Routing Algorithm for Ocean Shipping and Urban Deliveries

1.0

Generated by Doxygen 1.9.7

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 CPheadquarters Class Reference

Public Member Functions

- void read_edges (std::string path)
- void read_coordinates (std::string path)
- Graph getGraph () const
- void heuristic (long path[ ], unsigned int &nodesVisited, double &totalDistance, long id)

  Initial part of the algorithm, finds the route of the vertex with the given id, calling the recursive function.
- void heuristicRec (Vertex ∗v, long path[ ], unsigned int currentIndex, double distance, unsigned int &nodes↩ Visited, double &totalDistance)

  Recursive part of the heuristic that looks for the closest vertex to the actual one, the closest vertex is determine using the angle, so this heuristic uses geological information from the vertex.
- void chooseRoute ()

  Iterates through all vertex to determine with which one to starts.
- void backtrack ()

  Use a backtracking exhaustive approach for TSP Applicable only for very small graphs.
- void hamiltonianCycle ()
- void triangular_Approximation_Heuristic ()

  Calculates the total cost of the TSP problem using a 2-approximation strategy.
- void pathRec (Vertex ∗vertex)

  Generates a pre-order path of the MST using a DFS strategy, storing their indexes in a vector.
- double degToRad (double degrees)

  Translates degrees to radians.
- double haversineDistance (double latitude1, double longitude1, double latitude2, double longitude2)

  Returns the distance between two points using their coordinates in kilometers.
- double getDist (int a, int b)

  Returns the distance between the nodes with indexes id=a and id=b.

3.1.1 Detailed Description

Definition at line 13 of file CPheadquarters.h.

3.1.2 Member Function Documentation

3.1.2.1 backtrack()

```
void CPheadquarters::backtrack ( )
```

Use a backtracking exhaustive approach for TSP Applicable only for very small graphs.

Definition at line 205 of file CPheadquarters.cpp.

```
00205                                 {
00206     std::vector<Vertex*> shortestPath;
00207     double shortestPathCost = 0;
00208
00209
00210     if (this->graph.TSP(shortestPath, shortestPathCost)) {
00211         cout « "Shortest Hamiltonian cycle: ";
00212         for (auto vertex : shortestPath)
00213             cout « vertex->getId() « " ";
00214         cout « "\nCost: " « shortestPathCost « endl;
00215     }
00216     else {
00217         cout « "The graph does not have a Hamiltonian cycle" « endl;
00218     }
00219 }
```

3.1.2.2 chooseRoute()

```
void CPheadquarters::chooseRoute ( )
```

Iterates through all vertex to determine with which one to starts.

Note

this algorithm only work when it starts in some specifics vertex so that's the reason it needs to go through all possible nodes to choose the one who satisfies the needs of the problem

Attention

since this parte of the heuristic iterates through all vertex and call the recursive part from the algorithm that is O(E) the time complexity is O(EV)

Definition at line 169 of file CPheadquarters.cpp.

```
00169                                  {
00170     long id;
00171     auto pathSize = graph.getNumVertex();
00172     auto vertixes = graph.getVertexSet();
00173     long path[pathSize];
00174     unsigned int nodesVisited = 0;
00175     double distance = 0;
00176     for(auto it =vertixes.begin(); it != vertixes.end(); it++){
00177         id = it->first;
00178         heuristic(path, nodesVisited, distance, id);
00179         if(nodesVisited == pathSize){
00180             long sourceId = path[pathSize-1];
00181             long destId = path[0];
00182             Vertex *sourceV = graph.findVertex(sourceId);
00183             Edge *missingEdge = sourceV->getEdge(destId);
00184             if(missingEdge!= nullptr){
00185                 distance += missingEdge->getDistance();
00186                 for(int i = 0; i < pathSize; i++){
00187                     cout « path[i] « "->";
00188                 }
00189                 cout « destId;
00190                 cout « "\nTotal distance: " « distance « endl;
00191                 break;
00192             }
00193         }
00194     }
00195 }
```

3.1.2.3 degToRad()

```
double CPheadquarters::degToRad (
            double degrees )
```

Translates degrees to radians.

Parameters

degrees	

Returns

Attention

Time Complexity: O(1)

Definition at line 304 of file CPheadquarters.cpp.

```
00304                                         {
00305     return degrees*M_PI/180.0;
00306 }
```

3.1.2.4 getDist()

```
double CPheadquarters::getDist (
            int a,
            int b )
```

Returns the distance between the nodes with indexes id=a and id=b.

If their distance is not explicit in the edges, then it is calculated using the haversineDistance() function, if possible.

Parameters

a	
b	

Returns

Attention

Time Complexity: O(E)

Definition at line 295 of file CPheadquarters.cpp.

```
00295                                         {
00296     for (auto edge: graph.findVertex(a)->getAdj()){
00297         if (edge->getDest()->getId()==b) return edge->getDistance();
00298     }
00299     return haversineDistance(graph.findVertex(a)->getLatitude(),graph.findVertex(a)->getLongitude(),
      graph.findVertex(b)->getLatitude(), graph.findVertex(b)->getLongitude());
00300 }
```

3.1.2.5 getGraph()

```
Graph CPheadquarters::getGraph ( ) const
```

Definition at line 199 of file CPheadquarters.cpp.

```
00199                                         {
00200     return this->graph;
00201 }
```

3.1.2.6 hamiltonianCycle()

```
void CPheadquarters::hamiltonianCycle ( )
```

Definition at line 221 of file CPheadquarters.cpp.

```
00221                                        {
00222     std::vector<Vertex*> path;
00223     double cost = 0;
00224     if (this->graph.hasHamiltonianCycle(path, cost)) {
00225         cout « "Hamiltonian cycle: ";
00226         for (auto vertex : path)
00227             cout « vertex->getId() « " ";
00228         cout « "\nCost: " « cost « endl;
00229     }
00230     else {
00231         cout « "The graph does not have a Hamiltonian cycle" « endl;
00232     }
00233 }
```

3.1.2.7 haversineDistance()

```
double CPheadquarters::haversineDistance (
               double latitude1,
               double longitude1,
               double latitude2,
               double longitude2 )
```

Returns the distance between two points using their coordinates in kilometers.

Parameters

latitude1	
longitude1	
latitude2	
longitude2	

Returns

Attention

Time Complexity: O(1)

Definition at line 308 of file CPheadquarters.cpp.

```
00308
      {
00309     double ang_lat=degToRad(latitude2-latitude1);
00310     double ang_lon=degToRad(longitude2-longitude1);
00311     double a =sin(ang_lat / 2) * sin(ang_lat / 2) +
00312             cos(degToRad (latitude1)) * cos(degToRad (latitude2)) *
00313              sin(ang_lon / 2) * sin(ang_lon / 2);
00314
00315     double c = 2 * atan2(sqrt(a), sqrt(1 - a));
00316
00317     return EarthRadius * c;
00318 }
```

3.1.2.8 heuristic()

```
void CPheadquarters::heuristic (
            long path[ ],
            unsigned int & nodesVisited,
            double & totalDistance,
            long id )
```

Initial part of the algorithm, finds the route of the vertex with the given id, calling the recursive function.

Parameters

route	
nodesVisited	
totalDistance	
id	

Definition at line 150 of file CPheadquarters.cpp.

```
00150
     {
00151
00152     for (const auto vertex: graph.getVertexSet()) {
00153         vertex.second->setVisited(false);
00154
00155     }
00156
00157
00158     Vertex *actual = graph.findVertex(id);
00159
00160     double distance = 0;
00161     route[0] = actual->getId();
00162
00163     actual->setVisited(true);
00164
00165     heuristicRec(actual, route, 1, distance, nodesVisited, totalDistance);
00166 }
```

3.1.2.9 heuristicRec()

```
void CPheadquarters::heuristicRec (
            Vertex * v,
            long path[ ],
            unsigned int currentIndex,
            double distance,
            unsigned int & nodesVisited,
            double & totalDistance )
```

Recursive part of the heuristic that looks for the closest vertex to the actual one, the closest vertex is determine using the angle, so this heuristic uses geological information from the vertex.

Parameters

v	
route	
currentIndex	
distance	
nodesVisited	
totalDistance	

Attention

the time complexity of this part of the heuristic is O(E)

Definition at line 91 of file CPheadquarters.cpp.

```
00091
          {
00092
00093     bool nodesStillUnvisited = false;
00094
00095     double long1 = v->getLongitude();
00096     double lat1 = v->getLatitude();
00097
00098     Vertex *small;
00099     double smallAngle = 10000;
00100     double x;
00101     double y;
00102     double angle;
00103     double dist;
00104
00105     for (const auto &edge: v->getAdj()) {
00106         Vertex *v2 = edge->getDest();
00107         double dist2 = edge->getDistance();
00108         if(v2->isVisited() == false){
00109             nodesStillUnvisited = true;
00110
00111             double long2 = edge->getDest()->getLongitude();
00112             double lat2 = edge->getDest()->getLatitude();
00113
00114             x = long1 - long2;
00115             y = lat1 - lat2;
00116
00117             angle = atan2(y,x);
00118
00119             if(angle < smallAngle){
00120                 smallAngle = angle;
00121                 small = v2;
00122                 dist = dist2;
00123             }
00124
00125         }
00126     }
00127
00128     bool inRoute = false;
00129     for(int i = 0; i < currentIndex; i++){
00130         if(route[i] == v->getId()){
00131             inRoute = true;
00132         }
00133     }
00134
00135     if(nodesStillUnvisited){
00136         route[currentIndex] = small->getId();
00137         small->setVisited(true);
00138
00139         heuristicRec(small, route, currentIndex+1, distance + dist, nodesVisited, totalDistance);
00140     }
00141     else{
00142         nodesVisited = currentIndex;
00143         totalDistance = distance;
00144
00145     }
00146
00147
00148 }
```

3.1.2.10 pathRec()

```
void CPheadquarters::pathRec (
        Vertex * vertex )
```

Generates a pre-order path of the MST using a DFS strategy, storing their indexes in a vector.

Parameters

vertex	

Attention

Time Complexity: O(E)

Definition at line 234 of file CPheadquarters.cpp.

```
00234                                           {
00235     mst_preorder_path.push_back(vertex->getId());
00236     for (auto child : vertex->getChildren()) {
00237         pathRec(graph.getVertexSet()[child]);
00238     }
00239     return;
00240 }
```

3.1.2.11 read_coordinates()

```
void CPheadquarters::read_coordinates (
            std::string path )
```

Definition at line 54 of file CPheadquarters.cpp.

```
00054                                                {
00055     std::ifstream inputFile2(path);
00056     string line2;
00057     std::getline(inputFile2, line2); // ignore first line
00058
00059
00060     while (getline(inputFile2, line2, '\n')) {
00061
00062         if (!line2.empty() && line2.back() == '\n') { // Check if the last character is '\r'
00063             line2.pop_back(); // Remove the '\r' character
00064         }
00065
00066         string id_;
00067         string temp1;
00068         string temp2;
00069         double longitude_;
00070         double latitude_;
00071
00072         stringstream inputString(line2);
00073
00074         getline(inputString, id_, ',');
00075         getline(inputString, temp1, ',');
00076         getline(inputString, temp2, ',');
00077
00078         long long_id = std::stol(id_);
00079         longitude_ = stod(temp1);
00080         latitude_ = stod(temp2);
00081
00082         auto v = graph.findVertex(long_id);
00083         v->setLongitude(longitude_);
00084         v->setLatitude(latitude_);
00085
00086         cout « long_id « '\n';
00087     }
00088 }
```

3.1.2.12 read_edges()

```
void CPheadquarters::read_edges (
            std::string path )
```

Definition at line 16 of file CPheadquarters.cpp.

```
00016                                           {
00017     std::ifstream inputFile1(path);
00018     string line1;
00019     std::getline(inputFile1, line1); // ignore first line
00020     while (getline(inputFile1, line1, '\n')) {
00021
00022         if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
00023             line1.pop_back(); // Remove the '\r' character
00024         }
00025
00026         string origin;
00027         string destination;
```

```
        string temp;
        double distance;


        stringstream inputString(line1);

        getline(inputString, origin, ',');
        getline(inputString, destination, ',');
        getline(inputString, temp, ',');


        distance = stod(temp);

        long origin_id = std::stol(origin);
        graph.addVertex(origin_id);

        long destination_id = std::stol(destination);
        graph.addVertex(destination_id);

        graph.addEdge(origin_id, destination_id, distance);
        graph.addEdge(destination_id, origin_id, distance);
        cout « origin « '\n';
    }
}
```

3.1.2.13 triangular_Approximation_Heuristic()

`void CPheadquarters::triangular_Approximation_Heuristic ( )`

Calculates the total cost of the TSP problem using a 2-approximation strategy.

Firstly running a variation of prim's algorithn with complexity O((V+E)logV). Then running a DFS algorithm (pathRec()) with complexity O(E) And finally adding the distances between all the nodes, worst case complexity O(E∗E)

Attention

Time Complexity: O((V+E)logV + 2E)

Definition at line 243 of file CPheadquarters.cpp.

```
                                                           {
    std::unordered_map<long,Vertex *> vertexis = graph.getVertexSet();
    for (auto v: vertexis) {
        v.second->setVisited(false);
        v.second->setDist(std::numeric_limits<double>::max());
        v.second->eraseChildren();
    }

    Vertex *root = graph.findVertex(0);
    root->setDist(0);
    MutablePriorityQueue<Vertex> q;
    q.insert(root);
    while (!q.empty()) {
        auto v = q.extractMin();
        cout«"working on:"«v->getId()«'\n';
        v->setVisited(true);
        if(v->getId()!=0) {
            v->getPath()->getOrig()->addChildren(v->getId());
        }
        for (auto &e: v->getAdj()) {
            Vertex *w = e->getDest();
            if (!w->isVisited()) {
                auto oldDist = w->getDist();
                if (e->getDistance() < oldDist) {
                    w->setDist(e->getDistance());
                    w->setPath(e);
                    if (oldDist == std::numeric_limits<double>::max()) {
                        q.insert(w);
                    } else {
                        q.decreaseKey(w);
                    }
                }
            }
        }
    }

```

```
    mst_preorder_path.clear();
    pathRec(root);

    double result=0;

    for (int i = 0; i < mst_preorder_path.size()-1; i++) {
        result+= getDist(mst_preorder_path[i],mst_preorder_path[i+1]);
    }
    result+=getDist(mst_preorder_path[mst_preorder_path.size()-1],mst_preorder_path[0]);

    cout«"Result:"«result;

}
```

The documentation for this class was generated from the following files:

- CPheadquarters.h
- CPheadquarters.cpp

3.2 Edge Class Reference

Public Member Functions

- Edge (Vertex ∗orig, Vertex ∗dest, double d)
- Vertex ∗ getDest () const
- double getDistance () const
- Vertex ∗ getOrig () const

Protected Attributes

- Vertex ∗ dest
- double distance
- Vertex ∗ orig

3.2.1 Detailed Description

Definition at line 93 of file VertexEdge.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Edge()

```
Edge::Edge (
            Vertex * orig,
            Vertex * dest,
            double d )
```

Definition at line 141 of file VertexEdge.cpp.

```
00141 : orig(orig), dest(dest), distance(d) {}
```

3.2.3 Member Function Documentation

3.2.3.1 getDest()

```
Vertex * Edge::getDest ( ) const
```

Definition at line 143 of file VertexEdge.cpp.

```
00143                             {
00144     return this->dest;
00145 }
```

3.2.3.2 getDistance()

```
double Edge::getDistance ( ) const
```

Definition at line 147 of file VertexEdge.cpp.

```
00147                                 {
00148     return this->distance;
00149 }
```

3.2.3.3 getOrig()

```
Vertex * Edge::getOrig ( ) const
```

Definition at line 151 of file VertexEdge.cpp.

```
00151                             {
00152     return this->orig;
00153 }
```

3.2.4 Member Data Documentation

3.2.4.1 dest

```
Vertex* Edge::dest [protected]
```

Definition at line 104 of file VertexEdge.h.

3.2.4.2 distance

```
double Edge::distance [protected]
```

Definition at line 105 of file VertexEdge.h.

3.2.4.3 orig

```
Vertex* Edge::orig [protected]
```

Definition at line 108 of file VertexEdge.h.

The documentation for this class was generated from the following files:

- VertexEdge.h
- VertexEdge.cpp

3.3 Graph Class Reference

Public Member Functions

- Vertex ∗ findVertex (long id) const

  Auxiliary function to find a vertex with a given ID.
- bool addVertex (long id)

  Adds a vertex with a given content or info (in) to a graph (this).
- bool addEdge (long sourc, long dest, double d)

  Adds an edge to a graph (this), given the contents of the source and destination vertices and the edge weight (w).
- int getNumVertex () const
- std::unordered_map< long, Vertex ∗ > getVertexSet () const
- void print () const

  prints the graph
- bool TSP (std::vector< Vertex ∗ > &shortestPath, double &shortestPathCost)

  Check if the graph has a Hamiltonian cycle.
- bool hasHamiltonianCycle (std::vector< Vertex ∗ > &path, double &pathCost)

  Check if the graph has a Hamiltonian cycle.

Protected Member Functions

- void deleteVertex (long name)

  delete a vertex from the graph, making a subgraph from a graph
- double getPathCost (const std::vector< Vertex ∗ > &path)

  calculate the cost of the path
- bool TSPUtil (Vertex ∗v, std::vector< Vertex ∗ > &path, std::vector< Vertex ∗ > &shortestPath, double &shortestPathCost, int &numOfPossiblePaths, double &currentCost)

  Utility function to solve the TSP problem.
- double hasHamiltonianCycleUtil (Vertex ∗v, std::vector< Vertex ∗ > &path, double &pathCost)

  Utility function to check if the graph has a Hamiltonian cycle.
- bool hasPendantVertex ()

  Function to check for pendant vertices in the graph.
- bool hasArticulationPoint ()

  use Tarjan's Algorithm to find articulation points
- bool hasArticulationPointUtil (Vertex ∗pCurrentVertex, int time)

  Utility function to check if the graph contains a articulation point.

Protected Attributes

- std::unordered_map< long, Vertex ∗ > vertexSet
- double ∗∗ distMatrix = nullptr
- int ∗∗ pathMatrix = nullptr
- std::vector< int > disc
- std::vector< int > low
- std::vector< int > parent
- std::vector< bool > visited
- std::vector< bool > ap

3.3.1 Detailed Description

Definition at line 19 of file Graph.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 ~Graph()

```
Graph::~Graph ( )
```

Definition at line 63 of file Graph.cpp.

```
00063                  {
00064     deleteMatrix(distMatrix, vertexSet.size());
00065     deleteMatrix(pathMatrix, vertexSet.size());
00066 }
```

3.3.3 Member Function Documentation

3.3.3.1 addEdge()

```
bool Graph::addEdge (
                long sourc,
                long dest,
                double d )
```

Adds an edge to a graph (this), given the contents of the source and destination vertices and the edge weight (w).

Parameters

sourc	
dest	
w	

Returns

true if successful, and false if the source or destination vertex does not exist.

Definition at line 34 of file Graph.cpp.

```
00034                                                     {
00035     auto v1 = findVertex(sourc);
00036     auto v2 = findVertex(dest);
00037     if (v1 == nullptr || v2 == nullptr)
00038         return false;
00039     v1->addEdge(v2, d);
00040 
00041     return true;
00042 }
```

3.3.3.2 addVertex()

```
bool Graph::addVertex (
                long id )
```

Adds a vertex with a given content or info (in) to a graph (this).

Parameters

id	

Returns

true if successful, and false if a vertex with that content already exists.

Definition at line 26 of file Graph.cpp.

```
00026                                {
00027     if (findVertex(id) != nullptr)
00028         return false;
00029     vertexSet[id]=(new Vertex(id));
00030     return true;
00031 }
```

3.3.3.3 deleteVertex()

```
void Graph::deleteVertex (
            long name ) [protected]
```

delete a vertex from the graph, making a subgraph from a graph

Parameters

name	

Definition at line 86 of file Graph.cpp.

```
00086                                   {
00087     auto v = findVertex(name);
00088     for (auto e: v->getAdj()) {
00089         auto s = e->getDest()->getId();
00090         v->removeEdge(s);
00091     }
00092     for (auto e: v->getIncoming()) {
00093         e->getOrig()->removeEdge(name);
00094     }
00095     auto it = vertexSet.begin();
00096     while (it != vertexSet.end()) {
00097         auto currentVertex = *it;
00098         if (currentVertex.second->getId() == name) {
00099             it = vertexSet.erase(it);
00100         } else {
00101             it++;
00102         }
00103     }
00104 }
```

3.3.3.4 findVertex()

```
Vertex * Graph::findVertex (
            long id ) const
```

Auxiliary function to find a vertex with a given ID.

Parameters

id	

Returns

vertex pointer to vertex with given content, or nullptr if not found

Definition at line 17 of file Graph.cpp.

```
00017                                       {
00018     auto it = vertexSet.find(id);
00019     if(it!=vertexSet.end()){
00020         return it->second;
00021     }
00022     return nullptr;
00023 }
```

3.3.3.5 getNumVertex()

```
int Graph::getNumVertex ( ) const
```

Definition at line 8 of file Graph.cpp.

```
00008                               {
00009     return vertexSet.size();
00010 }
```

3.3.3.6 getPathCost()

```
double Graph::getPathCost (
            const std::vector< Vertex * > & path )  [protected]
```

calculate the cost of the path

Parameters

path	

Returns

double

Definition at line 119 of file Graph.cpp.

```
00119                                                    {
00120     double totalCost = 0;
00121     for (int i = 0; i < path.size() - 1; ++i) {
00122         for (auto edge: path[i]->getAdj()) {
00123             if (edge->getDest() == path[i + 1]) {
00124                 totalCost += edge->getDistance();
00125                 break;
00126             }
00127         }
00128     }
00129     return totalCost;
00130 }
```

3.3.3.7 getVertexSet()

```
std::unordered_map< long, Vertex * > Graph::getVertexSet ( ) const
```

Definition at line 12 of file Graph.cpp.

```
00012                                                       {
00013     return vertexSet;
00014 }
```

3.3.3.8 hasArticulationPoint()

```
bool Graph::hasArticulationPoint ( )  [protected]
```

use Tarjan's Algorithm to find articulation points

Attention

Time Complexity: O(V + E) (linear)

Returns

true/false

Definition at line 321 of file Graph.cpp.

```
00321                                  {
00322     int V = vertexSet.size();
00323     disc.assign(V, -1);
00324     low.assign(V, -1);
00325     parent.assign(V, -1);
00326     visited.assign(V, false);
00327     ap.assign(V, false);
00328
00329     for (auto vertex : vertexSet) {
00330         if (!visited[vertex.second->getId()]) {
00331             if (hasArticulationPointUtil(vertex.second, 0))
00332                 return true;
00333         }
00334     }
00335
00336     return false;
00337 }
```

3.3.3.9 hasArticulationPointUtil()

```
bool Graph::hasArticulationPointUtil (
            Vertex * pCurrentVertex,
            int time )  [protected]
```

Utility function to check if the graph contains a articulation point.

Parameters

pCurrentVertex	
time	

Returns

true/false

Definition at line 282 of file Graph.cpp.

```
00282                                                                    {
00283     int children = 0;
00284     long currentVertexIdInt = pCurrentVertex->getId();
00285     visited[currentVertexIdInt] = true;
00286     visited[currentVertexIdInt] = true;
00287
00288     disc[currentVertexIdInt] = low[currentVertexIdInt] = ++time;
00289
00290     for (auto edge : pCurrentVertex->getAdj()) {
```

```
        Vertex* pAdjacentVertex = edge->getDest();
        long adjacentVertexIdInt = pAdjacentVertex->getId();
        if (!visited[adjacentVertexIdInt]) {
            children++;
            parent[adjacentVertexIdInt] = currentVertexIdInt;

            if (hasArticulationPointUtil(pAdjacentVertex, time))
                return true;

            low[currentVertexIdInt] = std::min(low[currentVertexIdInt], low[adjacentVertexIdInt]);

            if (parent[currentVertexIdInt] == -1 && children > 1) {
                ap[currentVertexIdInt] = true;
                return true;
            }

            if (parent[currentVertexIdInt] != -1 && low[adjacentVertexIdInt] >= disc[currentVertexIdInt]) {
                ap[currentVertexIdInt] = true;
                return true;
            }
        }
        else if (adjacentVertexIdInt != parent[currentVertexIdInt]) {
            low[currentVertexIdInt] = std::min(low[currentVertexIdInt], disc[adjacentVertexIdInt]);
        }
    }

    return false;
}
```

3.3.3.10 hasHamiltonianCycle()

```
bool Graph::hasHamiltonianCycle (
            std::vector< Vertex * > & path,
            double & pathCost )
```

Check if the graph has a Hamiltonian cycle.

conditions:

- graph must be connected
- graph must not have pendant vertices
- graph must not have articulation points

Attention

Time Complexity: O(n!)

Note

Hamiltonian Cycle problem is NP-complete

Parameters

path	
pathCost	

Returns

Definition at line 246 of file Graph.cpp.

```
00246                                                                  {
00247
```

```
    if (this->vertexSet.empty()) {
        std::cout « "Graph is empty" « std::endl;
        return false;
    }

    if (hasPendantVertex()) {
        std::cout « "Graph has a pendant vertex" « std::endl;
        return false;
    }

    if(hasArticulationPoint()){
        std::cout « "Graph has an articulation point" « std::endl;
        return false;
    }

    // Start the timer
    auto start_time = std::chrono::high_resolution_clock::now();
    std::cout « "Searching for a Hamiltonian Cycle..." « std::endl;
    std::cout « "Please stand by..." « std::endl;

    // Measure execution time
    // ...
    path.push_back(this->vertexSet[0]);
    auto res = hasHamiltonianCycleUtil(this->vertexSet[0], path, pathCost);

    // End the timer
    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
    std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
    return res;
}
```

3.3.3.11 hasHamiltonianCycleUtil()

```
double Graph::hasHamiltonianCycleUtil (
            Vertex * v,
            std::vector< Vertex * > & path,
            double & pathCost )  [protected]
```

Utility function to check if the graph has a Hamiltonian cycle.

Parameters

v	
path	
pathCost	

Returns

double

Definition at line 219 of file Graph.cpp.

```
                                                                                  {
    if (path.size() == vertexSet.size()) {
        for (auto edge: v->getAdj()) {
            if (edge->getDest() == path[0]) {
                path.push_back(path[0]); // Closing the cycle
                pathCost = getPathCost(path);
                path.pop_back(); // Revert the cycle closing
                return true;  // found a Hamiltonian cycle
            }
        }
        return false;
    }

    for (auto edge: v->getAdj()) {
        Vertex *w = edge->getDest();
        if (std::find(path.begin(), path.end(), w) != path.end())
            continue;
        path.push_back(w);
```

```
        if (hasHamiltonianCycleUtil(w, path, pathCost))
            return true;  // propagate the success up the call stack
        path.pop_back();
    }

    return false;
}
```

3.3.3.12 hasPendantVertex()

bool Graph::hasPendantVertex ( ) [protected]

Function to check for pendant vertices in the graph.

Attention

Time complexity: O(V) (linear)

Returns

true if the graph has pendant vertices, false otherwise

Definition at line 108 of file Graph.cpp.

```
00108                                {
00109     for (auto v: vertexSet)
00110         if (v.second->getAdj().size() == 1) {
00111             std::cout « "Graph has pendant vertex: " « v.second->getId() « std::endl;
00112             return true;
00113         }
00114     return false;
00115 }
```

3.3.3.13 print()

void Graph::print ( ) const

prints the graph

Definition at line 69 of file Graph.cpp.

```
00069                           {
00070     std::cout « "---------------- Graph----------------\n";
00071     std::cout « "Number of vertices: " « vertexSet.size() « std::endl;
00072     std::cout « "Vertices:\n";
00073     for (const auto &vertex: vertexSet) {
00074         std::cout « vertex.second->getId() « " ";
00075     }
00076     std::cout « "\nEdges:\n";
00077     for (const auto &vertex: vertexSet) {
00078         for (const auto &edge: vertex.second->getAdj()) {
00079             std::cout « vertex.second->getId() « " -> " « edge->getDest()->getId() « " (distance: " «
      edge->getDistance()
00080                           « ")" « std::endl;
00081         }
00082     }
00083 }
```

3.3.3.14 TSP()

```
bool Graph::TSP (
            std::vector< Vertex * > & shortestPath,
            double & shortestPathCost )
```

Check if the graph has a Hamiltonian cycle.

(visit all nodes only once and return to the starting node) If it has, return the minimum cost cycle. conditions:

- graph must be connected
- graph must not have pendant vertices
- graph must not have articulation points

Attention

Time Complexity: O(n!)

Note

TSP is NP-hard problem, application to large graphs is infeasible

Parameters

shortestPath	
shortestPathCost	

Returns

true if the graph has a Hamiltonian cycle, false otherwise

Definition at line 180 of file Graph.cpp.

```
00180                                                                         {
00181     if (vertexSet.empty()) {
00182         std::cout « "Graph is empty" « std::endl;
00183         return false;
00184     }
00185
00186     if (hasPendantVertex()) {
00187         std::cout « "Graph has a pendant vertex" « std::endl;
00188         return false;
00189     }
00190
00191     if(hasArticulationPoint()){
00192         std::cout « "Graph has an articulation point" « std::endl;
00193         return false;
00194     }
00195
00196     // Start the timer
00197     auto start_time = std::chrono::high_resolution_clock::now();
00198
00199     std::cout « "Calculating TSP using backtracking..." « std::endl;
00200     std::cout « "Please stand by..." « std::endl;
00201
00202     // Measure execution time
00203     // ...
00204
00205     int numOfPossiblePaths = 0;
00206     std::vector<Vertex *> path;
00207     path.push_back(vertexSet[0]); // Start from any vertex
00208     double currentCost = 0;
00209     shortestPathCost = std::numeric_limits<double>::max(); // initialize to maximum possible double
00210     auto res = TSPUtil(vertexSet[0], path, shortestPath, shortestPathCost, numOfPossiblePaths,
      currentCost);
00211     std::cout « "Number of calculated paths: " « numOfPossiblePaths « std::endl;
00212     // End the timer
00213     auto end_time = std::chrono::high_resolution_clock::now();
```

```
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
    std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
    return res;
}
```

3.3.3.15 TSPUtil()

```
bool Graph::TSPUtil (
            Vertex * v,
            std::vector< Vertex * > & path,
            std::vector< Vertex * > & shortestPath,
            double & shortestPathCost,
            int & numOfPossiblePaths,
            double & currentCost )  [protected]
```

Utility function to solve the TSP problem.

Parameters

v	
path	
shortestPath	
shortestPathCost	
numOfPossiblePaths	

Returns

Definition at line 133 of file Graph.cpp.

```
{
    if (path.size() == vertexSet.size()) {
        for (auto edge: v->getAdj()) {
            if (edge->getDest() == path[0]) {
                path.push_back(path[0]);
                currentCost += edge->getDistance();  // Add the cost of returning to the start vertex

                if (currentCost < shortestPathCost) {  // Only consider path if it's the shortest so far
                    // Print path and its cost
                    std::cout « "Path: ";
                    for (auto vertex: path)
                        std::cout « vertex->getId() « " ";
                    std::cout « "Cost: " « currentCost « std::endl;
                    numOfPossiblePaths++;
                    shortestPath = path;
                    shortestPathCost = currentCost;
                }

                path.pop_back();
                currentCost -= edge->getDistance();  // Remove the cost of returning to the start vertex
                return true;
            }
        }
        return false;
    }

    for (auto edge: v->getAdj()) {
        Vertex *w = edge->getDest();
        if (std::find(path.begin(), path.end(), w) != path.end())
            continue;

        // If the current path cost plus the cost of the edge is already greater than the shortest path cost, skip
        if (currentCost + edge->getDistance() >= shortestPathCost)
            continue;

```

```
        path.push_back(w);
        currentCost += edge->getDistance();
        TSPUtil(w, path, shortestPath, shortestPathCost, numOfPossiblePaths, currentCost);
        path.pop_back();
        currentCost -= edge->getDistance();
    }

    return !shortestPath.empty();
}
```

3.3.4 Member Data Documentation

3.3.4.1 ap

`std::vector<bool> Graph::ap [protected]`

Definition at line 99 of file Graph.h.

3.3.4.2 disc

`std::vector<int> Graph::disc [protected]`

Definition at line 98 of file Graph.h.

3.3.4.3 distMatrix

`double** Graph::distMatrix = nullptr [protected]`

Definition at line 94 of file Graph.h.

3.3.4.4 low

`std::vector<int> Graph::low [protected]`

Definition at line 98 of file Graph.h.

3.3.4.5 parent

`std::vector<int> Graph::parent [protected]`

Definition at line 98 of file Graph.h.

3.3.4.6 pathMatrix

`int** Graph::pathMatrix = nullptr [protected]`

Definition at line 95 of file Graph.h.

3.3.4.7 vertexSet

```
std::unordered_map<long,Vertex *> Graph::vertexSet [protected]
```

Definition at line 92 of file Graph.h.

3.3.4.8 visited

```
std::vector<bool> Graph::visited [protected]
```

Definition at line 99 of file Graph.h.

The documentation for this class was generated from the following files:

- Graph.h
- Graph.cpp

3.4 MutablePriorityQueue< T > Class Template Reference

class T must have: (i) accessible field int queueIndex; (ii) operator< defined.

```
#include <MutablePriorityQueue.h>
```

Public Member Functions

- void insert (T *x)
- T * extractMin ()
- void decreaseKey (T *x)
- bool empty ()

3.4.1 Detailed Description

template<class T>
class MutablePriorityQueue< T >

class T must have: (i) accessible field int queueIndex; (ii) operator< defined.

Definition at line 21 of file MutablePriorityQueue.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 MutablePriorityQueue()

```
template<class T >
MutablePriorityQueue< T >::MutablePriorityQueue
```

Definition at line 39 of file MutablePriorityQueue.h.

```
00039                                               {
00040     H.push_back(nullptr);
00041     // indices will be used starting in 1
00042     // to facilitate parent/child calculations
00043 }
```

3.4.3 Member Function Documentation

3.4.3.1 decreaseKey()

```
template<class T >
void MutablePriorityQueue< T >::decreaseKey (
            T * x )
```

Definition at line 67 of file MutablePriorityQueue.h.

```
00067                                               {
00068     heapifyUp(x->queueIndex);
00069 }
```

3.4.3.2 empty()

```
template<class T >
bool MutablePriorityQueue< T >::empty
```

Definition at line 46 of file MutablePriorityQueue.h.

```
00046                                     {
00047     return H.size() == 1;
00048 }
```

3.4.3.3 extractMin()

```
template<class T >
T * MutablePriorityQueue< T >::extractMin
```

Definition at line 51 of file MutablePriorityQueue.h.

```
00051                                       {
00052     auto x = H[1];
00053     H[1] = H.back();
00054     H.pop_back();
00055     if(H.size() > 1) heapifyDown(1);
00056     x->queueIndex = 0;
00057     return x;
00058 }
```

3.4.3.4 insert()

```
template<class T >
void MutablePriorityQueue< T >::insert (
            T * x )
```

Definition at line 61 of file MutablePriorityQueue.h.

```
00061                                         {
00062     H.push_back(x);
00063     heapifyUp(H.size()-1);
00064 }
```

The documentation for this class was generated from the following file:

- MutablePriorityQueue.h

3.5 Vertex Class Reference

Public Member Functions

- Vertex (long id)
- bool operator$<$ (Vertex &vertex) const
- long getId () const
- std::vector$<$ Edge $*$ $>$ getAdj () const
- bool isVisited () const
- double getDist () const
- Edge $*$ getPath () const
- std::vector$<$ Edge $*$ $>$ getIncoming () const
- void setId (int info)
- void setVisited (bool visited)
- void setDist (double dist)
- void setPath (Edge $*$path)
- Edge $*$ addEdge (Vertex $*$dest, double w)
- bool removeEdge (long destID)
- Edge $*$ getEdge (long destID)
- void eraseChildren ()
- void addChildren (long s)
- std::vector$<$ long $>$ getChildren ()
- double getLatitude ()
- double getLongitude ()
- void setLatitude (double latitude)
- void setLongitude (double longitude)

Public Attributes

- int queueIndex = 0

Protected Member Functions

- void print () const

Protected Attributes

- long id
- std::vector$<$ Edge $*$ $>$ adj
- std::vector$<$ long $>$ children
- bool visited = false
- double dist = 0
- double longitude =0
- double latitude =0
- Edge $*$ path = nullptr
- std::vector$<$ Edge $*$ $>$ incoming

3.5.1 Detailed Description

Definition at line 22 of file VertexEdge.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Vertex()

```
Vertex::Vertex (
            long id )
```

Definition at line 9 of file VertexEdge.cpp.

```
00009 : id(id) {}
```

3.5.3 Member Function Documentation

3.5.3.1 addChildren()

```
void Vertex::addChildren (
            long s )
```

Definition at line 130 of file VertexEdge.cpp.

```
00130                            {
00131     children.push_back(s);
00132 }
```

3.5.3.2 addEdge()

```
Edge * Vertex::addEdge (
            Vertex * dest,
            double w )
```

Definition at line 15 of file VertexEdge.cpp.

```
00015                                            {
00016     auto newEdge = new Edge(this, d, dist);
00017     adj.push_back(newEdge);
00018     d->incoming.push_back(newEdge);
00019     return newEdge;
00020 }
```

3.5.3.3 eraseChildren()

```
void Vertex::eraseChildren ( )
```

Definition at line 126 of file VertexEdge.cpp.

```
00126                        {
00127     children.clear();
00128 }
```

3.5.3.4 getAdj()

```
std::vector< Edge * > Vertex::getAdj ( ) const
```

Definition at line 77 of file VertexEdge.cpp.

```
00077                                       {
00078     return this->adj;
00079 }
```

3.5.3.5 getChildren()

```
std::vector< long > Vertex::getChildren ( )
```

Definition at line 134 of file VertexEdge.cpp.

```
00134                                   {
00135     return children;
00136 }
```

3.5.3.6 getDist()

```
double Vertex::getDist ( ) const
```

Definition at line 85 of file VertexEdge.cpp.

```
00085                             {
00086     return this->dist;
00087 }
```

3.5.3.7 getEdge()

```
Edge * Vertex::getEdge (
            long destID )
```

Definition at line 53 of file VertexEdge.cpp.

```
00053                                  {
00054
00055     auto it = adj.begin();
00056     while (it != adj.end()) {
00057         Edge *edge = *it;
00058         Vertex *dest = edge->getDest();
00059         if (dest->getId() == destID) {
00060
00061             return edge; // allows for multiple edges to connect the same pair of vertices
      (multigraph)
00062         } else {
00063             it++;
00064         }
00065     }
00066     return nullptr;
00067 }
```

3.5.3.8 getId()

```
long Vertex::getId ( ) const
```

Definition at line 73 of file VertexEdge.cpp.

```
00073                         {
00074     return this->id;
00075 }
```

3.5.3.9 getIncoming()

```
std::vector< Edge * > Vertex::getIncoming ( ) const
```

Definition at line 93 of file VertexEdge.cpp.

```
00093                                             {
00094     return this->incoming;
00095 }
```

3.5.3.10 getLatitude()

```
double Vertex::getLatitude ( )
```

Definition at line 155 of file VertexEdge.cpp.

```
00155                          {
00156     return latitude;
00157 }
```

3.5.3.11 getLongitude()

```
double Vertex::getLongitude ( )
```

Definition at line 159 of file VertexEdge.cpp.

```
00159                           {
00160     return longitude;
00161 }
```

3.5.3.12 getPath()

```
Edge * Vertex::getPath ( ) const
```

Definition at line 89 of file VertexEdge.cpp.

```
00089                           {
00090     return this->path;
00091 }
```

3.5.3.13 isVisited()

```
bool Vertex::isVisited ( ) const
```

Definition at line 81 of file VertexEdge.cpp.

```
00081                            {
00082     return this->visited;
00083 }
```

3.5.3.14 operator<()

```
bool Vertex::operator< (
            Vertex & vertex ) const
```

Definition at line 69 of file VertexEdge.cpp.

```
00069                                          {
00070     return this->dist < vertex.dist;
00071 }
```

3.5.3.15 print()

```
void Vertex::print ( ) const  [protected]
```

Definition at line 114 of file VertexEdge.cpp.

```
00114                        {
00115     std::cout « "Vertex: " « id « std::endl;
00116     std::cout « "Adjacent to: ";
00117     for (const Edge *e: adj) {
00118         std::cout « e->getDest()->getId() « " ";
00119     }
00120     std::cout « std::endl;
00121     std::cout « "Visited: " « visited « std::endl;
00122     std::cout « "Distance: " « dist « std::endl;
00123     std::cout « "Path: " « path « std::endl;
00124 }
```

3.5.3.16 removeEdge()

```
bool Vertex::removeEdge (
            long destID )
```

Definition at line 27 of file VertexEdge.cpp.

```
00027                                    {
00028     bool removedEdge = false;
00029     auto it = adj.begin();
00030     while (it != adj.end()) {
00031         Edge *edge = *it;
00032         Vertex *dest = edge->getDest();
00033         if (dest->getId() == destID) {
00034             it = adj.erase(it);
00035             // Also remove the corresponding edge from the incoming list
00036             auto it2 = dest->incoming.begin();
00037             while (it2 != dest->incoming.end()) {
00038                 if ((*it2)->getOrig()->getId() == id) {
00039                     it2 = dest->incoming.erase(it2);
00040                 } else {
00041                     it2++;
00042                 }
00043             }
00044             delete edge;
00045             removedEdge = true; // allows for multiple edges to connect the same pair of vertices
      (multigraph)
00046         } else {
00047             it++;
00048         }
00049     }
00050     return removedEdge;
00051 }
```

3.5.3.17 setDist()

```
void Vertex::setDist (
            double dist )
```

Definition at line 105 of file VertexEdge.cpp.

```
00105                                  {
00106     this->dist = dist;
00107 }
```

3.5.3.18 setId()

```
void Vertex::setId (
            int info )
```

Definition at line 97 of file VertexEdge.cpp.

```
00097                            {
00098     this->id = id;
00099 }
```

3.5.3.19 setLatitude()

```
void Vertex::setLatitude (
            double latitude )
```

Definition at line 163 of file VertexEdge.cpp.

```
00163                                          {
00164     latitude=latitude_;
00165 }
```

3.5.3.20 setLongitude()

```
void Vertex::setLongitude (
            double longitude )
```

Definition at line 167 of file VertexEdge.cpp.

```
00167                                         {
00168     longitude=longitude_;
00169 }
```

3.5.3.21 setPath()

```
void Vertex::setPath (
            Edge * path )
```

Definition at line 109 of file VertexEdge.cpp.

```
00109                              {
00110     this->path = path;
00111 }
```

3.5.3.22 setVisited()

```
void Vertex::setVisited (
            bool visited )
```

Definition at line 101 of file VertexEdge.cpp.

```
00101                                   {
00102     this->visited = visited;
00103 }
```

3.5.4 Member Data Documentation

3.5.4.1 adj

```
std::vector<Edge *> Vertex::adj  [protected]
```

Definition at line 71 of file VertexEdge.h.

3.5.4.2 children

```
std::vector<long> Vertex::children  [protected]
```

Definition at line 72 of file VertexEdge.h.

3.5.4.3 dist

```
double Vertex::dist = 0  [protected]
```

Definition at line 76 of file VertexEdge.h.

3.5.4.4 id

```
long Vertex::id [protected]
```

Definition at line 70 of file VertexEdge.h.

3.5.4.5 incoming

```
std::vector<Edge *> Vertex::incoming [protected]
```

Definition at line 83 of file VertexEdge.h.

3.5.4.6 latitude

```
double Vertex::latitude =0 [protected]
```

Definition at line 78 of file VertexEdge.h.

3.5.4.7 longitude

```
double Vertex::longitude =0 [protected]
```

Definition at line 77 of file VertexEdge.h.

3.5.4.8 path

```
Edge* Vertex::path = nullptr [protected]
```

Definition at line 81 of file VertexEdge.h.

3.5.4.9 queueIndex

```
int Vertex::queueIndex = 0
```

Definition at line 68 of file VertexEdge.h.

3.5.4.10 visited

```
bool Vertex::visited = false [protected]
```

Definition at line 75 of file VertexEdge.h.

The documentation for this class was generated from the following files:

- VertexEdge.h
- VertexEdge.cpp

Chapter 4

File Documentation

4.1 CPheadquarters.cpp

```
//
// Created by david on 5/8/23.
//

#include <fstream>
#include <sstream>
#include "CPheadquarters.h"
#include "MutablePriorityQueue.h"
#include <chrono>
#include <set>
#include <cmath>
#include <string>

using namespace std;

void CPheadquarters::read_edges(string path){
    std::ifstream inputFile1(path);
    string line1;
    std::getline(inputFile1, line1); // ignore first line
    while (getline(inputFile1, line1, '\n')) {

        if (!line1.empty() && line1.back() == '\r') { // Check if the last character is '\r'
            line1.pop_back(); // Remove the '\r' character
        }

        string origin;
        string destination;
        string temp;
        double distance;


        stringstream inputString(line1);

        getline(inputString, origin, ',');
        getline(inputString, destination, ',');
        getline(inputString, temp, ',');


        distance = stod(temp);

        long origin_id = std::stol(origin);
        graph.addVertex(origin_id);

        long destination_id = std::stol(destination);
        graph.addVertex(destination_id);

        graph.addEdge(origin_id, destination_id, distance);
        graph.addEdge(destination_id, origin_id, distance);
        cout « origin « '\n';
    }
}


void CPheadquarters::read_coordinates(string path){
    std::ifstream inputFile2(path);
    string line2;
    std::getline(inputFile2, line2); // ignore first line

```

```

    while (getline(inputFile2, line2, '\n')) {

        if (!line2.empty() && line2.back() == '\n') { // Check if the last character is '\r'
            line2.pop_back(); // Remove the '\r' character
        }

        string id_;
        string temp1;
        string temp2;
        double longitude_;
        double latitude_;

        stringstream inputString(line2);

        getline(inputString, id_, ',');
        getline(inputString, temp1, ',');
        getline(inputString, temp2, ',');

        long long_id = std::stol(id_);
        longitude_ = stod(temp1);
        latitude_ = stod(temp2);

        auto v = graph.findVertex(long_id);
        v->setLongitude(longitude_);
        v->setLatitude(latitude_);

        cout « long_id « '\n';
    }
}


void CPheadquarters::heuristicRec(Vertex *v, long route[], unsigned int currentIndex, double distance, unsigned int &nodesVisited, double &totalDistance){

    bool nodesStillUnvisited = false;

    double long1 = v->getLongitude();
    double lat1 = v->getLatitude();

    Vertex *small;
    double smallAngle = 10000;
    double x;
    double y;
    double angle;
    double dist;

    for (const auto &edge: v->getAdj()) {
        Vertex *v2 = edge->getDest();
        double dist2 = edge->getDistance();
        if(v2->isVisited() == false){
            nodesStillUnvisited = true;

            double long2 = edge->getDest()->getLongitude();
            double lat2 = edge->getDest()->getLatitude();

            x = long1 - long2;
            y = lat1 - lat2;

            angle = atan2(y,x);

            if(angle < smallAngle){
                smallAngle = angle;
                small = v2;
                dist = dist2;
            }

        }
    }

    bool inRoute = false;
    for(int i = 0; i < currentIndex; i++){
        if(route[i] == v->getId()){
            inRoute = true;
        }
    }

    if(nodesStillUnvisited){
        route[currentIndex] = small->getId();
        small->setVisited(true);

        heuristicRec(small, route, currentIndex+1, distance + dist, nodesVisited, totalDistance);
    }
    else{
        nodesVisited = currentIndex;
        totalDistance = distance;

```

```
    }


}

void CPheadquarters::heuristic(long route[], unsigned int &nodesVisited, double &totalDistance, long id) {

    for (const auto vertex: graph.getVertexSet()) {
        vertex.second->setVisited(false);

    }


    Vertex *actual = graph.findVertex(id);

    double distance = 0;
    route[0] = actual->getId();

    actual->setVisited(true);

    heuristicRec(actual, route, 1, distance, nodesVisited, totalDistance);
}


void CPheadquarters::chooseRoute(){
    long id;
    auto pathSize = graph.getNumVertex();
    auto vertixes = graph.getVertexSet();
    long path[pathSize];
    unsigned int nodesVisited = 0;
    double distance = 0;
    for(auto it =vertixes.begin(); it != vertixes.end(); it++){
        id = it->first;
        heuristic(path, nodesVisited, distance, id);
        if(nodesVisited == pathSize){
            long sourceId = path[pathSize-1];
            long destId = path[0];
            Vertex *sourceV = graph.findVertex(sourceId);
            Edge *missingEdge = sourceV->getEdge(destId);
            if(missingEdge!= nullptr){
                distance += missingEdge->getDistance();
                for(int i = 0; i < pathSize; i++){
                    cout « path[i] « "->";
                }
                cout « destId;
                cout « "\nTotal distance: " « distance « endl;
                break;
            }
        }
    }
}



Graph CPheadquarters::getGraph() const {
    return this->graph;
}



void CPheadquarters::backtrack() {
    std::vector<Vertex*> shortestPath;
    double shortestPathCost = 0;


    if (this->graph.TSP(shortestPath, shortestPathCost)) {
        cout « "Shortest Hamiltonian cycle: ";
        for (auto vertex : shortestPath)
            cout « vertex->getId() « " ";
        cout « "\nCost: " « shortestPathCost « endl;
    }
    else {
        cout « "The graph does not have a Hamiltonian cycle" « endl;
    }
}

void CPheadquarters::hamiltonianCycle() {
    std::vector<Vertex*> path;
    double cost = 0;
    if (this->graph.hasHamiltonianCycle(path, cost)) {
        cout « "Hamiltonian cycle: ";
        for (auto vertex : path)
            cout « vertex->getId() « " ";
        cout « "\nCost: " « cost « endl;
    }
    else {
```

```
        cout « "The graph does not have a Hamiltonian cycle" « endl;
    }
}
void CPheadquarters::pathRec(Vertex* vertex){
    mst_preorder_path.push_back(vertex->getId());
    for (auto child : vertex->getChildren()) {
        pathRec(graph.getVertexSet()[child]);
    }
    return;
}


void CPheadquarters::triangular_Approximation_Heuristic() {
    std::unordered_map<long,Vertex *> vertexis = graph.getVertexSet();
    for (auto v: vertexis) {
        v.second->setVisited(false);
        v.second->setDist(std::numeric_limits<double>::max());
        v.second->eraseChildren();
    }

    Vertex *root = graph.findVertex(0);
    root->setDist(0);
    MutablePriorityQueue<Vertex> q;
    q.insert(root);
    while (!q.empty()) {
        auto v = q.extractMin();
        cout«"working on:"«v->getId()«'\n';
        v->setVisited(true);
        if(v->getId()!=0) {
            v->getPath()->getOrig()->addChildren(v->getId());
        }
        for (auto &e: v->getAdj()) {
            Vertex *w = e->getDest();
            if (!w->isVisited()) {
                auto oldDist = w->getDist();
                if (e->getDistance() < oldDist) {
                    w->setDist(e->getDistance());
                    w->setPath(e);
                    if (oldDist == std::numeric_limits<double>::max()) {
                        q.insert(w);
                    } else {
                        q.decreaseKey(w);
                    }
                }
            }
        }
    }

    mst_preorder_path.clear();
    pathRec(root);

    double result=0;

    for (int i = 0; i < mst_preorder_path.size()-1; i++) {
        result+= getDist(mst_preorder_path[i],mst_preorder_path[i+1]);
    }
    result+=getDist(mst_preorder_path[mst_preorder_path.size()-1],mst_preorder_path[0]);

    cout«"Result:"«result;

}

constexpr double EarthRadius = 6371.0;

double CPheadquarters::getDist(int a,int b){
    for (auto edge: graph.findVertex(a)->getAdj()){
        if (edge->getDest()->getId()==b) return edge->getDistance();
    }
    return haversineDistance(graph.findVertex(a)->getLatitude(),graph.findVertex(a)->getLongitude(), graph.findVertex(b)->getLatitude(), graph.findVertex(b)->getLongitude());
}



double CPheadquarters::degToRad (double degrees) {
    return degrees*M_PI/180.0;
}

double CPheadquarters::haversineDistance(double latitude1, double longitude1, double latitude2, double longitude2) {
    double ang_lat=degToRad(latitude2-latitude1);
    double ang_lon=degToRad(longitude2-longitude1);
    double a =sin(ang_lat / 2) * sin(ang_lat / 2) +
            cos(degToRad (latitude1)) * cos(degToRad (latitude2)) *
             sin(ang_lon / 2) * sin(ang_lon / 2);

    double c = 2 * atan2(sqrt(a), sqrt(1 - a));
```

```
    return EarthRadius * c;
}
```

4.2 CPheadquarters.h

```
//
// Created by david on 5/8/23.
//

#ifndef PROJECT_2_CPHEADQUARTERS_H
#define PROJECT_2_CPHEADQUARTERS_H


#include "Graph.h"

using namespace std;

class CPheadquarters {
    Graph graph;
    vector<long> mst_preorder_path;
public:

    void read_edges(std::string path);

    void read_coordinates(std::string path);


    Graph getGraph() const;

    void heuristic(long path[], unsigned int &nodesVisited, double &totalDistance, long id);

    void heuristicRec(Vertex *v, long path[], unsigned int currentIndex, double distance, unsigned int &nodesVisited, double &totalDistance);

    void chooseRoute();


    void backtrack();

    void hamiltonianCycle();


    void triangular_Approximation_Heuristic();

    void pathRec(Vertex *vertex);

    double degToRad(double degrees);

    double haversineDistance(double latitude1, double longitude1, double latitude2, double longitude2);

    double getDist(int a, int b);
};


#endif //PROJECT_2_CPHEADQUARTERS_H
```

4.3 Graph.cpp

```
#include <climits>
#include <queue>
#include "Graph.h"
#include <algorithm>
#include <unordered_set>
#include <chrono>

int Graph::getNumVertex() const {
    return vertexSet.size();
}

std::unordered_map<long,Vertex *> Graph::getVertexSet() const {
    return vertexSet;
}


Vertex *Graph:: findVertex(const long id) const {
    auto it = vertexSet.find(id);
    if(it!=vertexSet.end()){
```

```
        return it->second;
    }
    return nullptr;
}


bool Graph::addVertex(const long id) {
    if (findVertex(id) != nullptr)
        return false;
    vertexSet[id]=(new Vertex(id));
    return true;
}


bool Graph::addEdge(const long sourc, const long dest, double d) {
    auto v1 = findVertex(sourc);
    auto v2 = findVertex(dest);
    if (v1 == nullptr || v2 == nullptr)
        return false;
    v1->addEdge(v2, d);

    return true;
}


void deleteMatrix(int **m, int n) {
    if (m != nullptr) {
        for (int i = 0; i < n; i++)
            if (m[i] != nullptr)
                delete[] m[i];
        delete[] m;
    }
}

void deleteMatrix(double **m, int n) {
    if (m != nullptr) {
        for (int i = 0; i < n; i++)
            if (m[i] != nullptr)
                delete[] m[i];
        delete[] m;
    }
}

Graph::~Graph() {
    deleteMatrix(distMatrix, vertexSet.size());
    deleteMatrix(pathMatrix, vertexSet.size());
}


void Graph::print() const {
    std::cout « "---------------- Graph----------------\n";
    std::cout « "Number of vertices: " « vertexSet.size() « std::endl;
    std::cout « "Vertices:\n";
    for (const auto &vertex: vertexSet) {
        std::cout « vertex.second->getId() « " ";
    }
    std::cout « "\nEdges:\n";
    for (const auto &vertex: vertexSet) {
        for (const auto &edge: vertex.second->getAdj()) {
            std::cout « vertex.second->getId() « " -> " « edge->getDest()->getId() « " (distance: " «
edge->getDistance()
                      « ")" « std::endl;
        }
    }
}


void Graph::deleteVertex(long name) {
    auto v = findVertex(name);
    for (auto e: v->getAdj()) {
        auto s = e->getDest()->getId();
        v->removeEdge(s);
    }
    for (auto e: v->getIncoming()) {
        e->getOrig()->removeEdge(name);
    }
    auto it = vertexSet.begin();
    while (it != vertexSet.end()) {
        auto currentVertex = *it;
        if (currentVertex.second->getId() == name) {
            it = vertexSet.erase(it);
        } else {
            it++;
        }
    }
}

```

```
bool Graph::hasPendantVertex() {
    for (auto v: vertexSet)
        if (v.second->getAdj().size() == 1) {
            std::cout « "Graph has pendant vertex: " « v.second->getId() « std::endl;
            return true;
        }
    return false;
}


double Graph::getPathCost(const std::vector<Vertex *> &path) {
    double totalCost = 0;
    for (int i = 0; i < path.size() - 1; ++i) {
        for (auto edge: path[i]->getAdj()) {
            if (edge->getDest() == path[i + 1]) {
                totalCost += edge->getDistance();
                break;
            }
        }
    }
    return totalCost;
}


bool Graph::TSPUtil(Vertex *v, std::vector<Vertex *> &path, std::vector<Vertex *> &shortestPath, double &shortestPathCost,
                    int &numOfPossiblePaths, double &currentCost) {
    if (path.size() == vertexSet.size()) {
        for (auto edge: v->getAdj()) {
            if (edge->getDest() == path[0]) {
                path.push_back(path[0]);
                currentCost += edge->getDistance();  // Add the cost of returning to the start vertex

                if (currentCost < shortestPathCost) {  // Only consider path if it's the shortest so far
                    // Print path and its cost
                    std::cout « "Path: ";
                    for (auto vertex: path)
                        std::cout « vertex->getId() « " ";
                    std::cout « "Cost: " « currentCost « std::endl;
                    numOfPossiblePaths++;
                    shortestPath = path;
                    shortestPathCost = currentCost;
                }

                path.pop_back();
                currentCost -= edge->getDistance();  // Remove the cost of returning to the start vertex
                return true;
            }
        }
        return false;
    }

    for (auto edge: v->getAdj()) {
        Vertex *w = edge->getDest();
        if (std::find(path.begin(), path.end(), w) != path.end())
            continue;

        // If the current path cost plus the cost of the edge is already greater than the shortest path cost, skip
        if (currentCost + edge->getDistance() >= shortestPathCost)
            continue;

        path.push_back(w);
        currentCost += edge->getDistance();
        TSPUtil(w, path, shortestPath, shortestPathCost, numOfPossiblePaths, currentCost);
        path.pop_back();
        currentCost -= edge->getDistance();
    }

    return !shortestPath.empty();
}


bool Graph::TSP(std::vector<Vertex *> &shortestPath, double &shortestPathCost) {
    if (vertexSet.empty()) {
        std::cout « "Graph is empty" « std::endl;
        return false;
    }

    if (hasPendantVertex()) {
        std::cout « "Graph has a pendant vertex" « std::endl;
        return false;
```

```
    }

    if(hasArticulationPoint()){
        std::cout « "Graph has an articulation point" « std::endl;
        return false;
    }

    // Start the timer
    auto start_time = std::chrono::high_resolution_clock::now();

    std::cout « "Calculating TSP using backtracking..." « std::endl;
    std::cout « "Please stand by..." « std::endl;

    // Measure execution time
    // ...

    int numOfPossiblePaths = 0;
    std::vector<Vertex *> path;
    path.push_back(vertexSet[0]); // Start from any vertex
    double currentCost = 0;
    shortestPathCost = std::numeric_limits<double>::max(); // initialize to maximum possible double
    auto res = TSPUtil(vertexSet[0], path, shortestPath, shortestPathCost, numOfPossiblePaths, currentCost);
    std::cout « "Number of calculated paths: " « numOfPossiblePaths « std::endl;
    // End the timer
    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
    std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
    return res;
}

double Graph::hasHamiltonianCycleUtil(Vertex *v, std::vector<Vertex *> &path, double &pathCost) {
    if (path.size() == vertexSet.size()) {
        for (auto edge: v->getAdj()) {
            if (edge->getDest() == path[0]) {
                path.push_back(path[0]); // Closing the cycle
                pathCost = getPathCost(path);
                path.pop_back(); // Revert the cycle closing
                return true;  // found a Hamiltonian cycle
            }
        }
        return false;
    }

    for (auto edge: v->getAdj()) {
        Vertex *w = edge->getDest();
        if (std::find(path.begin(), path.end(), w) != path.end())
            continue;
        path.push_back(w);
        if (hasHamiltonianCycleUtil(w, path, pathCost))
            return true;  // propagate the success up the call stack
        path.pop_back();
    }

    return false;
}


bool Graph::hasHamiltonianCycle(std::vector<Vertex *> &path, double &pathCost) {

    if (this->vertexSet.empty()) {
        std::cout « "Graph is empty" « std::endl;
        return false;
    }

    if (hasPendantVertex()) {
        std::cout « "Graph has a pendant vertex" « std::endl;
        return false;
    }

    if(hasArticulationPoint()){
        std::cout « "Graph has an articulation point" « std::endl;
        return false;
    }

    // Start the timer
    auto start_time = std::chrono::high_resolution_clock::now();
    std::cout « "Searching for a Hamiltonian Cycle..." « std::endl;
    std::cout « "Please stand by..." « std::endl;

    // Measure execution time
    // ...
    path.push_back(this->vertexSet[0]);
    auto res = hasHamiltonianCycleUtil(this->vertexSet[0], path, pathCost);

    // End the timer
    auto end_time = std::chrono::high_resolution_clock::now();
```

```
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
    std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
    return res;
}



bool Graph::hasArticulationPointUtil(Vertex* pCurrentVertex, int time) {
    int children = 0;
    long currentVertexIdInt = pCurrentVertex->getId();
    visited[currentVertexIdInt] = true;
    visited[currentVertexIdInt] = true;

    disc[currentVertexIdInt] = low[currentVertexIdInt] = ++time;

    for (auto edge : pCurrentVertex->getAdj()) {
        Vertex* pAdjacentVertex = edge->getDest();
        long adjacentVertexIdInt = pAdjacentVertex->getId();
        if (!visited[adjacentVertexIdInt]) {
            children++;
            parent[adjacentVertexIdInt] = currentVertexIdInt;

            if (hasArticulationPointUtil(pAdjacentVertex, time))
                return true;

            low[currentVertexIdInt] = std::min(low[currentVertexIdInt], low[adjacentVertexIdInt]);

            if (parent[currentVertexIdInt] == -1 && children > 1) {
                ap[currentVertexIdInt] = true;
                return true;
            }

            if (parent[currentVertexIdInt] != -1 && low[adjacentVertexIdInt] >=
disc[currentVertexIdInt]) {
                ap[currentVertexIdInt] = true;
                return true;
            }
        }
        else if (adjacentVertexIdInt != parent[currentVertexIdInt]) {
            low[currentVertexIdInt] = std::min(low[currentVertexIdInt], disc[adjacentVertexIdInt]);
        }
    }

    return false;
}


bool Graph::hasArticulationPoint() {
    int V = vertexSet.size();
    disc.assign(V, -1);
    low.assign(V, -1);
    parent.assign(V, -1);
    visited.assign(V, false);
    ap.assign(V, false);

    for (auto vertex : vertexSet) {
        if (!visited[vertex.second->getId()]) {
            if (hasArticulationPointUtil(vertex.second, 0))
                return true;
        }
    }

    return false;
}
```

4.4 Graph.h

```
//
// Created by david on 5/8/23.
//

#ifndef PROJECT_2_GRAPH_H
#define PROJECT_2_GRAPH_H


#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <algorithm>
#include <unordered_set>
```

```
#include "VertexEdge.h"
class Graph {
public:
    ~Graph();
    Vertex *findVertex(long id) const;
    bool addVertex(long id);
    bool addEdge(long sourc, long dest, double d);
    [[nodiscard]] int getNumVertex() const;
    [[nodiscard]] std::unordered_map<long,Vertex *> getVertexSet() const;
    void print() const;
    bool TSP(std::vector<Vertex *> &shortestPath, double &shortestPathCost);
    bool hasHamiltonianCycle(std::vector<Vertex *> &path, double &pathCost);
protected:
    std::unordered_map<long,Vertex *> vertexSet;    // vertex set
    double **distMatrix = nullptr;   // dist matrix for Floyd-Warshall
    int **pathMatrix = nullptr;   // path matrix for Floyd-Warshall
    // for Tarjan's algorithm
    std::vector<int> disc, low, parent;
    std::vector<bool> visited, ap;
    void deleteVertex(long name);
    double getPathCost(const std::vector<Vertex *> &path);
    bool TSPUtil(Vertex *v, std::vector<Vertex *> &path, std::vector<Vertex *> &shortestPath, double &shortestPathCost,
                 int &numOfPossiblePaths, double &currentCost);
    double hasHamiltonianCycleUtil(Vertex *v, std::vector<Vertex *> &path, double &pathCost);
    bool hasPendantVertex();
    bool hasArticulationPoint();
    bool hasArticulationPointUtil(Vertex *pCurrentVertex, int time);
};
void deleteMatrix(int **m, int n);
void deleteMatrix(double **m, int n);
#endif //PROJECT_2_GRAPH_H
```

4.5 main.cpp

```
#include <iostream>
#include <chrono>
#include "CPheadquarters.h"
using namespace std;
int getValidInput(int lowerLimit, int upperLimit) {
    int n;
    bool validInput = false;
    while (!validInput) {
        cout « "Insert your option:\n";
        cin » n;
        if (cin.fail() || n < lowerLimit || n > upperLimit) {
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            cout « "Invalid input. Please enter a number between " « lowerLimit « " and " « upperLimit « "." « endl;
        } else {
```

```
            validInput = true;
        }
    }
    return n;
}
int main() {
    CPheadquarters CP;
    string path;
    cout « "Insert path to file to construct a graph "
            "\ne.g"
            "\n|Toy Graphs           |: ../Toy-Graphs/Toy-Graphs/shipping.csv"
            "\n|Real World Graphs    |: ../Real-world Graphs/Real-world Graphs/graph1/edges.csv"
            "\n|Extra Fully Connected|: ../Extra_Fully_Connected_Graphs/Extra_Fully_Connected_Graphs/edges_25.csv )"
            "\n:";
    getline(cin, path);
    CP.read_edges(path);
    cout«"If necessary, insert path to file that contains latitude and longitude"
          "\n(e.g ../Real-world Graphs/Real-world Graphs/graph1/nodes.csv)"
          "\n Otherwise, press enter."
          "\n:";
    getline(cin, path);
    cout«endl;
    if (!path.empty()) {
        CP.read_coordinates(path);
    }
    //CP.getGraph().print();
    int n;
    cout « "\n-------------- An Analysis Tool for Railway Network Management --------------\n" « endl;
    do {
        cout « "\n1 - T2.1 Backtracking Algorithm\n";
        cout « "2 - T2.2 Triangular Approximation Heuristic\n";
        cout « "3 - T2.3 other heuristic algorithms\n";
        cout « "8 - Exit\n";
        n = getValidInput(1, 8);
        switch (n) {
            case 1: {
                cout « "1 - TSP using Backtracking algorithm (for small graphs)\n";
                cout « "2 - Just find ANY the Hamiltonian Cycle (for big graphs)\n";
                int backtrack_choice;
                backtrack_choice = getValidInput(1, 2);
                switch(backtrack_choice){
                    case 1:
                        CP.backtrack();
                        break;
                    case 2:
                        CP.hamiltonianCycle();
                        break;
                    default: {
                        cerr « "Error: Invalid option selected." « endl;
                        break;
                    }
                }
                break;
            }
            case 2: {
                auto start_time = std::chrono::high_resolution_clock::now();
                CP.triangular_Approximation_Heuristic();
                auto end_time = std::chrono::high_resolution_clock::now();
                auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
                std::cout « "Time taken: " « duration.count() « " ms" « std::endl;
                break;
            }
            case 3: {
                auto start_time = std::chrono::high_resolution_clock::now();
                CP.chooseRoute();
                // Code block to measure goes here
                // End the timer
                auto end_time = std::chrono::high_resolution_clock::now();
```

```
                // Compute the duration
                auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time -
    start_time);
                // Print the duration
                std::cout « "Time taken: " « duration.count() « " ms" « std::endl;


                break;
            }

            case 8: {
                cout « "Exiting program..." « endl;
                break;
            }

            default: {
                cerr « "Error: Invalid option selected." « endl;
                break;
            }
        }
    } while (n != 8);

    return 0;
}
```

4.6 MutablePriorityQueue.h

```
/*
 * MutablePriorityQueue.h
 * A simple implementation of mutable priority queues, required by Dijkstra algorithm.
 *
 * Created on: 17/03/2018
 *      Author: João Pascoal Faria
 */

#ifndef DA_TP_CLASSES_MUTABLEPRIORITYQUEUE
#define DA_TP_CLASSES_MUTABLEPRIORITYQUEUE

#include <vector>



template <class T>
class MutablePriorityQueue {
    std::vector<T *> H;
    void heapifyUp(unsigned i);
    void heapifyDown(unsigned i);
    inline void set(unsigned i, T * x);
public:
    MutablePriorityQueue();
    void insert(T * x);
    T * extractMin();
    void decreaseKey(T * x);
    bool empty();
};

// Index calculations
#define parent(i) ((i) / 2)
#define leftChild(i) ((i) * 2)

template <class T>
MutablePriorityQueue<T>::MutablePriorityQueue() {
    H.push_back(nullptr);
    // indices will be used starting in 1
    // to facilitate parent/child calculations
}

template <class T>
bool MutablePriorityQueue<T>::empty() {
    return H.size() == 1;
}

template <class T>
T* MutablePriorityQueue<T>::extractMin() {
    auto x = H[1];
    H[1] = H.back();
    H.pop_back();
    if(H.size() > 1) heapifyDown(1);
    x->queueIndex = 0;
    return x;
}

template <class T>
```

```
void MutablePriorityQueue<T>::insert(T *x) {
    H.push_back(x);
    heapifyUp(H.size()-1);
}

template <class T>
void MutablePriorityQueue<T>::decreaseKey(T *x) {
    heapifyUp(x->queueIndex);
}

template <class T>
void MutablePriorityQueue<T>::heapifyUp(unsigned i) {
    auto x = H[i];
    while (i > 1 && *x < *H[parent(i)]) {
        set(i, H[parent(i)]);
        i = parent(i);
    }
    set(i, x);
}

template <class T>
void MutablePriorityQueue<T>::heapifyDown(unsigned i) {
    auto x = H[i];
    while (true) {
        unsigned k = leftChild(i);
        if (k >= H.size())
            break;
        if (k+1 < H.size() && *H[k+1] < *H[k])
            ++k; // right child of i
        if ( ! (*H[k] < *x) )
            break;
        set(i, H[k]);
        i = k;
    }
    set(i, x);
}

template <class T>
void MutablePriorityQueue<T>::set(unsigned i, T * x) {
    H[i] = x;
    x->queueIndex = i;
}

#endif /* DA_TP_CLASSES_MUTABLEPRIORITYQUEUE */
```

4.7 VertexEdge.cpp

```
//
// Created by david on 5/8/23.
//

#include "VertexEdge.h"

/************************* Vertex  **************************/

Vertex::Vertex(long id) : id(id) {}

/*
 * Auxiliary function to add an outgoing edge to a vertex (this),
 * with a given destination vertex (d) and edge weight (w).
 */
Edge *Vertex::addEdge(Vertex *d, double dist) {
    auto newEdge = new Edge(this, d, dist);
    adj.push_back(newEdge);
    d->incoming.push_back(newEdge);
    return newEdge;
}

/*
 * Auxiliary function to remove an outgoing edge (with a given destination (d))
 * from a vertex (this).
 * Returns true if successful, and false if such edge does not exist.
 */
bool Vertex::removeEdge(long destID) {
    bool removedEdge = false;
    auto it = adj.begin();
    while (it != adj.end()) {
        Edge *edge = *it;
        Vertex *dest = edge->getDest();
        if (dest->getId() == destID) {
            it = adj.erase(it);
            // Also remove the corresponding edge from the incoming list
            auto it2 = dest->incoming.begin();
```

```
            while (it2 != dest->incoming.end()) {
                if ((*it2)->getOrig()->getId() == id) {
                    it2 = dest->incoming.erase(it2);
                } else {
                    it2++;
                }
            }
            delete edge;
            removedEdge = true; // allows for multiple edges to connect the same pair of vertices (multigraph)
        } else {
            it++;
        }
    }
    return removedEdge;
}

Edge *Vertex::getEdge(long destID){

    auto it = adj.begin();
    while (it != adj.end()) {
        Edge *edge = *it;
        Vertex *dest = edge->getDest();
        if (dest->getId() == destID) {

            return edge; // allows for multiple edges to connect the same pair of vertices (multigraph)
        } else {
            it++;
        }
    }
    return nullptr;
}

bool Vertex::operator<(Vertex &vertex) const {
    return this->dist < vertex.dist;
}

long Vertex::getId() const {
    return this->id;
}

std::vector<Edge *> Vertex::getAdj() const {
    return this->adj;
}

bool Vertex::isVisited() const {
    return this->visited;
}

double Vertex::getDist() const {
    return this->dist;
}

Edge *Vertex::getPath() const {
    return this->path;
}

std::vector<Edge *> Vertex::getIncoming() const {
    return this->incoming;
}

void Vertex::setId(int id) {
    this->id = id;
}

void Vertex::setVisited(bool visited) {
    this->visited = visited;
}

void Vertex::setDist(double dist) {
    this->dist = dist;
}

void Vertex::setPath(Edge *path) {
    this->path = path;
}


void Vertex::print() const {
    std::cout « "Vertex: " « id « std::endl;
    std::cout « "Adjacent to: ";
    for (const Edge *e: adj) {
        std::cout « e->getDest()->getId() « " ";
    }
    std::cout « std::endl;
    std::cout « "Visited: " « visited « std::endl;
```

```
    std::cout « "Distance: " « dist « std::endl;
    std::cout « "Path: " « path « std::endl;
}

void Vertex::eraseChildren() {
    children.clear();
}

void Vertex::addChildren(long s) {
    children.push_back(s);
}

std::vector<long> Vertex::getChildren() {
    return children;
}


/********************** Edge  ****************************/

Edge::Edge(Vertex *orig, Vertex *dest, double d) : orig(orig), dest(dest), distance(d) {}

Vertex *Edge::getDest() const {
    return this->dest;
}

double Edge::getDistance() const {
    return this->distance;
}

Vertex *Edge::getOrig() const {
    return this->orig;
}

double Vertex::getLatitude() {
    return latitude;
}

double Vertex::getLongitude() {
    return longitude;
}

void Vertex::setLatitude(double latitude_) {
    latitude=latitude_;
}

void Vertex::setLongitude(double longitude_) {
    longitude=longitude_;
}



```

4.8 VertexEdge.h

```
//
// Created by david on 5/8/23.
//

#ifndef PROJECT_2_VERTEXEDGE_H
#define PROJECT_2_VERTEXEDGE_H


#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <algorithm>


class Edge;

#define INF std::numeric_limits<double>::max()

/************************* Vertex  **************************/

class Vertex {
public:
    Vertex(long id);

    bool operator<(Vertex &vertex) const; // // required by MutablePriorityQueue

    long getId() const;

```

```
    std::vector<Edge *> getAdj() const;

    bool isVisited() const;

    double getDist() const;

    Edge *getPath() const;

    std::vector<Edge *> getIncoming() const;

    void setId(int info);

    void setVisited(bool visited);

    void setDist(double dist);

    void setPath(Edge *path);

    Edge *addEdge(Vertex *dest, double w);

    bool removeEdge(long destID);

    Edge *getEdge(long destID);

    void eraseChildren();

    void addChildren(long s);

    std::vector<long> getChildren();

    double getLatitude();

    double getLongitude();

    void setLatitude(double latitude);

    void setLongitude(double longitude);

    int queueIndex = 0;
protected:
    long id;            // identifier
    std::vector<Edge *> adj;  // outgoing edges
    std::vector<long> children;

    // auxiliary fields
    bool visited = false; // used by DFS, BFS, Prim ...
    double dist = 0;
    double longitude=0;
    double latitude=0;


    Edge *path = nullptr;

    std::vector<Edge *> incoming; // incoming edges

    // required by MutablePriorityQueue and UFDS
    void print() const;

};


/********************** Edge  ***************************/

class Edge {
public:
    Edge(Vertex *orig, Vertex *dest, double d);

    Vertex *getDest() const;

    double getDistance() const;

    Vertex *getOrig() const;

protected:
    Vertex *dest; // destination vertex
    double distance; // edge weight, can also be used for capacity

    // used for bidirectional edges
    Vertex *orig;

};



#endif //PROJECT_2_VERTEXEDGE_H
```

Index