

# Routing Algorithm for Ocean Shipping and Urban Deliveries

1.0

Generated by Doxygen 1.9.7



|   |          |
|---|----------|
| <b>1 Class Index</b>                          | <b>1</b> |
| 1.1 Class List                                | 1        |
| <b>2 File Index</b>                           | <b>3</b> |
| 2.1 File List                                 | 3        |
| <b>3 Class Documentation</b>                  | <b>5</b> |
| 3.1 CPheadquarters Class Reference            | 5        |
| 3.1.1 Detailed Description                    | 5        |
| 3.1.2 Member Function Documentation           | 6        |
| 3.1.2.1 backtrack()                           | 6        |
| 3.1.2.2 chooseRoute()                         | 6        |
| 3.1.2.3 degToRad()                            | 7        |
| 3.1.2.4 getDist()                             | 8        |
| 3.1.2.5 getGraph()                            | 8        |
| 3.1.2.6 hamiltonianCycle()                    | 9        |
| 3.1.2.7 haversineDistance()                   | 9        |
| 3.1.2.8 heuristic()                           | 10       |
| 3.1.2.9 heuristicRec()                        | 10       |
| 3.1.2.10 pathRec()                            | 11       |
| 3.1.2.11 read_coordinates()                   | 12       |
| 3.1.2.12 read_edges()                         | 12       |
| 3.1.2.13 triangular_Approximation_Heuristic() | 13       |
| 3.2 Edge Class Reference                      | 14       |
| 3.2.1 Detailed Description                    | 14       |
| 3.2.2 Constructor & Destructor Documentation  | 14       |
| 3.2.2.1 Edge()                                | 14       |
| 3.2.3 Member Function Documentation           | 15       |
| 3.2.3.1 getDest()                             | 15       |
| 3.2.3.2 getDistance()                         | 15       |
| 3.2.3.3 getOrig()                             | 15       |
| 3.2.4 Member Data Documentation               | 15       |
| 3.2.4.1 dest                                  | 15       |
| 3.2.4.2 distance                              | 15       |
| 3.2.4.3 orig                                  | 15       |
| 3.3 Graph Class Reference                     | 16       |
| 3.3.1 Detailed Description                    | 17       |
| 3.3.2 Constructor & Destructor Documentation  | 17       |
| 3.3.2.1 ~Graph()                              | 17       |
| 3.3.3 Member Function Documentation           | 17       |
| 3.3.3.1 addEdge()                             | 17       |
| 3.3.3.2 addVertex()                           | 17       |
| 3.3.3.3 deleteVertex()                        | 18       |

|  |    |
|--|----|
| 3.3.3.4 findVertex()                                   | 18 |
| 3.3.3.5 getNumVertex()                                 | 19 |
| 3.3.3.6 getPathCost()                                  | 19 |
| 3.3.3.7 getVertexSet()                                 | 19 |
| 3.3.3.8 hasArticulationPoint()                         | 20 |
| 3.3.3.9 hasArticulationPointUtil()                     | 20 |
| 3.3.3.10 hasHamiltonianCycle()                         | 21 |
| 3.3.3.11 hasHamiltonianCycleUtil()                     | 22 |
| 3.3.3.12 hasPendantVertex()                            | 23 |
| 3.3.3.13 print()                                       | 23 |
| 3.3.3.14 TSP()   | 24 |
| 3.3.3.15 TSPUtil()                                     | 25 |
| 3.3.4 Member Data Documentation                        | 26 |
| 3.3.4.1 ap   | 26 |
| 3.3.4.2 disc   | 26 |
| 3.3.4.3 distMatrix                                     | 26 |
| 3.3.4.4 low  | 26 |
| 3.3.4.5 parent   | 26 |
| 3.3.4.6 pathMatrix                                     | 26 |
| 3.3.4.7 vertexSet                                      | 27 |
| 3.3.4.8 visited  | 27 |
| 3.4 MutablePriorityQueue< T > Class Template Reference | 27 |
| 3.4.1 Detailed Description                             | 27 |
| 3.4.2 Constructor & Destructor Documentation           | 27 |
| 3.4.2.1 MutablePriorityQueue()                         | 27 |
| 3.4.3 Member Function Documentation                    | 28 |
| 3.4.3.1 decreaseKey()                                  | 28 |
| 3.4.3.2 empty()  | 28 |
| 3.4.3.3 extractMin()                                   | 28 |
| 3.4.3.4 insert()                                       | 28 |
| 3.5 Vertex Class Reference                             | 29 |
| 3.5.1 Detailed Description                             | 29 |
| 3.5.2 Constructor & Destructor Documentation           | 30 |
| 3.5.2.1 Vertex()                                       | 30 |
| 3.5.3 Member Function Documentation                    | 30 |
| 3.5.3.1 addChildren()                                  | 30 |
| 3.5.3.2 addEdge()                                      | 30 |
| 3.5.3.3 eraseChildren()                                | 30 |
| 3.5.3.4 getAdj()                                       | 30 |
| 3.5.3.5 getChildren()                                  | 31 |
| 3.5.3.6 getDist()                                      | 31 |
| 3.5.3.7 getEdge()                                      | 31 |

|                                 |           |
|---------------------------------|-----------|
| 3.5.3.8 getId()                 | 31        |
| 3.5.3.9 getIncoming()           | 31        |
| 3.5.3.10 getLatitude()          | 32        |
| 3.5.3.11 getLongitude()         | 32        |
| 3.5.3.12 getPath()              | 32        |
| 3.5.3.13 isVisited()            | 32        |
| 3.5.3.14 operator<()            | 32        |
| 3.5.3.15 print()                | 32        |
| 3.5.3.16 removeEdge()           | 33        |
| 3.5.3.17 setDist()              | 33        |
| 3.5.3.18 setId()                | 33        |
| 3.5.3.19 setLatitude()          | 33        |
| 3.5.3.20 setLongitude()         | 34        |
| 3.5.3.21 setPath()              | 34        |
| 3.5.3.22 setVisited()           | 34        |
| 3.5.4 Member Data Documentation | 34        |
| 3.5.4.1 adj                     | 34        |
| 3.5.4.2 children                | 34        |
| 3.5.4.3 dist                    | 34        |
| 3.5.4.4 id                      | 35        |
| 3.5.4.5 incoming                | 35        |
| 3.5.4.6 latitude                | 35        |
| 3.5.4.7 longitude               | 35        |
| 3.5.4.8 path                    | 35        |
| 3.5.4.9 queueIndex              | 35        |
| 3.5.4.10 visited                | 35        |
| <b>4 File Documentation</b>     | <b>37</b> |
| 4.1 CPheadquarters.cpp          | 37        |
| 4.2 CPheadquarters.h            | 41        |
| 4.3 Graph.cpp                   | 41        |
| 4.4 Graph.h                     | 45        |
| 4.5 main.cpp                    | 46        |
| 4.6 MutablePriorityQueue.h      | 48        |
| 4.7 VertexEdge.cpp              | 49        |
| 4.8 VertexEdge.h                | 51        |
| <b>Index</b>                    | <b>53</b> |



# Chapter 1

## Class Index

### 1.1 Class List

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

|  |    |
|--|----|
| <a href="#">CPheadquarters</a>   | 5  |
| <a href="#">Edge</a>   | 14 |
| <a href="#">Graph</a>  | 16 |
| <a href="#">MutablePriorityQueue&lt; T &gt;</a>                                |    |
| Class T must have: (i) accessible field int queueIndex; (ii) operator< defined | 27 |
| <a href="#">Vertex</a>   | 29 |





## Chapter 2

# File Index

### 2.1 File List

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

|  |    |
|--|----|
| <a href="#">CPheadquarters.cpp</a>     | ?? |
| <a href="#">CPheadquarters.h</a>       | ?? |
| <a href="#">Graph.cpp</a>              | ?? |
| <a href="#">Graph.h</a>                | ?? |
| <a href="#">main.cpp</a>               | ?? |
| <a href="#">MutablePriorityQueue.h</a> | ?? |
| <a href="#">VertexEdge.cpp</a>         | ?? |
| <a href="#">VertexEdge.h</a>           | ?? |



## 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 &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.*
- 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 vertexes = graph.getVertexSet();
00173             long path[pathSize];
00174             unsigned int nodesVisited = 0;
00175             double distance = 0;
00176             for(auto it =vertexes.begin(); it != vertexes.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**

|                |  |
|----------------|--|
| <i>degrees</i> |  |
|----------------|--|

**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**

|          |  |
|----------|--|
| <i>a</i> |  |
| <i>b</i> |  |

**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(),
00299                             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

|                   |  |
|-------------------|--|
| <i>latitude1</i>  |  |
| <i>longitude1</i> |  |
| <i>latitude2</i>  |  |
| <i>longitude2</i> |  |

#### 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

|                      |  |
|----------------------|--|
| <i>route</i>         |  |
| <i>nodesVisited</i>  |  |
| <i>totalDistance</i> |  |
| <i>id</i>            |  |

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

|                      |  |
|----------------------|--|
| <i>v</i>             |  |
| <i>route</i>         |  |
| <i>currentIndex</i>  |  |
| <i>distance</i>      |  |
| <i>nodesVisited</i>  |  |
| <i>totalDistance</i> |  |



**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**

|               |
|---------------|
| <i>vertex</i> |
|---------------|

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

```

00028         string temp;
00029         double distance;
00030
00031
00032         stringstream inputString(line1);
00033
00034         getline(inputString, origin, ',');
00035         getline(inputString, destination, ',');
00036         getline(inputString, temp, ',');
00037
00038
00039         distance = stod(temp);
00040
00041         long origin_id = std::stol(origin);
00042         graph.addVertex(origin_id);
00043
00044         long destination_id = std::stol(destination);
00045         graph.addVertex(destination_id);
00046
00047         graph.addEdge(origin_id, destination_id, distance);
00048         graph.addEdge(destination_id, origin_id, distance);
00049         cout << origin << '\n';
00050     }
00051 }

```

### 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 algorithm with complexity  $O((V+E)\log V)$ . 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:  $O((V+E)\log V + 2E)$

Definition at line 243 of file [CPheadquarters.cpp](#).

```

00243                                     {
00244         std::unordered_map<long,Vertex *> vertexis = graph.getVertexSet();
00245         for (auto v: vertexis) {
00246             v.second->setVisited(false);
00247             v.second->setDist(std::numeric_limits<double>::max());
00248             v.second->eraseChildren();
00249         }
00250
00251         Vertex *root = graph.findVertex(0);
00252         root->setDist(0);
00253         MutablePriorityQueue<Vertex> q;
00254         q.insert(root);
00255         while (!q.empty()) {
00256             auto v = q.extractMin();
00257             cout<<"working on:"<<v->getId()<<'\n';
00258             v->setVisited(true);
00259             if (v->getId() != 0) {
00260                 v->getPath()->getOrig()->addChildren(v->getId());
00261             }
00262             for (auto &e: v->getAdj()) {
00263                 Vertex *w = e->getDest();
00264                 if (!w->isVisited()) {
00265                     auto oldDist = w->getDist();
00266                     if (e->getDistance() < oldDist) {
00267                         w->setDist(e->getDistance());
00268                         w->setPath(e);
00269                         if (oldDist == std::numeric_limits<double>::max()) {
00270                             q.insert(w);
00271                         } else {
00272                             q.decreaseKey(w);
00273                         }
00274                     }
00275                 }
00276             }
00277         }
00278     }

```

```

00279     mst_preorder_path.clear();
00280     pathRec(root);
00281
00282     double result=0;
00283
00284     for (int i = 0; i < mst_preorder_path.size()-1; i++) {
00285         result+= getDist(mst_preorder_path[i],mst_preorder_path[i+1]);
00286     }
00287     result+=getDist(mst_preorder_path[mst_preorder_path.size()-1],mst_preorder_path[0]);
00288
00289     cout<<"Result:"<<result;
00290
00291 }

```

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 source,
    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

|               |  |
|---------------|--|
| <i>source</i> |  |
| <i>dest</i>   |  |
| <i>w</i>      |  |

##### 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(source);
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

|           |  |
|-----------|--|
| <i>id</i> |  |
|-----------|--|

## 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

|             |  |
|-------------|--|
| <i>name</i> |  |
|-------------|--|

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

|           |  |
|-----------|--|
| <i>id</i> |  |
|-----------|--|



**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**

|             |  |
|-------------|--|
| <i>path</i> |  |
|-------------|--|

**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

|                       |  |
|-----------------------|--|
| <i>pCurrentVertex</i> |  |
| <i>time</i>           |  |

#### 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()) {
```

```

00291     Vertex* pAdjacentVertex = edge->getDest();
00292     long adjacentVertexIdInt = pAdjacentVertex->getId();
00293     if (!visited[adjacentVertexIdInt]) {
00294         children++;
00295         parent[adjacentVertexIdInt] = currentVertexIdInt;
00296
00297         if (hasArticulationPointUtil(pAdjacentVertex, time))
00298             return true;
00299
00300         low[currentVertexIdInt] = std::min(low[currentVertexIdInt], low[adjacentVertexIdInt]);
00301
00302         if (parent[currentVertexIdInt] == -1 && children > 1) {
00303             ap[currentVertexIdInt] = true;
00304             return true;
00305         }
00306
00307         if (parent[currentVertexIdInt] != -1 && low[adjacentVertexIdInt] >=
disc[currentVertexIdInt]) {
00308             ap[currentVertexIdInt] = true;
00309             return true;
00310         }
00311     }
00312     else if (adjacentVertexIdInt != parent[currentVertexIdInt]) {
00313         low[currentVertexIdInt] = std::min(low[currentVertexIdInt], disc[adjacentVertexIdInt]);
00314     }
00315 }
00316
00317 return false;
00318 }

```

### 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

|                 |  |
|-----------------|--|
| <i>path</i>     |  |
| <i>pathCost</i> |  |

#### Returns

Definition at line 246 of file [Graph.cpp](#).

```

00246
00247

```

```

00248     if (this->vertexSet.empty()) {
00249         std::cout << "Graph is empty" << std::endl;
00250         return false;
00251     }
00252
00253     if (hasPendantVertex()) {
00254         std::cout << "Graph has a pendant vertex" << std::endl;
00255         return false;
00256     }
00257
00258     if (hasArticulationPoint()) {
00259         std::cout << "Graph has an articulation point" << std::endl;
00260         return false;
00261     }
00262
00263     // Start the timer
00264     auto start_time = std::chrono::high_resolution_clock::now();
00265     std::cout << "Searching for a Hamiltonian Cycle..." << std::endl;
00266     std::cout << "Please stand by..." << std::endl;
00267
00268     // Measure execution time
00269     // ...
00270     path.push_back(this->vertexSet[0]);
00271     auto res = hasHamiltonianCycleUtil(this->vertexSet[0], path, pathCost);
00272
00273     // End the timer
00274     auto end_time = std::chrono::high_resolution_clock::now();
00275     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00276     std::cout << "Time taken: " << duration.count() << " ms" << std::endl;
00277     return res;
00278 }

```

### 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

|                 |  |
|-----------------|--|
| <i>v</i>        |  |
| <i>path</i>     |  |
| <i>pathCost</i> |  |

#### Returns

double

Definition at line 219 of file [Graph.cpp](#).

```

00219     {
00220         if (path.size() == vertexSet.size()) {
00221             for (auto edge: v->getAdj()) {
00222                 if (edge->getDest() == path[0]) {
00223                     path.push_back(path[0]); // Closing the cycle
00224                     pathCost = getPathCost(path);
00225                     path.pop_back(); // Revert the cycle closing
00226                     return true; // found a Hamiltonian cycle
00227                 }
00228             }
00229             return false;
00230         }
00231
00232         for (auto edge: v->getAdj()) {
00233             Vertex *w = edge->getDest();
00234             if (std::find(path.begin(), path.end(), w) != path.end())
00235                 continue;
00236             path.push_back(w);

```

```

00237         if (hasHamiltonianCycleUtil(w, path, pathCost))
00238             return true; // propagate the success up the call stack
00239         path.pop_back();
00240     }
00241
00242     return false;
00243 }

```

### 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

|                         |  |
|-------------------------|--|
| <i>shortestPath</i>     |  |
| <i>shortestPathCost</i> |  |

#### 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,
00211         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();
```

```

00214     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00215     std::cout << "Time taken: " << duration.count() << " ms" << std::endl;
00216     return res;
00217 }

```

### 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

|                           |  |
|---------------------------|--|
| <i>v</i>                  |  |
| <i>path</i>               |  |
| <i>shortestPath</i>       |  |
| <i>shortestPathCost</i>   |  |
| <i>numOfPossiblePaths</i> |  |

#### Returns

Definition at line 133 of file [Graph.cpp](#).

```

00134                                                                 {
00135     if (path.size() == vertexSet.size()) {
00136         for (auto edge: v->getAdj()) {
00137             if (edge->getDest() == path[0]) {
00138                 path.push_back(path[0]);
00139                 currentCost += edge->getDistance(); // Add the cost of returning to the start vertex
00140
00141                 if (currentCost < shortestPathCost) { // Only consider path if it's the shortest so
00142                     far
00143                         // Print path and its cost
00144                         std::cout << "Path: ";
00145                         for (auto vertex: path)
00146                             std::cout << vertex->getId() << " ";
00147                         std::cout << "Cost: " << currentCost << std::endl;
00148                         numOfPossiblePaths++;
00149                         shortestPath = path;
00150                         shortestPathCost = currentCost;
00151                     }
00152                     path.pop_back();
00153                     currentCost -= edge->getDistance(); // Remove the cost of returning to the start
00154                 vertex
00155                     return true;
00156             }
00157         }
00158         return false;
00159     }
00160     for (auto edge: v->getAdj()) {
00161         Vertex *w = edge->getDest();
00162         if (std::find(path.begin(), path.end(), w) != path.end())
00163             continue;
00164
00165         // If the current path cost plus the cost of the edge is already greater than the shortest
00166         path cost, skip
00167         if (currentCost + edge->getDistance() >= shortestPathCost)
00168             continue;
00169     }

```

```
00169         path.push_back(w);
00170         currentCost += edge->getDistance();
00171         TSPUtil(w, path, shortestPath, shortestPathCost, numOfPossiblePaths, currentCost);
00172         path.pop_back();
00173         currentCost -= edge->getDistance();
00174     }
00175
00176     return !shortestPath.empty();
00177 }
```

### 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
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
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

```
00001 //
00002 // Created by david on 5/8/23.
00003 //
00004
00005 #include <fstream>
00006 #include <sstream>
00007 #include "CPheadquarters.h"
00008 #include "MutablePriorityQueue.h"
00009 #include <chrono>
00010 #include <set>
00011 #include <cmath>
00012 #include <string>
00013
00014 using namespace std;
00015
00016 void CPheadquarters::read_edges(string path){
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;
00028         string temp;
00029         double distance;
00030
00031
00032         stringstream inputString(line1);
00033
00034         getline(inputString, origin, ',');
00035         getline(inputString, destination, ',');
00036         getline(inputString, temp, ',');
00037
00038
00039         distance = stod(temp);
00040
00041         long origin_id = std::stol(origin);
00042         graph.addVertex(origin_id);
00043
00044         long destination_id = std::stol(destination);
00045         graph.addVertex(destination_id);
00046
00047         graph.addEdge(origin_id, destination_id, distance);
00048         graph.addEdge(destination_id, origin_id, distance);
00049         cout << origin << '\n';
00050     }
00051 }
00052
00053
00054 void CPheadquarters::read_coordinates(string path){
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 '\n'
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 }
00089
00090
00091 void CPheadquarters::heuristicRec(Vertex *v, long route[], unsigned int currentIndex, double distance,
    unsigned int &nodesVisited, double &totalDistance){
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 }
00149
00150 void CPheadquarters::heuristic(long route[], unsigned int &nodesVisited, double &totalDistance, long
id) {
00151
00152     for (const auto vertex: graph.getVertexSet()) {
00153         vertex.second->setVisited(false);
00154     }
00155
00156     Vertex *actual = graph.findVertex(id);
00157
00158     double distance = 0;
00159     route[0] = actual->getId();
00160
00161     actual->setVisited(true);
00162
00163     heuristicRec(actual, route, 1, distance, nodesVisited, totalDistance);
00164 }
00165
00166 void CPheadquarters::chooseRoute() {
00167     long id;
00168     auto pathSize = graph.getNumVertex();
00169     auto vertexes = graph.getVertexSet();
00170     long path[pathSize];
00171     unsigned int nodesVisited = 0;
00172     double distance = 0;
00173     for(auto it =vertexes.begin(); it != vertexes.end(); it++){
00174         id = it->first;
00175         heuristic(path, nodesVisited, distance, id);
00176         if(nodesVisited == pathSize){
00177             long sourceId = path[pathSize-1];
00178             long destId = path[0];
00179             Vertex *sourceV = graph.findVertex(sourceId);
00180             Edge *missingEdge = sourceV->getEdge(destId);
00181             if(missingEdge!= nullptr){
00182                 distance += missingEdge->getDistance();
00183                 for(int i = 0; i < pathSize; i++){
00184                     cout << path[i] << "->";
00185                 }
00186                 cout << destId;
00187                 cout << "\nTotal distance: " << distance << endl;
00188                 break;
00189             }
00190         }
00191     }
00192 }
00193
00194 }
00195
00196
00197
00198
00199 Graph CPheadquarters::getGraph() const {
00200     return this->graph;
00201 }
00202
00203
00204
00205 void CPheadquarters::backtrack() {
00206     std::vector<Vertex*> shortestPath;
00207     double shortestPathCost = 0;
00208
00209     if (this->graph.TSP(shortestPath, shortestPathCost)) {
00210         cout << "Shortest Hamiltonian cycle: ";
00211         for (auto vertex : shortestPath)
00212             cout << vertex->getId() << " ";
00213         cout << "\nCost: " << shortestPathCost << endl;
00214     }
00215     else {
00216         cout << "The graph does not have a Hamiltonian cycle" << endl;
00217     }
00218 }
00219
00220
00221 void CPheadquarters::hamiltonianCycle() {
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 }
00234 void CPheadquarters::pathRec(Vertex* vertex){
00235     mst_preorder_path.push_back(vertex->getId());
00236     for (auto child : vertex->getChildren()) {
00237         pathRec(graph.getVertexSet()[child]);
00238     }
00239     return;
00240 }
00241
00242
00243 void CPheadquarters::triangular_Approximation_Heuristic() {
00244     std::unordered_map<long,Vertex *> vertexis = graph.getVertexSet();
00245     for (auto v: vertexis) {
00246         v.second->setVisited(false);
00247         v.second->setDist(std::numeric_limits<double>::max());
00248         v.second->eraseChildren();
00249     }
00250
00251     Vertex *root = graph.findVertex(0);
00252     root->setDist(0);
00253     MutablePriorityQueue<Vertex> q;
00254     q.insert(root);
00255     while (!q.empty()) {
00256         auto v = q.extractMin();
00257         cout<<"working on:"<<v->getId()<<'\n';
00258         v->setVisited(true);
00259         if (v->getId()!=0) {
00260             v->getPath()->getOrig()->addChildren(v->getId());
00261         }
00262         for (auto &e: v->getAdj()) {
00263             Vertex *w = e->getDest();
00264             if (!w->isVisited()) {
00265                 auto oldDist = w->getDist();
00266                 if (e->getDistance() < oldDist) {
00267                     w->setDist(e->getDistance());
00268                     w->setPath(e);
00269                     if (oldDist == std::numeric_limits<double>::max()) {
00270                         q.insert(w);
00271                     } else {
00272                         q.decreaseKey(w);
00273                     }
00274                 }
00275             }
00276         }
00277     }
00278
00279     mst_preorder_path.clear();
00280     pathRec(root);
00281
00282     double result=0;
00283
00284     for (int i = 0; i < mst_preorder_path.size()-1; i++) {
00285         result+= getDist(mst_preorder_path[i],mst_preorder_path[i+1]);
00286     }
00287     result+=getDist(mst_preorder_path[mst_preorder_path.size()-1],mst_preorder_path[0]);
00288
00289     cout<<"Result:"<<result;
00290
00291 }
00292
00293 constexpr double EarthRadius = 6371.0;
00294
00295 double CPheadquarters::getDist(int a,int b){
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 }
00301
00302
00303
00304 double CPheadquarters::degToRad (double degrees) {
00305     return degrees*M_PI/180.0;
00306 }
00307
00308 double CPheadquarters::haversineDistance(double latitude1, double longitude1, double latitude2, double
longitude2) {
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 }

```

## 4.2 CPheadquarters.h

```

00001 //
00002 // Created by david on 5/8/23.
00003 //
00004
00005 #ifndef PROJECT_2_CPHEADQUARTERS_H
00006 #define PROJECT_2_CPHEADQUARTERS_H
00007
00008
00009 #include "Graph.h"
00010
00011 using namespace std;
00012
00013 class CPheadquarters {
00014     Graph graph;
00015     vector<long> mst_preorder_path;
00016 public:
00017
00018     void read_edges(std::string path);
00019
00020     void read_coordinates(std::string path);
00021
00022
00023     Graph getGraph() const;
00024
00025     void heuristic(long path[], unsigned int &nodesVisited, double &totalDistance, long id);
00026
00027     void heuristicRec(Vertex *v, long path[], unsigned int currentIndex, double distance, unsigned int
&nodesVisited, double &totalDistance);
00028
00029     void chooseRoute();
00030
00031     void backtrack();
00032
00033     void hamiltonianCycle();
00034
00035     void triangular_Approximation_Heuristic();
00036
00037     void pathRec(Vertex *vertex);
00038
00039     double degToRad(double degrees);
00040
00041     double haversineDistance(double latitude1, double longitude1, double latitude2, double
longitude2);
00042
00043     double getDist(int a, int b);
00044 };
00045
00046 #endif //PROJECT_2_CPHEADQUARTERS_H

```

## 4.3 Graph.cpp

```

00001 #include <climits>
00002 #include <queue>
00003 #include "Graph.h"
00004 #include <algorithm>
00005 #include <unordered_set>
00006 #include <chrono>
00007
00008 int Graph::getNumVertex() const {
00009     return vertexSet.size();
00010 }
00011
00012 std::unordered_map<long, Vertex *> Graph::getVertexSet() const {
00013     return vertexSet;
00014 }
00015
00016
00017 Vertex *Graph::findVertex(const long id) const {
00018     auto it = vertexSet.find(id);
00019     if(it != vertexSet.end()) {

```

```

00020         return it->second;
00021     }
00022     return nullptr;
00023 }
00024
00025
00026 bool Graph::addVertex(const long id) {
00027     if (findVertex(id) != nullptr)
00028         return false;
00029     vertexSet[id]=(new Vertex(id));
00030     return true;
00031 }
00032
00033
00034 bool Graph::addEdge(const long sourc, const long dest, double d) {
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 }
00043
00044
00045 void deleteMatrix(int **m, int n) {
00046     if (m != nullptr) {
00047         for (int i = 0; i < n; i++)
00048             if (m[i] != nullptr)
00049                 delete[] m[i];
00050         delete[] m;
00051     }
00052 }
00053
00054 void deleteMatrix(double **m, int n) {
00055     if (m != nullptr) {
00056         for (int i = 0; i < n; i++)
00057             if (m[i] != nullptr)
00058                 delete[] m[i];
00059         delete[] m;
00060     }
00061 }
00062
00063 Graph::~Graph() {
00064     deleteMatrix(distMatrix, vertexSet.size());
00065     deleteMatrix(pathMatrix, vertexSet.size());
00066 }
00067
00068
00069 void Graph::print() const {
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 }
00084
00085
00086 void Graph::deleteVertex(long name) {
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 }
00105

```



```

00106
00107
00108 bool Graph::hasPendantVertex() {
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 }
00116
00117
00118
00119 double Graph::getPathCost(const std::vector<Vertex *> &path) {
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 }
00131
00132
00133 bool Graph::TSPUtil(Vertex *v, std::vector<Vertex *> &path, std::vector<Vertex *> &shortestPath,
00134                     double &shortestPathCost,
00135                     int &numOfPossiblePaths, double &currentCost) {
00136     if (path.size() == vertexSet.size()) {
00137         for (auto edge: v->getAdj()) {
00138             if (edge->getDest() == path[0]) {
00139                 path.push_back(path[0]);
00140                 currentCost += edge->getDistance(); // Add the cost of returning to the start vertex
00141                 if (currentCost < shortestPathCost) { // Only consider path if it's the shortest so
00142                     // Print path and its cost
00143                     std::cout << "Path: ";
00144                     for (auto vertex: path)
00145                         std::cout << vertex->getId() << " ";
00146                     std::cout << "Cost: " << currentCost << std::endl;
00147                     numOfPossiblePaths++;
00148                     shortestPath = path;
00149                     shortestPathCost = currentCost;
00150                 }
00151             }
00152             path.pop_back();
00153             currentCost -= edge->getDistance(); // Remove the cost of returning to the start
00154             return true;
00155         }
00156     }
00157     return false;
00158 }
00159
00160 for (auto edge: v->getAdj()) {
00161     Vertex *w = edge->getDest();
00162     if (std::find(path.begin(), path.end(), w) != path.end())
00163         continue;
00164
00165     // If the current path cost plus the cost of the edge is already greater than the shortest
00166     path cost, skip
00167     if (currentCost + edge->getDistance() >= shortestPathCost)
00168         continue;
00169
00170     path.push_back(w);
00171     currentCost += edge->getDistance();
00172     TSPUtil(w, path, shortestPath, shortestPathCost, numOfPossiblePaths, currentCost);
00173     path.pop_back();
00174     currentCost -= edge->getDistance();
00175 }
00176 return !shortestPath.empty();
00177 }
00178
00179
00180 bool Graph::TSP(std::vector<Vertex *> &shortestPath, double &shortestPathCost) {
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,
00211         currentCost);
00212     std::cout << "Number of calculated paths: " << numOfPossiblePaths << std::endl;
00213     // End the timer
00214     auto end_time = std::chrono::high_resolution_clock::now();
00215     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00216     std::cout << "Time taken: " << duration.count() << " ms" << std::endl;
00217     return res;
00218 }
00219
00219 double Graph::hasHamiltonianCycleUtil(Vertex *v, std::vector<Vertex *> &path, double &pathCost) {
00220     if (path.size() == vertexSet.size()) {
00221         for (auto edge: v->getAdj()) {
00222             if (edge->getDest() == path[0]) {
00223                 path.push_back(path[0]); // Closing the cycle
00224                 pathCost = getPathCost(path);
00225                 path.pop_back(); // Revert the cycle closing
00226                 return true; // found a Hamiltonian cycle
00227             }
00228         }
00229         return false;
00230     }
00231
00232     for (auto edge: v->getAdj()) {
00233         Vertex *w = edge->getDest();
00234         if (std::find(path.begin(), path.end(), w) != path.end())
00235             continue;
00236         path.push_back(w);
00237         if (hasHamiltonianCycleUtil(w, path, pathCost))
00238             return true; // propagate the success up the call stack
00239         path.pop_back();
00240     }
00241
00242     return false;
00243 }
00244
00245
00246 bool Graph::hasHamiltonianCycle(std::vector<Vertex *> &path, double &pathCost) {
00247
00248     if (this->vertexSet.empty()) {
00249         std::cout << "Graph is empty" << std::endl;
00250         return false;
00251     }
00252
00253     if (hasPendantVertex()) {
00254         std::cout << "Graph has a pendant vertex" << std::endl;
00255         return false;
00256     }
00257
00258     if(hasArticulationPoint()){
00259         std::cout << "Graph has an articulation point" << std::endl;
00260         return false;
00261     }
00262
00263     // Start the timer
00264     auto start_time = std::chrono::high_resolution_clock::now();
00265     std::cout << "Searching for a Hamiltonian Cycle..." << std::endl;
00266     std::cout << "Please stand by..." << std::endl;
00267
00268     // Measure execution time
00269     // ...
00270     path.push_back(this->vertexSet[0]);
00271     auto res = hasHamiltonianCycleUtil(this->vertexSet[0], path, pathCost);
00272
00273     // End the timer
00274     auto end_time = std::chrono::high_resolution_clock::now();

```

```

00275     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);
00276     std::cout << "Time taken: " << duration.count() << " ms" << std::endl;
00277     return res;
00278 }
00279
00280
00281
00282 bool Graph::hasArticulationPointUtil(Vertex* pCurrentVertex, int time) {
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()) {
00291         Vertex* pAdjacentVertex = edge->getDest();
00292         long adjacentVertexIdInt = pAdjacentVertex->getId();
00293         if (!visited[adjacentVertexIdInt]) {
00294             children++;
00295             parent[adjacentVertexIdInt] = currentVertexIdInt;
00296
00297             if (hasArticulationPointUtil(pAdjacentVertex, time))
00298                 return true;
00299
00300             low[currentVertexIdInt] = std::min(low[currentVertexIdInt], low[adjacentVertexIdInt]);
00301
00302             if (parent[currentVertexIdInt] == -1 && children > 1) {
00303                 ap[currentVertexIdInt] = true;
00304                 return true;
00305             }
00306
00307             if (parent[currentVertexIdInt] != -1 && low[adjacentVertexIdInt] >=
disc[currentVertexIdInt]) {
00308                 ap[currentVertexIdInt] = true;
00309                 return true;
00310             }
00311         }
00312         else if (adjacentVertexIdInt != parent[currentVertexIdInt]) {
00313             low[currentVertexIdInt] = std::min(low[currentVertexIdInt], disc[adjacentVertexIdInt]);
00314         }
00315     }
00316
00317     return false;
00318 }
00319
00320
00321 bool Graph::hasArticulationPoint() {
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 }
00338
00339

```

## 4.4 Graph.h

```

00001 //
00002 // Created by david on 5/8/23.
00003 //
00004
00005 #ifndef PROJECT_2_GRAPH_H
00006 #define PROJECT_2_GRAPH_H
00007
00008
00009 #include <iostream>
00010 #include <vector>
00011 #include <queue>
00012 #include <limits>
00013 #include <algorithm>
00014 #include <unordered_set>

```

```

00015
00016
00017 #include "VertexEdge.h"
00018
00019 class Graph {
00020 public:
00021     ~Graph();
00022
00023     Vertex *findVertex(long id) const;
00024
00025     bool addVertex(long id);
00026
00027     bool addEdge(long sourc, long dest, double d);
00028
00029
00030     [[nodiscard]] int getNumVertex() const;
00031
00032     [[nodiscard]] std::unordered_map<long,Vertex *> getVertexSet() const;
00033
00034     void print() const;
00035
00036
00037     bool TSP(std::vector<Vertex *> &shortestPath, double &shortestPathCost);
00038
00039     bool hasHamiltonianCycle(std::vector<Vertex *> &path, double &pathCost);
00040
00041 protected:
00042     std::unordered_map<long,Vertex *> vertexSet;    // vertex set
00043
00044     double **distMatrix = nullptr;    // dist matrix for Floyd-Warshall
00045     int **pathMatrix = nullptr;    // path matrix for Floyd-Warshall
00046
00047     // for Tarjan's algorithm
00048     std::vector<int> disc, low, parent;
00049     std::vector<bool> visited, ap;
00050
00051     void deleteVertex(long name);
00052
00053     double getPathCost(const std::vector<Vertex *> &path);
00054     bool TSPUtil(Vertex *v, std::vector<Vertex *> &path, std::vector<Vertex *> &shortestPath, double
&shortestPathCost,
00055         int &numOfPossiblePaths, double &currentCost);
00056     double hasHamiltonianCycleUtil(Vertex *v, std::vector<Vertex *> &path, double &pathCost);
00057
00058     bool hasPendantVertex();
00059
00060     bool hasArticulationPoint();
00061
00062     bool hasArticulationPointUtil(Vertex *pCurrentVertex, int time);
00063 };
00064
00065 void deleteMatrix(int **m, int n);
00066
00067 void deleteMatrix(double **m, int n);
00068
00069 #endif //PROJECT_2_GRAPH_H

```

## 4.5 main.cpp

```

00001 #include <iostream>
00002 #include <chrono>
00003 #include "CPheadquarters.h"
00004
00005 using namespace std;
00006
00007 int getValidInput(int lowerLimit, int upperLimit) {
00008     int n;
00009     bool validInput = false;
00010     while (!validInput) {
00011         cout << "Insert your option:\n";
00012         cin >> n;
00013
00014         if (cin.fail() || n < lowerLimit || n > upperLimit) {
00015             cin.clear();
00016             cin.ignore(numeric_limits<streamsize>::max(), '\n');
00017             cout << "Invalid input. Please enter a number between " << lowerLimit << " and " << upperLimit
<< "." << endl;
00018         } else {

```

```

00019         validInput = true;
00020     }
00021 }
00022 return n;
00023 }
00024
00025 int main() {
00026     CPheadquarters CP;
00027     string path;
00028     cout << "Insert path to file to construct a graph "
00029            "\ne.g"
00030            "\n|Toy Graphs          |: ../Toy-Graphs/Toy-Graphs/shipping.csv"
00031            "\n|Real World Graphs      |: ../Real-world Graphs/Real-world Graphs/graph1/edges.csv"
00032            "\n|Extra Fully Connected|:
    ../Extra_Fully_Connected_Graphs/Extra_Fully_Connected_Graphs/edges_25.csv )"
00033            "\n:";
00034     getline(cin, path);
00035     CP.read_edges(path);
00036     cout<<"If necessary, insert path to file that contains latitude and longitude"
00037            "\n(e.g ../Real-world Graphs/Real-world Graphs/graph1/nodes.csv)"
00038            "\n Otherwise, press enter."
00039            "\n:";
00040     getline(cin, path);
00041     cout<<endl;
00042     if (!path.empty()) {
00043         CP.read_coordinates(path);
00044     }
00045     //CP.getGraph().print();
00046     int n;
00047     cout << "\n----- An Analysis Tool for Railway Network Management ----- \n" << endl;
00048     do {
00049         cout << "\n1 - T2.1 Backtracking Algorithm\n";
00050         cout << "2 - T2.2 Triangular Approximation Heuristic\n";
00051         cout << "3 - T2.3 other heuristic algorithms\n";
00052         cout << "8 - Exit\n";
00053
00054         n = getValidInput(1, 8);
00055
00056         switch (n) {
00057             case 1: {
00058                 cout << "1 - TSP using Backtracking algorithm (for small graphs)\n";
00059                 cout << "2 - Just find ANY the Hamiltonian Cycle (for big graphs)\n";
00060                 int backtrack_choice;
00061                 backtrack_choice = getValidInput(1, 2);
00062                 switch(backtrack_choice){
00063                     case 1:
00064                         CP.backtrack();
00065                         break;
00066
00067                     case 2:
00068                         CP.hamiltonianCycle();
00069                         break;
00070
00071                     default: {
00072                         cerr << "Error: Invalid option selected." << endl;
00073                         break;
00074                     }
00075                 }
00076                 break;
00077             }
00078
00079             case 2: {
00080                 auto start_time = std::chrono::high_resolution_clock::now();
00081
00082                 CP.triangular_Approximation_Heuristic();
00083
00084                 auto end_time = std::chrono::high_resolution_clock::now();
00085                 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time -
00086 start_time);
00087
00088                 std::cout << "Time taken: " << duration.count() << " ms" << std::endl;
00089
00090                 break;
00091             }
00092
00093             case 3: {
00094                 auto start_time = std::chrono::high_resolution_clock::now();
00095
00096                 CP.chooseRoute();
00097
00098                 // Code block to measure goes here
00099                 // End the timer
00100                 auto end_time = std::chrono::high_resolution_clock::now();
00101
00102
00103

```

```

00104         // Compute the duration
00105         auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time -
start_time);
00106         // Print the duration
00107         std::cout << "Time taken: " << duration.count() << " ms" << std::endl;
00108
00109         break;
00110     }
00111 }
00112
00113     case 8: {
00114         cout << "Exiting program..." << endl;
00115         break;
00116     }
00117
00118     default: {
00119         cerr << "Error: Invalid option selected." << endl;
00120         break;
00121     }
00122 }
00123 } while (n != 8);
00124
00125 return 0;
00126 }

```

## 4.6 MutablePriorityQueue.h

```

00001 /*
00002  * MutablePriorityQueue.h
00003  * A simple implementation of mutable priority queues, required by Dijkstra algorithm.
00004  *
00005  * Created on: 17/03/2018
00006  * Author: João Pascoal Faria
00007  */
00008
00009 #ifndef DA_TP_CLASSES_MUTABLEPRIORITYQUEUE
00010 #define DA_TP_CLASSES_MUTABLEPRIORITYQUEUE
00011
00012 #include <vector>
00013
00014
00015
00020 template <class T>
00021 class MutablePriorityQueue {
00022     std::vector<T *> H;
00023     void heapifyUp(unsigned i);
00024     void heapifyDown(unsigned i);
00025     inline void set(unsigned i, T * x);
00026 public:
00027     MutablePriorityQueue();
00028     void insert(T * x);
00029     T * extractMin();
00030     void decreaseKey(T * x);
00031     bool empty();
00032 };
00033
00034 // Index calculations
00035 #define parent(i) ((i) / 2)
00036 #define leftChild(i) ((i) * 2)
00037
00038 template <class T>
00039 MutablePriorityQueue<T>::MutablePriorityQueue() {
00040     H.push_back(nullptr);
00041     // indices will be used starting in 1
00042     // to facilitate parent/child calculations
00043 }
00044
00045 template <class T>
00046 bool MutablePriorityQueue<T>::empty() {
00047     return H.size() == 1;
00048 }
00049
00050 template <class T>
00051 T* MutablePriorityQueue<T>::extractMin() {
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 }
00059
00060 template <class T>

```

```

00061 void MutablePriorityQueue<T>::insert(T *x) {
00062     H.push_back(x);
00063     heapifyUp(H.size()-1);
00064 }
00065
00066 template <class T>
00067 void MutablePriorityQueue<T>::decreaseKey(T *x) {
00068     heapifyUp(x->queueIndex);
00069 }
00070
00071 template <class T>
00072 void MutablePriorityQueue<T>::heapifyUp(unsigned i) {
00073     auto x = H[i];
00074     while (i > 1 && *x < *H[parent(i)]) {
00075         set(i, H[parent(i)]);
00076         i = parent(i);
00077     }
00078     set(i, x);
00079 }
00080
00081 template <class T>
00082 void MutablePriorityQueue<T>::heapifyDown(unsigned i) {
00083     auto x = H[i];
00084     while (true) {
00085         unsigned k = leftChild(i);
00086         if (k >= H.size())
00087             break;
00088         if (k+1 < H.size() && *H[k+1] < *H[k])
00089             ++k; // right child of i
00090         if ( ! (*H[k] < *x) )
00091             break;
00092         set(i, H[k]);
00093         i = k;
00094     }
00095     set(i, x);
00096 }
00097
00098 template <class T>
00099 void MutablePriorityQueue<T>::set(unsigned i, T * x) {
00100     H[i] = x;
00101     x->queueIndex = i;
00102 }
00103
00104 #endif /* DA_TP_CLASSES_MUTABLEPRIORITYQUEUE */

```

## 4.7 VertexEdge.cpp

```

00001 //
00002 // Created by david on 5/8/23.
00003 //
00004
00005 #include "VertexEdge.h"
00006
00007 /***** Vertex *****/
00008
00009 Vertex::Vertex(long id) : id(id) {}
00010
00011 /*
00012  * Auxiliary function to add an outgoing edge to a vertex (this),
00013  * with a given destination vertex (d) and edge weight (w).
00014  */
00015 Edge *Vertex::addEdge(Vertex *d, double dist) {
00016     auto newEdge = new Edge(this, d, dist);
00017     adj.push_back(newEdge);
00018     d->incoming.push_back(newEdge);
00019     return newEdge;
00020 }
00021
00022 /*
00023  * Auxiliary function to remove an outgoing edge (with a given destination (d))
00024  * from a vertex (this).
00025  * Returns true if successful, and false if such edge does not exist.
00026  */
00027 bool Vertex::removeEdge(long destID) {
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
00046     } else {
00047         it++;
00048     }
00049 }
00050 return removedEdge;
00051 }
00052
00053 Edge *Vertex::getEdge(long destID){
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
00062         } else {
00063             it++;
00064         }
00065     }
00066     return nullptr;
00067 }
00068
00069 bool Vertex::operator<(Vertex &vertex) const {
00070     return this->dist < vertex.dist;
00071 }
00072
00073 long Vertex::getId() const {
00074     return this->id;
00075 }
00076
00077 std::vector<Edge *> Vertex::getAdj() const {
00078     return this->adj;
00079 }
00080
00081 bool Vertex::isVisited() const {
00082     return this->visited;
00083 }
00084
00085 double Vertex::getDist() const {
00086     return this->dist;
00087 }
00088
00089 Edge *Vertex::getPath() const {
00090     return this->path;
00091 }
00092
00093 std::vector<Edge *> Vertex::getIncoming() const {
00094     return this->incoming;
00095 }
00096
00097 void Vertex::setId(int id) {
00098     this->id = id;
00099 }
00100
00101 void Vertex::setVisited(bool visited) {
00102     this->visited = visited;
00103 }
00104
00105 void Vertex::setDist(double dist) {
00106     this->dist = dist;
00107 }
00108
00109 void Vertex::setPath(Edge *path) {
00110     this->path = path;
00111 }
00112
00113 void Vertex::print() const {
00114     std::cout << "Vertex: " << id << std::endl;
00115     std::cout << "Adjacent to: ";
00116     for (const Edge *e: adj) {
00117         std::cout << e->getDest()->getId() << " ";
00118     }
00119     std::cout << std::endl;
00120     std::cout << "Visited: " << visited << std::endl;

```



```

00122     std::cout << "Distance: " << dist << std::endl;
00123     std::cout << "Path: " << path << std::endl;
00124 }
00125
00126 void Vertex::eraseChildren() {
00127     children.clear();
00128 }
00129
00130 void Vertex::addChildren(long s) {
00131     children.push_back(s);
00132 }
00133
00134 std::vector<long> Vertex::getChildren() {
00135     return children;
00136 }
00137
00138
00139 /***** Edge *****/
00140
00141 Edge::Edge(Vertex *orig, Vertex *dest, double d) : orig(orig), dest(dest), distance(d) {}
00142
00143 Vertex *Edge::getDest() const {
00144     return this->dest;
00145 }
00146
00147 double Edge::getDistance() const {
00148     return this->distance;
00149 }
00150
00151 Vertex *Edge::getOrig() const {
00152     return this->orig;
00153 }
00154
00155 double Vertex::getLatitude() {
00156     return latitude;
00157 }
00158
00159 double Vertex::getLongitude() {
00160     return longitude;
00161 }
00162
00163 void Vertex::setLatitude(double latitude_) {
00164     latitude=latitude_;
00165 }
00166
00167 void Vertex::setLongitude(double longitude_) {
00168     longitude=longitude_;
00169 }
00170
00171
00172

```

## 4.8 VertexEdge.h

```

00001 //
00002 // Created by david on 5/8/23.
00003 //
00004
00005 #ifndef PROJECT_2_VERTEXEDGE_H
00006 #define PROJECT_2_VERTEXEDGE_H
00007
00008
00009 #include <iostream>
00010 #include <vector>
00011 #include <queue>
00012 #include <limits>
00013 #include <algorithm>
00014
00015
00016 class Edge;
00017
00018 #define INF std::numeric_limits<double>::max()
00019
00020 /***** Vertex *****/
00021
00022 class Vertex {
00023 public:
00024     Vertex(long id);
00025
00026     bool operator<(Vertex &vertex) const; // // required by MutablePriorityQueue
00027
00028     long getId() const;
00029

```

```

00030     std::vector<Edge *> getAdj() const;
00031
00032     bool isVisited() const;
00033
00034     double getDist() const;
00035
00036     Edge *getPath() const;
00037
00038     std::vector<Edge *> getIncoming() const;
00039
00040     void setId(int info);
00041
00042     void setVisited(bool visited);
00043
00044     void setDist(double dist);
00045
00046     void setPath(Edge *path);
00047
00048     Edge *addEdge(Vertex *dest, double w);
00049
00050     bool removeEdge(long destID);
00051
00052     Edge *getEdge(long destID);
00053
00054     void eraseChildren();
00055
00056     void addChildren(long s);
00057
00058     std::vector<long> getChildren();
00059
00060     double getLatitude();
00061
00062     double getLongitude();
00063
00064     void setLatitude(double latitude);
00065
00066     void setLongitude(double longitude);
00067
00068     int queueIndex = 0;
00069 protected:
00070     long id; // identifier
00071     std::vector<Edge *> adj; // outgoing edges
00072     std::vector<long> children;
00073
00074     // auxiliary fields
00075     bool visited = false; // used by DFS, BFS, Prim ...
00076     double dist = 0;
00077     double longitude=0;
00078     double latitude=0;
00079
00080
00081     Edge *path = nullptr;
00082
00083     std::vector<Edge *> incoming; // incoming edges
00084
00085     // required by MutablePriorityQueue and UFDS
00086     void print() const;
00087
00088 };
00089
00090
00091 /***** Edge *****/
00092
00093 class Edge {
00094 public:
00095     Edge(Vertex *orig, Vertex *dest, double d);
00096
00097     Vertex *getDest() const;
00098
00099     double getDistance() const;
00100
00101     Vertex *getOrig() const;
00102
00103 protected:
00104     Vertex *dest; // destination vertex
00105     double distance; // edge weight, can also be used for capacity
00106
00107     // used for bidirectional edges
00108     Vertex *orig;
00109
00110 };
00111
00112
00113
00114 #endif //PROJECT_2_VERTEXEDGE_H

```

# Index

- ~Graph
  - Graph, [17](#)
- addChildren
  - Vertex, [30](#)
- addEdge
  - Graph, [17](#)
  - Vertex, [30](#)
- addVertex
  - Graph, [17](#)
- adj
  - Vertex, [34](#)
- ap
  - Graph, [26](#)
- backtrack
  - CPheadquarters, [6](#)
- children
  - Vertex, [34](#)
- chooseRoute
  - CPheadquarters, [6](#)
- CPheadquarters, [5](#)
  - backtrack, [6](#)
  - chooseRoute, [6](#)
  - degToRad, [6](#)
  - getDist, [8](#)
  - getGraph, [8](#)
  - hamiltonianCycle, [8](#)
  - haversineDistance, [9](#)
  - heuristic, [9](#)
  - heuristicRec, [10](#)
  - pathRec, [11](#)
  - read\_coordinates, [12](#)
  - read\_edges, [12](#)
  - triangular\_Approximation\_Heuristic, [13](#)
- decreaseKey
  - MutablePriorityQueue< T >, [28](#)
- degToRad
  - CPheadquarters, [6](#)
- deleteVertex
  - Graph, [18](#)
- dest
  - Edge, [15](#)
- disc
  - Graph, [26](#)
- dist
  - Vertex, [34](#)
- distance
  - Edge, [15](#)
- distMatrix
  - Graph, [26](#)
- Edge, [14](#)
  - dest, [15](#)
  - distance, [15](#)
  - Edge, [14](#)
  - getDest, [15](#)
  - getDistance, [15](#)
  - getOrig, [15](#)
  - orig, [15](#)
- empty
  - MutablePriorityQueue< T >, [28](#)
- eraseChildren
  - Vertex, [30](#)
- extractMin
  - MutablePriorityQueue< T >, [28](#)
- findVertex
  - Graph, [18](#)
- getAdj
  - Vertex, [30](#)
- getChildren
  - Vertex, [30](#)
- getDest
  - Edge, [15](#)
- getDist
  - CPheadquarters, [8](#)
  - Vertex, [31](#)
- getDistance
  - Edge, [15](#)
- getEdge
  - Vertex, [31](#)
- getGraph
  - CPheadquarters, [8](#)
- getId
  - Vertex, [31](#)
- getIncoming
  - Vertex, [31](#)
- getLatitude
  - Vertex, [31](#)
- getLongitude
  - Vertex, [32](#)
- getNumVertex
  - Graph, [19](#)
- getOrig
  - Edge, [15](#)
- getPath

- Vertex, [32](#)
- getPathCost
  - Graph, [19](#)
- getVertexSet
  - Graph, [19](#)
- Graph, [16](#)
  - ~Graph, [17](#)
  - addEdge, [17](#)
  - addVertex, [17](#)
  - ap, [26](#)
  - deleteVertex, [18](#)
  - disc, [26](#)
  - distMatrix, [26](#)
  - findVertex, [18](#)
  - getNumVertex, [19](#)
  - getPathCost, [19](#)
  - getVertexSet, [19](#)
  - hasArticulationPoint, [19](#)
  - hasArticulationPointUtil, [20](#)
  - hasHamiltonianCycle, [21](#)
  - hasHamiltonianCycleUtil, [22](#)
  - hasPendantVertex, [23](#)
  - low, [26](#)
  - parent, [26](#)
  - pathMatrix, [26](#)
  - print, [23](#)
  - TSP, [23](#)
  - TSPUtil, [25](#)
  - vertexSet, [26](#)
  - visited, [27](#)
- hamiltonianCycle
  - CPheadquarters, [8](#)
- hasArticulationPoint
  - Graph, [19](#)
- hasArticulationPointUtil
  - Graph, [20](#)
- hasHamiltonianCycle
  - Graph, [21](#)
- hasHamiltonianCycleUtil
  - Graph, [22](#)
- hasPendantVertex
  - Graph, [23](#)
- haversineDistance
  - CPheadquarters, [9](#)
- heuristic
  - CPheadquarters, [9](#)
- heuristicRec
  - CPheadquarters, [10](#)
- id
  - Vertex, [34](#)
- incoming
  - Vertex, [35](#)
- insert
  - MutablePriorityQueue< T >, [28](#)
- isVisited
  - Vertex, [32](#)
- latitude
  - Vertex, [35](#)
- longitude
  - Vertex, [35](#)
- low
  - Graph, [26](#)
- MutablePriorityQueue
  - MutablePriorityQueue< T >, [27](#)
- MutablePriorityQueue< T >, [27](#)
  - decreaseKey, [28](#)
  - empty, [28](#)
  - extractMin, [28](#)
  - insert, [28](#)
  - MutablePriorityQueue, [27](#)
- operator<
  - Vertex, [32](#)
- orig
  - Edge, [15](#)
- parent
  - Graph, [26](#)
- path
  - Vertex, [35](#)
- pathMatrix
  - Graph, [26](#)
- pathRec
  - CPheadquarters, [11](#)
- print
  - Graph, [23](#)
  - Vertex, [32](#)
- queueIndex
  - Vertex, [35](#)
- read\_coordinates
  - CPheadquarters, [12](#)
- read\_edges
  - CPheadquarters, [12](#)
- removeEdge
  - Vertex, [32](#)
- setDist
  - Vertex, [33](#)
- setId
  - Vertex, [33](#)
- setLatitude
  - Vertex, [33](#)
- setLongitude
  - Vertex, [33](#)
- setPath
  - Vertex, [34](#)
- setVisited
  - Vertex, [34](#)
- triangular\_Approximation\_Heuristic
  - CPheadquarters, [13](#)
- TSP
  - Graph, [23](#)

## TSPUtil

- Graph, [25](#)

Vertex, [29](#)

- addChildren, [30](#)
- addEdge, [30](#)
- adj, [34](#)
- children, [34](#)
- dist, [34](#)
- eraseChildren, [30](#)
- getAdj, [30](#)
- getChildren, [30](#)
- getDist, [31](#)
- getEdge, [31](#)
- getId, [31](#)
- getIncoming, [31](#)
- getLatitude, [31](#)
- getLongitude, [32](#)
- getPath, [32](#)
- id, [34](#)
- incoming, [35](#)
- isVisited, [32](#)
- latitude, [35](#)
- longitude, [35](#)
- operator<, [32](#)
- path, [35](#)
- print, [32](#)
- queueIndex, [35](#)
- removeEdge, [32](#)
- setDist, [33](#)
- setId, [33](#)
- setLatitude, [33](#)
- setLongitude, [33](#)
- setPath, [34](#)
- setVisited, [34](#)
- Vertex, [30](#)
- visited, [35](#)

## vertexSet

- Graph, [26](#)

## visited

- Graph, [27](#)
- Vertex, [35](#)