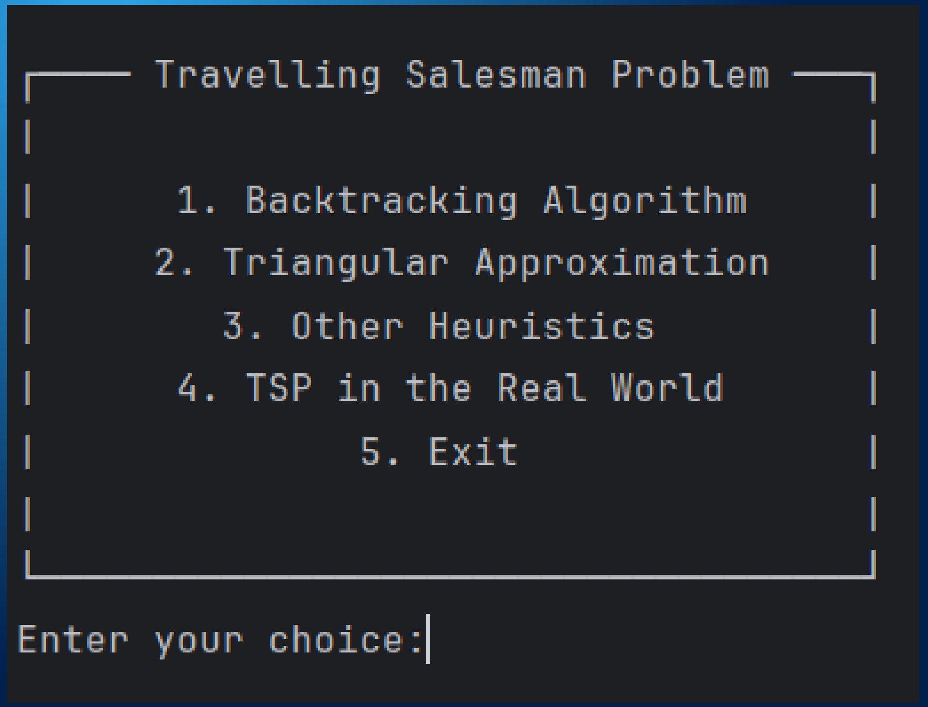
- Triangular Approximation

This is one of the asked functions, that performs a
Triangle Aproximation in our Graphs.



```cpp
double Graph::triangularApproximationTSP(const vector<Node *> nodes) {
    int n = this->nodes_vector_.size();
    double totalDistance = 0.0;
    // Start with the first Node
    Node* currentNode = nodes_vector_[0];
    // Visit remaining nodes in the order of nearest neighbor
    for (int i = 1; i < n; ++i) {
        double minDistance = numeric_limits<double>::max();
        int nearestNodeIndex = -1;
        for (int j = 1; j < n; ++j) {
            if (i != j) {
                double dist = Node::haversine_formula(currentNode);
                if (dist < minDistance) {
                    minDistance = dist;
                    nearestNodeIndex = j;
                }}}
        totalDistance += minDistance;
        currentNode = this->nodes_vector_[nearestNodeIndex];
    }// Return to the starting Node
    totalDistance += Node::haversine_formula(currentNode);
    return totalDistance;
}
```

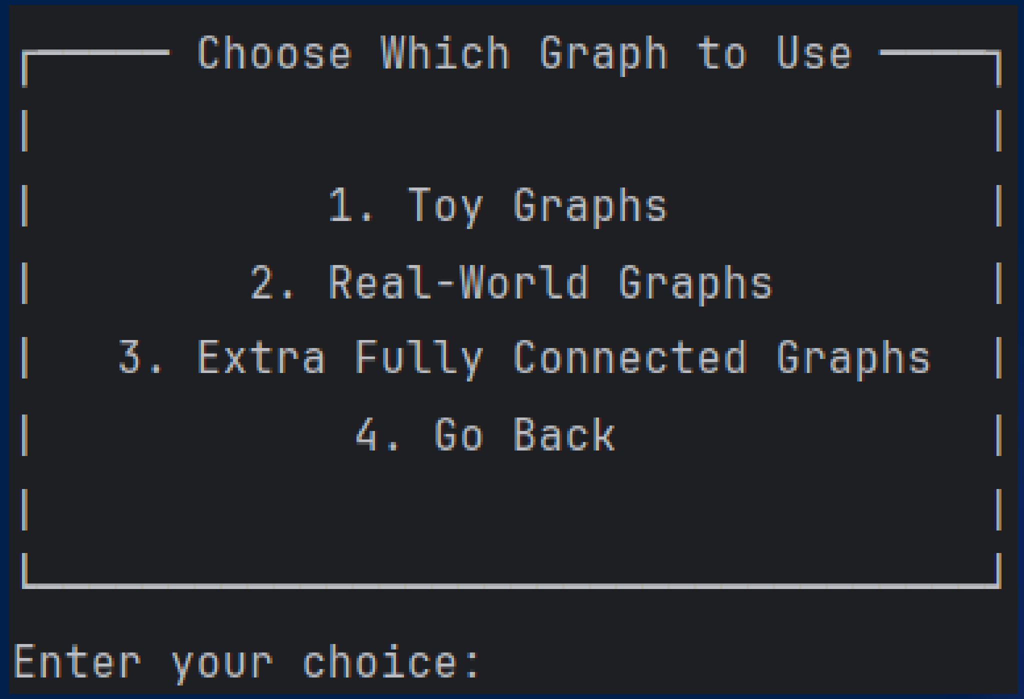
User Interface



```
┌──── Travelling Salesman Problem ────┐
|                                     |
|      1. Backtracking Algorithm      |
|     2. Triangular Approximation     |
|        3. Other Heuristics          |
|      4. TSP in the Real World       |
|              5. Exit                |
|                                     |
└─────────────────────────────────────┘
Enter your choice:
```

- ## Main Menu

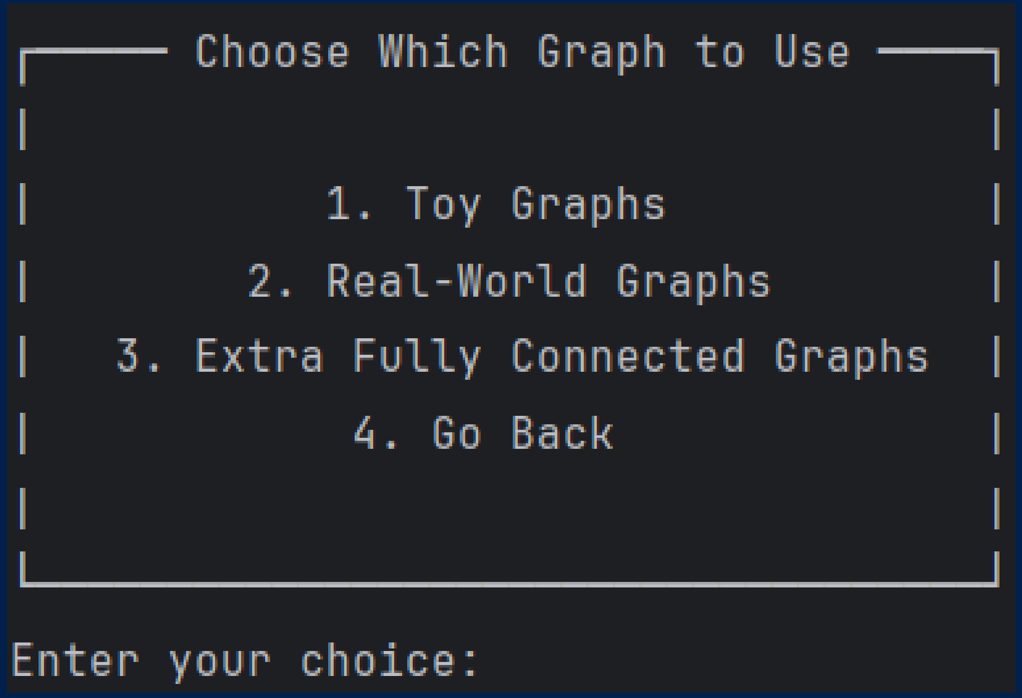
Allow the user to choose between which algorithm he wants to go through



```
┌──── Choose Which Graph to Use ────┐
|                                   |
|           1. Toy Graphs           |
|       2. Real-World Graphs        |
|  3. Extra Fully Connected Graphs  |
|            4. Go Back             |
|                                   |
└───────────────────────────────────┘
Enter your choice:
```

- ## Graph Menu

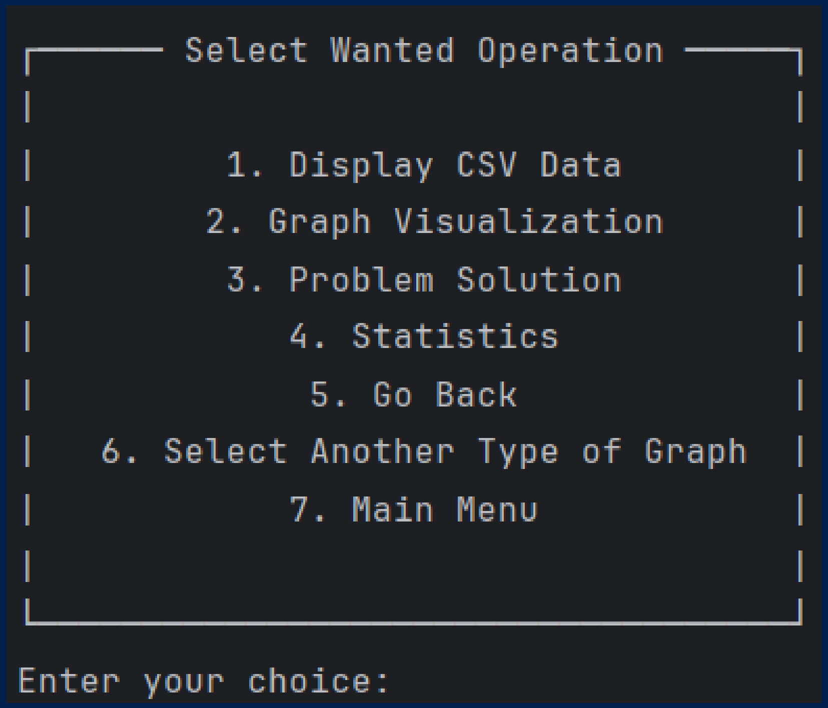
Allow the user to choose which type of graphs he would prefer to work with based on our datasets



```
┌──── Choose Which Graph to Use ────┐
|                                   |
|           1. Toy Graphs           |
|       2. Real-World Graphs        |
|  3. Extra Fully Connected Graphs  |
|            4. Go Back             |
|                                   |
└───────────────────────────────────┘
Enter your choice:
```

- ## In Depth Graph Menu

Allow the user to choose which graphs he would prefer to work with based on their choice of graph type, i.e, allowing the user to choose the csv



```
┌──── Select Wanted Operation ────┐
|                                 |
|       1. Display CSV Data       |
|     2. Graph Visualization      |
|       3. Problem Solution       |
|          4. Statistics          |
|           5. Go Back            |
| 6. Select Another Type of Graph |
|          7. Main Menu           |
|                                 |
└─────────────────────────────────┘
Enter your choice:
```

- ## Operation Menu

Allow the user to choose what to do with the data, that being:

- Display The CSV
- Show the Graph Created
- Solve the Algorithm Previously Chosen
- Obtain Statistics of the Algoritmh

Biggest Difficulties

- The Part of Working with the DataSets was probably the simplest part, along with the creation of the menu.
- The Implementation of the Graph caused some issues initially but were quickly fixed and started working as intended
- The Functions Implementations was a bit harder, as to trying to use our data and perform the algorithms asked of us
- The Statistics Part we did not manage to do.
- The Hardest was probably implementing our algorithmic functions in with our interface, stuff that we couldn’t manage to finish and make it work.