Bus Navigation Program

Generated by Doxygen 1.9.2

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

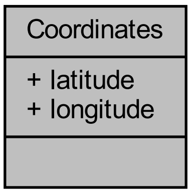
Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 Coordinates Struct Reference

Collaboration diagram for Coordinates:



Public Attributes

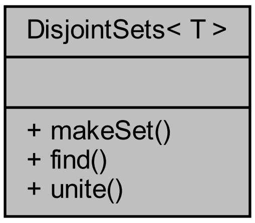
- double **latitude**
- double **longitude**

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

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Node.h

3.2 DisjointSets< T > Class Template Reference

Collaboration diagram for DisjointSets< T >:



Public Member Functions

- void **makeSet** (const T &x)
- T **find** (const T &x)
- void **unite** (const T &x, const T &y)

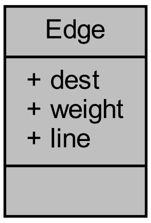
The documentation for this class was generated from the following file:

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/DisjointSets.h

3.3 Edge Struct Reference

`#include <Node.h>`

Collaboration diagram for Edge:



Public Attributes

- int **dest**
- double **weight**
- string **line**

3.3.1 Detailed Description

Lines that connect the bus stops

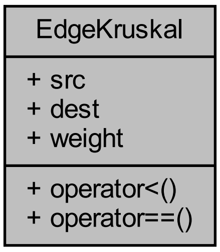
The documentation for this struct was generated from the following file:

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Node.h

3.4 EdgeKruskal Struct Reference

```
#include <Node.h>
```

Collaboration diagram for EdgeKruskal:



Public Member Functions

- bool **operator<** (const EdgeKruskal &other) const
- bool **operator==** (const EdgeKruskal &other) const

Public Attributes

- int **src**
- int **dest**
- double **weight**

3.4.1 Detailed Description

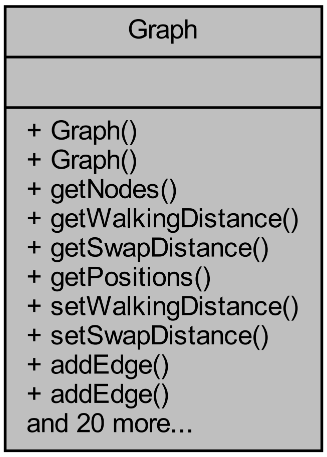
Struct only used for calculating Kruskal MST value

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

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Node.h

3.5 Graph Class Reference

Collaboration diagram for Graph:



Public Member Functions

- **Graph** (int nodes=0, bool dir=true)
- **Graph** (vector< Node > nodes, unordered_map< string, int > positions, bool dir=true)
- vector< Node > & **getNodes** ()
- double **getWalkingDistance** ()
- double **getSwapDistance** ()
- unordered_map< string, int > **getPositions** ()
- void **setWalkingDistance** (double walkingDist)
- void setSwapDistance (double swapDist)
- void addEdge (string src, string dest, string line)
- void addEdge (int src, int dest, string line)
- void addEdge (int src, int dest, double weight=1.0)
- double calculateDistance (Coordinates c1, Coordinates c2)
- double calculateDistance (int src, int dest)
- int calculateChangesOfLine (map< string, pair< double, int > > lines, int numPrevChangesOfLines, string newLine)

- double calculateWeightDijkstraLine (int min, double edgeWeight, string newLine)
- long double prim (int r)
- long double kruskal ()
- double bfs (string src, string dest)
- double bfs (int a, int b)
- double dijkstra (string src, string dest)
- double dijkstra (int a, int b)
- double dijkstraLine (string src, string dest)
- double dijkstraLine (int a, int b)
- stack$<$ int $>$ getPathFromGraph (string src, string dest)
- stack$<$ int $>$ getPathFromGraph (int a, int b)
- void printPath (stack$<$ int $>$ path)
- void printPathLinesAlgorithm (stack$<$ int $>$ path)
- void printNodes ()
- void insertTemporaryNode (Coordinates c, bool startType)
- void removeTemporaryNodes ()

3.5.1 Member Function Documentation

3.5.1.1 addEdge() [1/3]

```
void Graph::addEdge (
             int src,
             int dest,
             double weight = 1.0 )
```

Adds an edge to the list of outgoing edges to adjacent nodes of the origin node. It does the same for the destiny node, if the graph is undirected.

Parameters

src	The origin's position
dest	The destiny's position
weight	The predetermined weight

3.5.1.2 addEdge() [2/3]

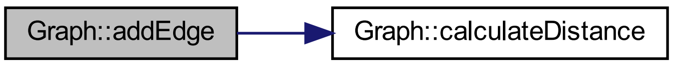
```
void Graph::addEdge (
             int src,
             int dest,
             string line )
```

Adds an edge to the list of outgoing edges to adjacent nodes of the origin node. It does the same for the destiny node, if the graph is undirected. It also calculates the weight of the edge (distance) based on the origin and destiny and includes the line that covers this edge.

Parameters

src	The origin's position
dest	The destiny's position
line	The line's code

Here is the call graph for this function:



3.5.1.3 addEdge() `[3/3]`

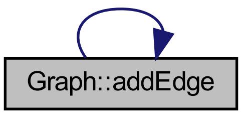
```
void Graph::addEdge (
             string src,
             string dest,
             string line )
```

Gets the position of an origin and destiny from their station names, and uses them on the other addEdge method.

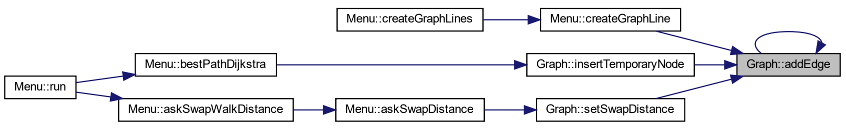
Parameters

src	The origin code
dest	The destiny code
line	The line's code

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.1.4 bfs() [1/2]

```
double Graph::bfs (
              int a,
              int b )
```

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

Parameters

Beginning	node a
Ending	node b

Returns

value of the distance of the path with the least amount of stops

3.5.1.5 bfs() [2/2]

```
double Graph::bfs (
              string src,
              string dest )
```

Determines the positions of an origin and destiny to use on the other bfs method.

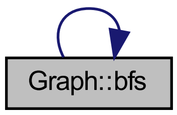
Parameters

src	The origin's code
dest	The destiny's code

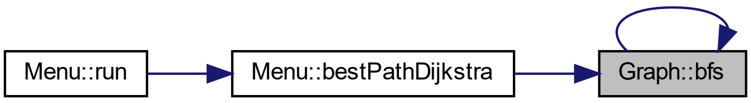
Returns

value of the distance of the path with the least amount of stops

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.1.6 calculateChangesOfLine()

```
int Graph::calculateChangesOfLine (
            map< string, pair< double, int > > lines,
            int numPrevChangesOfLines,
            string newLine )
```

Calculates the number of line changes in the new node.

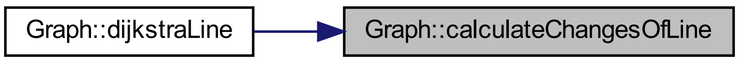
Parameters

lines	The bus lines that reached the previous node
numPrevChangesOfLines	The number of line changes of the previous node
newLine	The line of the edge that we are currently analysing

Returns

number of line changes

Here is the caller graph for this function:



3.5.1.7 calculateDistance() `[1/2]`

```
double Graph::calculateDistance (
            Coordinates c1,
            Coordinates c2 )
```

Uses the mathematical formula to calculate the distance between two points

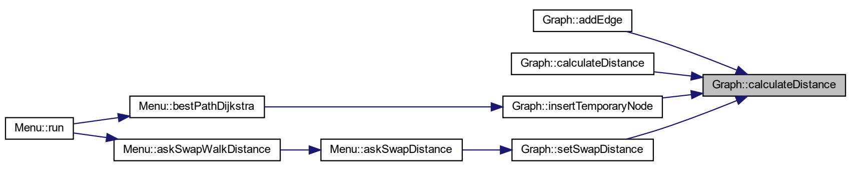
Parameters

c1	The origin's coordinates
c2	The destiny's coordinates

Returns

The distance between two nodes

Here is the caller graph for this function:



3.5.1.8 calculateDistance() `[2/2]`

```
double Graph::calculateDistance (
             int src,
             int dest )
```

Determines the coordinates of a the origin and destiny and uses them on the other method with this name

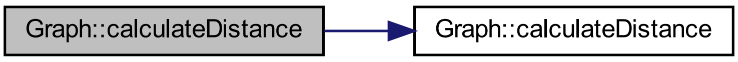
Parameters

src	The origin's position
dest	The destiny's position

Returns

The distance between two nodes

Here is the call graph for this function:



3.5.1.9 calculateWeightDijkstraLine()

```
double Graph::calculateWeightDijkstraLine (
             int min,
             double edgeWeight,
             string newLine )
```

Calculates the distance of the new node according to the line of the edge.

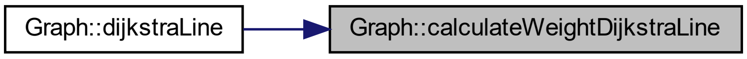
Parameters

min	previousNode
edgeWeight	
newLine	

Returns

distance

Here is the caller graph for this function:



3.5.1.10 dijkstra() `[1/2]`

```
double Graph::dijkstra (
            int a,
            int b )
```

Time Complexity: O(|E| long(|V|))

Parameters

Beginning	node a
Ending	node b

Returns

value of the distance of the shortest path

3.5.1.11 dijkstra() `[2/2]`

```
double Graph::dijkstra (
            string src,
            string dest )
```

Determines the positions of an origin and destiny to use on the other dijkstra method.

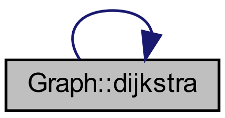
Parameters

src	The origin's code
dest	The destiny's code

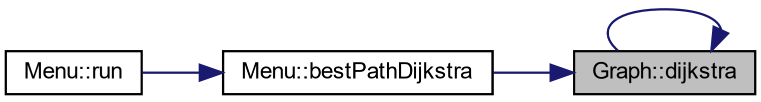
Returns

value of the distance of the shortest path

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.1.12 dijkstraLine() `[1/2]`

```
double Graph::dijkstraLine (
              int a,
              int b )
```

Time Complexity: O(|E| long(|V|)) (overlapping lines make it hard to tell time complexity)

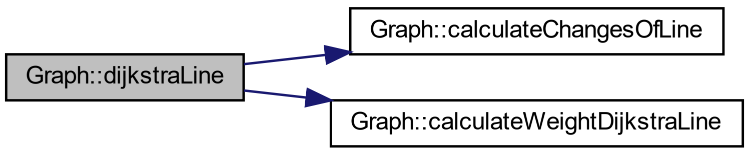
Parameters

Beginning	node a
Ending	node b

Returns

value of the distance of the path with the fewest changes in lines

Here is the call graph for this function:


Graph::dijkstraLine
Graph::calculateChangesOfLine
Graph::calculateWeightDijkstraLine

3.5.1.13 dijkstraLine() `[2/2]`

```
double Graph::dijkstraLine (
            string src,
            string dest )
```

Determines the positions of an origin and destiny to use on the other dijkstraLine method.

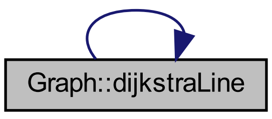
Parameters

src	The origin's code
dest	The destiny's code

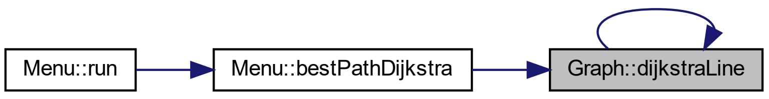
Returns

value of the distance of the path with the fewest changes in lines

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.1.14 getPathFromGraph() [1/2]

```
stack< int > Graph::getPathFromGraph (
            int a,
            int b )
```

Stores the path in a stack and then returns it.

Parameters

a	The origin's position
b	The destinty's position

Returns

The positions of every node on the way between two nodes

3.5.1.15 getPathFromGraph() [2/2]

```
stack< int > Graph::getPathFromGraph (
            string src,
            string dest )
```

Determines the positions of an origin and destiny to use them in the other method of this name.

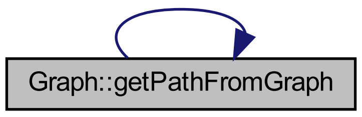
Parameters

src	The origin's code
dest	The destiny's code

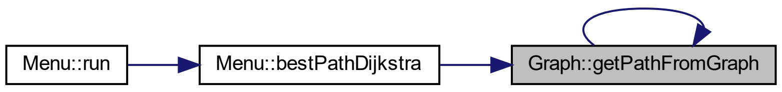
Returns

The positions of every node on the way between two nodes

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.1.16 insertTemporaryNode()

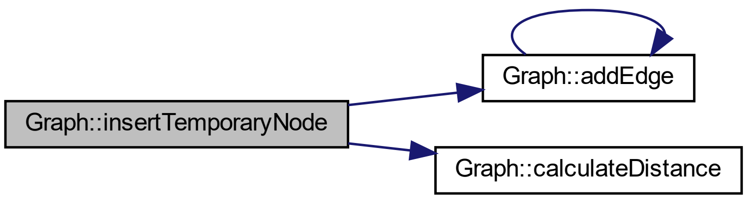
```
void Graph::insertTemporaryNode (
             Coordinates c,
             bool startType )
```

Inserts a temporary node that are at the coordinates.

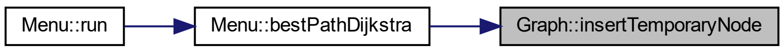
Parameters

c	Coordinates of origin or destiny
startType	True if the coordinates refer to the origin, false if they refer to the destiny

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.1.17 kruskal()

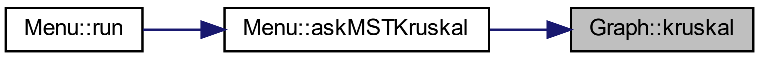
```
long double Graph::kruskal ( )
```

Time Complexity: |E| log(|E|)

Returns

The value of the minimum spanning tree

Here is the caller graph for this function:



3.5.1.18 prim()

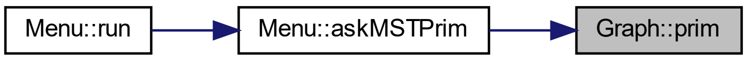
```
long double Graph::prim (
             int r )
```

Time Complexity: |E| log(|V|)

Returns

The value of the minimum spanning tree

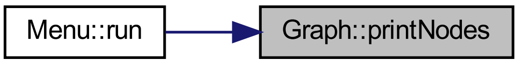
Here is the caller graph for this function:



3.5.1.19 printNodes()

```
void Graph::printNodes ( )
```

Prints the code, position, and local of every node. It also prints the code of every adjacent node and its distance from it, and line that links them. Here is the caller graph for this function:



3.5.1.20 printPath()

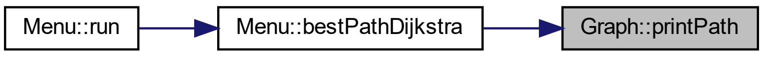
```
void Graph::printPath (
             stack< int > path )
```

Prints every edge in the path between two nodes. It also shows the line and distance of each edge.

Parameters

path	The positions of every node on the way between two nodes

Here is the caller graph for this function:



3.5.1.21 printPathLinesAlgorithm()

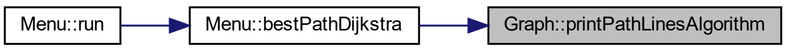
```
void Graph::printPathLinesAlgorithm (
             stack< int > path )
```

Prints every edge in the path between two nodes. It also shows the line and distance of each edge. This works for the dijkstraLine algorithm

Parameters

path	The positions of every node on the way between two nodes

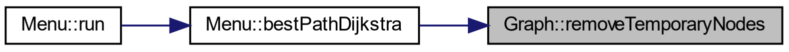
Here is the caller graph for this function:



3.5.1.22 removeTemporaryNodes()

```
void Graph::removeTemporaryNodes ( )
```

Removes every temporary starting and ending nodes. Here is the caller graph for this function:



3.5.1.23 setSwapDistance()

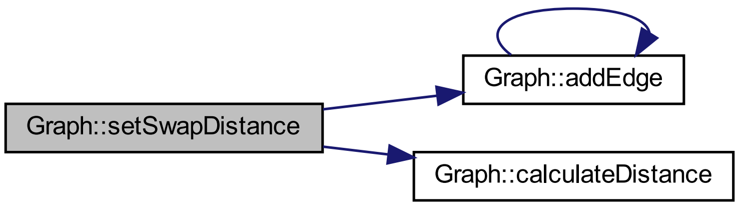
```
void Graph::setSwapDistance (
             double swapDist )
```

Sets a new value for swapDistance. But it also cleans every walking edge that existed before and creates new edges according to this new value.

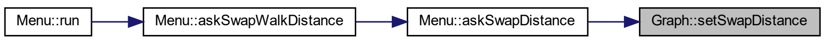
Parameters

swapDist	The distance the user is willing to walk to swap buses

Here is the call graph for this function:



Here is the caller graph for this function:

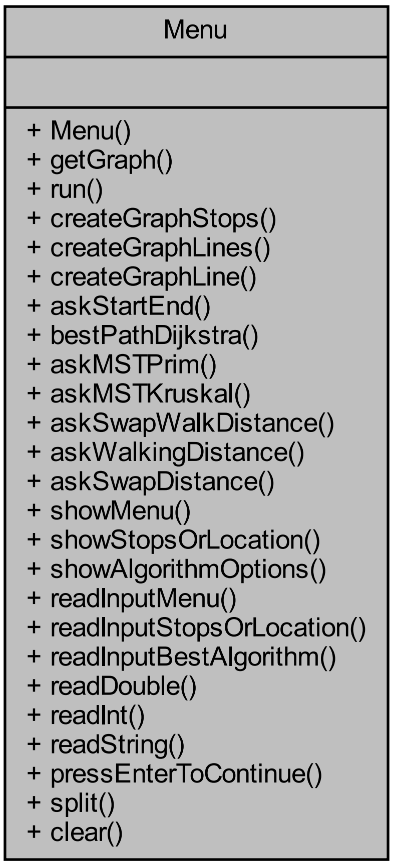


The documentation for this class was generated from the following files:

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Graph.h
- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Graph.cpp

3.6 Menu Class Reference

Collaboration diagram for Menu:



Public Member Functions

- **Menu** (string directory)
- Graph & **getGraph** ()
- void run ()
- void createGraphStops ()
- void createGraphLines ()
- void createGraphLine (string line)
- int askStartEnd (string &stopBegin, string &stopEnd, Coordinates &cBegin, Coordinates &cEnd)
- void bestPathDijkstra ()
- void askMSTPrim ()
- void askMSTKruskal ()
- void askSwapWalkDistance ()
- void askWalkingDistance ()
- void askSwapDistance ()

Static Public Member Functions

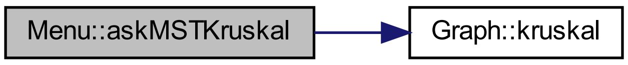
- static void showMenu ()
- static void showStopsOrLocation ()
- static void showAlgorithmOptions ()
- static int readInputMenu ()
- static int readInputStopsOrLocation ()
- static int readInputBestAlgorithm ()
- static double readDouble ()
- static int readInt ()
- static string readString ()
- static void pressEnterToContinue ()
- static vector< string > split (string line, string delimeter)
- static void clear ()

3.6.1 Member Function Documentation

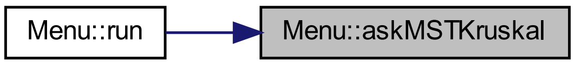
3.6.1.1 askMSTKruskal()

```
void Menu::askMSTKruskal ( )
```

Shows the user the resultant MST from Kruskal's Algorithm. Here is the call graph for this function:



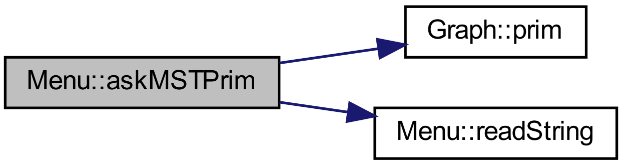
Here is the caller graph for this function:



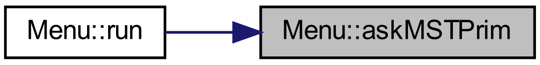
3.6.1.2 askMSTPrim()

```
void Menu::askMSTPrim ( )
```

Asks the user the origin stop to start with Prim' Algorithm. It also shows the resultant MST from that algorithm. Here is the call graph for this function:



Here is the caller graph for this function:



3.6.1.3 askStartEnd()

```
int Menu::askStartEnd (
             string & stopBegin,
             string & stopEnd,
             Coordinates & cBegin,
             Coordinates & cEnd )
```

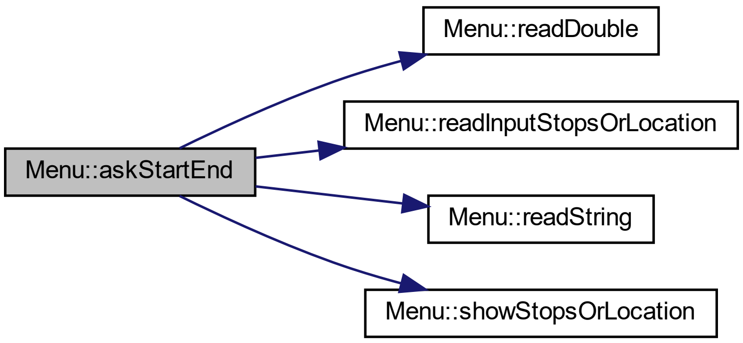
Asks the user the codes or coordinates of their wanted origin and destination. If the user inserts coordinates that don't belong to any stop and refuses to walk, the system warns them about it.

Parameters

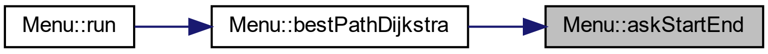
stopBegin	The origin's code
stopEnd	The destiny's code
cBegin	The origin's coordinates
cEnd	The destiny's coordinates

Returns

Here is the call graph for this function:



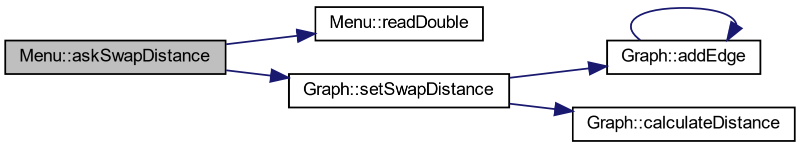
Here is the caller graph for this function:



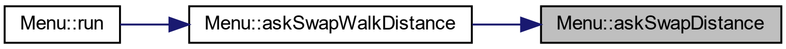
3.6.1.4 askSwapDistance()

```
void Menu::askSwapDistance ( )
```

Asks the user how much they are willing to walk to swap buses in the middle of their journey. Here is the call graph for this function:



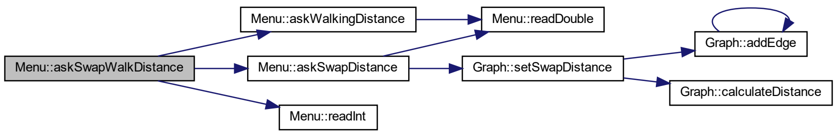
Here is the caller graph for this function:



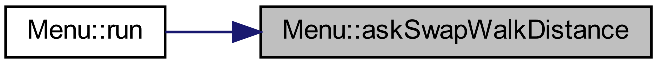
3.6.1.5 askSwapWalkDistance()

```
void Menu::askSwapWalkDistance ( )
```

Ask the user which of the two types of walking distances they want to set. Here is the call graph for this function:



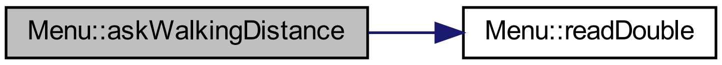
Here is the caller graph for this function:



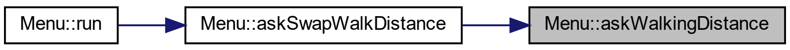
3.6.1.6 askWalkingDistance()

```
void Menu::askWalkingDistance ( )
```

Asks the user how much they are willing to walk to either their first stop or from their last stop to destiny if they give coordinates instead of the stops. Here is the call graph for this function:



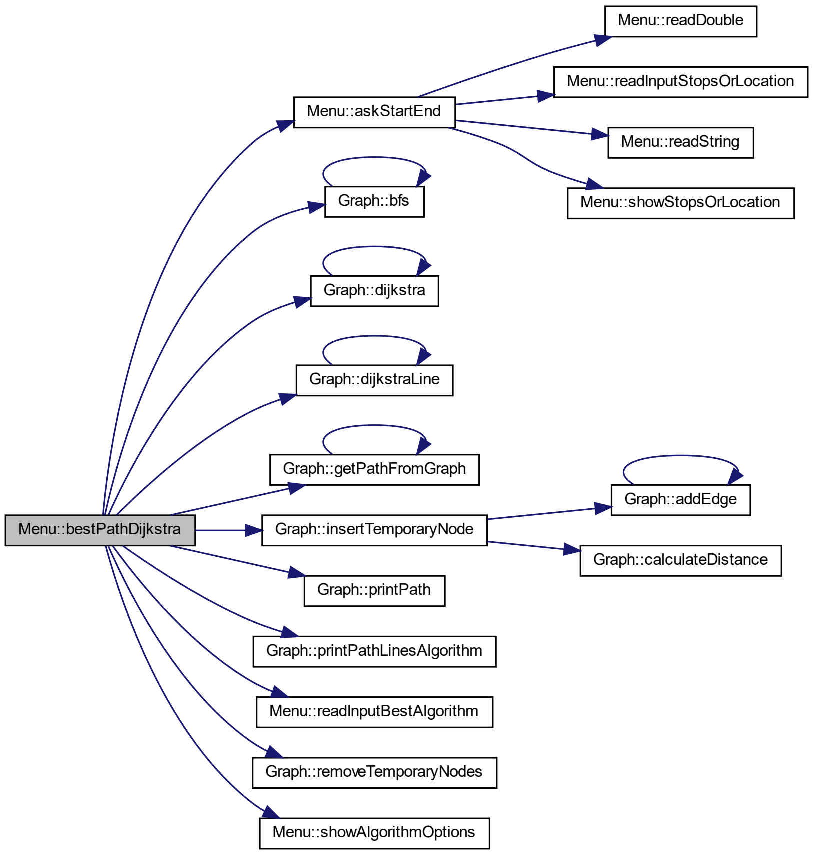
Here is the caller graph for this function:



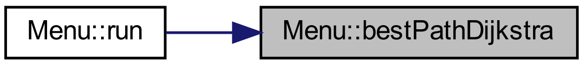
3.6.1.7 bestPathDijkstra()

```
void Menu::bestPathDijkstra ( )
```

Method that deals with all the calculations of paths. Here is the call graph for this function:


Menu::readDouble
Menu::readInputStopsOrLocation
Menu::askStartEnd
Menu::readString
Menu::showStopsOrLocation
Graph::bfs
Graph::dijkstra
Graph::dijkstraLine
Graph::getPathFromGraph
Graph::addEdge
Menu::bestPathDijkstra
Graph::insertTemporaryNode
Graph::calculateDistance
Graph::printPath
Graph::printPathLinesAlgorithm
Menu::readInputBestAlgorithm
Graph::removeTemporaryNodes
Menu::showAlgorithmOptions

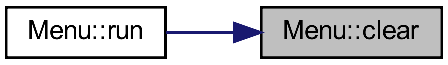
Here is the caller graph for this function:



3.6.1.8 clear()

```
void Menu::clear ( )  [static]
```

It clears the console by calling system(). (It might have problems in Clion, but it works well in the terminal) Here is the caller graph for this function:



3.6.1.9 createGraphLine()

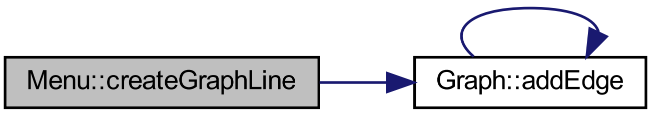
```
void Menu::createGraphLine (
             string line )
```

Reads a string representing the attributes of a line with them, creates a new line. It also adds this new line to every edges that it passes through.

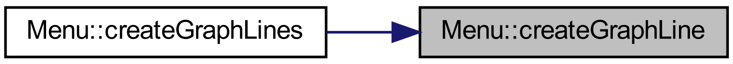
Parameters

line	Attributes of a line

Here is the call graph for this function:



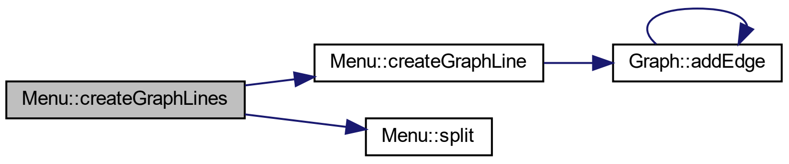
Here is the caller graph for this function:



3.6.1.10 createGraphLines()

```
void Menu::createGraphLines ( )
```

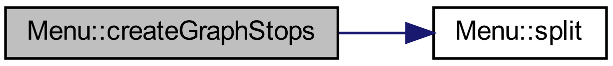
Read lines.csv and for every line on the document calls the method responsible for creating lines. Here is the call graph for this function:



3.6.1.11 createGraphStops()

```
void Menu::createGraphStops ( )
```

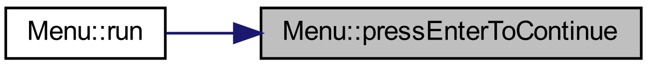
Reads stops.csv and for every line on the document it creates a node and adds it to the graph. Here is the call graph for this function:



3.6.1.12 pressEnterToContinue()

```
void Menu::pressEnterToContinue ( ) [static]
```

Waits for user to press enter to continue. Here is the caller graph for this function:



3.6.1.13 readDouble()

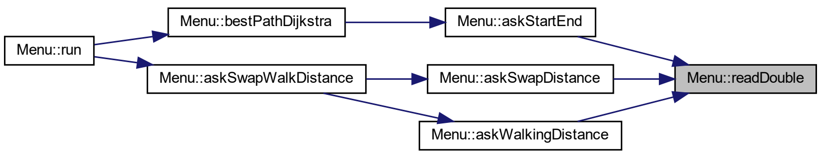
```
double Menu::readDouble ( ) [static]
```

Reads double from the user (General use).

Returns

Value given by the user

Here is the caller graph for this function:



3.6.1.14 readInputBestAlgorithm()

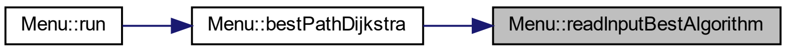
```
int Menu::readInputBestAlgorithm ( ) [static]
```

Read an input from the user regarding their algorithm of choice.

Returns

The chosen algorithm's number

Here is the caller graph for this function:



3.6.1.15 readInputMenu()

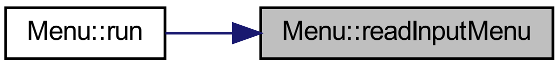
```
int Menu::readInputMenu ( )  [static]
```

Reads input from the user in the main menu.

Returns

choice of the user

Here is the caller graph for this function:



3.6.1.16 readInputStopsOrLocation()

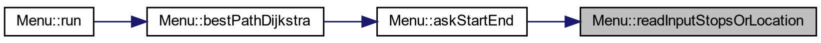
```
int Menu::readInputStopsOrLocation ( )  [static]
```

Reads an input from the user regarding if they want to tell the stops' code or coordinates.

Returns

The value of the user's choice.

Here is the caller graph for this function:



3.6.1.17 readInt()

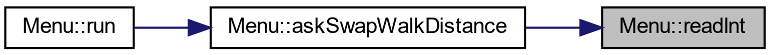
```
int Menu::readInt ( ) [static]
```

Reads integer from the user (General use).

Returns

Value given by the user

Here is the caller graph for this function:



3.6.1.18 readString()

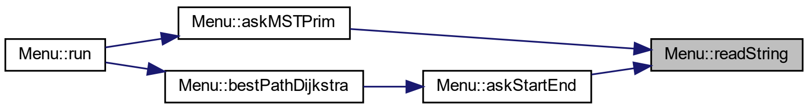
```
string Menu::readString ( ) [static]
```

Reads string from the user (General use).

Returns

Value given by the user

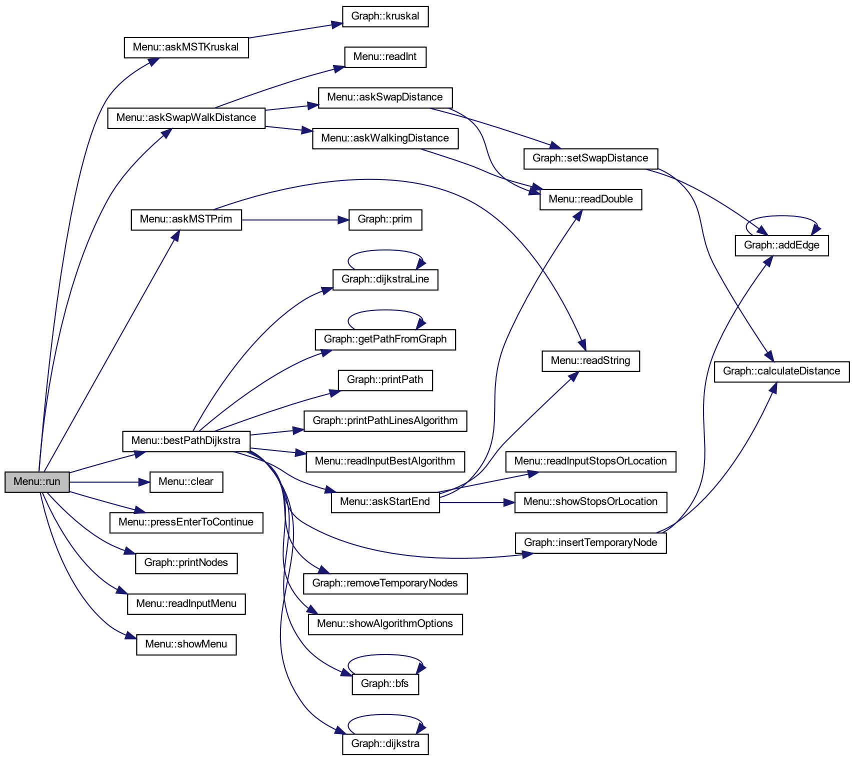
Here is the caller graph for this function:



3.6.1.19 run()

```
void Menu::run ( )
```

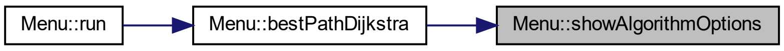
Takes the user's input to determine which functionality to execute. Here is the call graph for this function:



3.6.1.20 showAlgorithmOptions()

```
void Menu::showAlgorithmOptions ( )  [static]
```

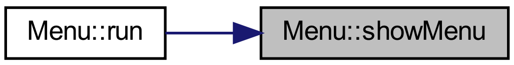
Prints the options of the priorities that can define "the best path" for the user. Here is the caller graph for this function:



3.6.1.21 showMenu()

```
void Menu::showMenu ( )  [static]
```

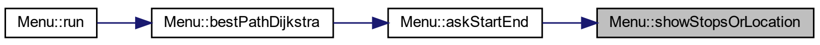
Prints the main menu showing every functionality that the user can choose. Here is the caller graph for this function:



3.6.1.22 showStopsOrLocation()

```
void Menu::showStopsOrLocation ( )  [static]
```

Prints the options to indicate the stops' code or coordinates. Here is the caller graph for this function:



3.6.1.23 split()

```
vector< string > Menu::split (
             string line,
             string delimeter )  [static]
```

It separates a string in to a vector of strings by the delimiter.

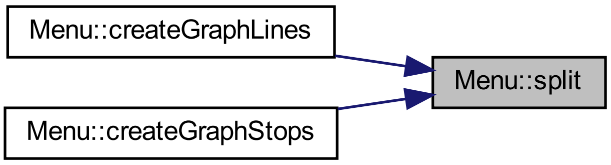
Parameters

line	The line's code
delimeter	The expression where the systems splits the string

Returns

Vector of strings

Here is the caller graph for this function:

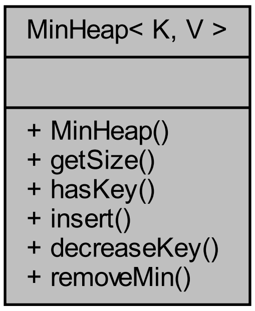


The documentation for this class was generated from the following files:

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Menu.h
- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Menu.cpp

3.7 MinHeap< K, V > Class Template Reference

Collaboration diagram for MinHeap< K, V >:



Public Member Functions

- **MinHeap** (int n, const K &notFound)
- int **getSize** ()
- bool **hasKey** (const K &key)
- void **insert** (const K &key, const V &value)
- void **decreaseKey** (const K &key, const V &value)
- K **removeMin** ()

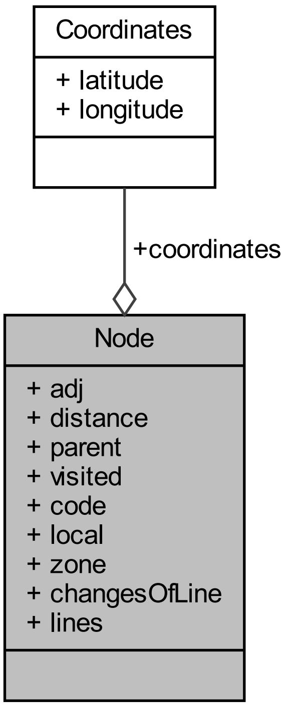
The documentation for this class was generated from the following file:

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/MinHeap.h

3.8 Node Struct Reference

```
#include <Node.h>
```

Collaboration diagram for Node:



Public Attributes

- list< Edge > **adj**
- double **distance**
- int **parent**
- bool **visited**
- string **code**
- string **local**
- string **zone**
- Coordinates **coordinates**
- int **changesOfLine**
- map< string, pair< double, int > > **lines**

3.8.1 Detailed Description

A Node represents a Bus Stop

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

- C:/clion-projetos/BusNavigationProgram/BusNavigationProgram/src/Node.h

Chapter 4

File Documentation

4.1 DisjointSets.h

```cpp
//
// Created by johnny on 29/01/22.
//

#ifndef _DISJOINTSETS_H_
#define _DISJOINTSETS_H_
#include <unordered_map>

template <class T> class DisjointSets {
    struct Node {
        T parent;
        int myrank; // to use on union by rank
    };

    std::unordered_map<T, Node> a;

public:
    void makeSet(const T &x); // Create a set with a single element x
    T find(const T &x);       // Find the representative of the set of element x
    void unite(const T &x, const T &y); // Merge the sets of elements x and y
};

// Create a set with a single element x
template <class T> void DisjointSets<T>::makeSet(const T &x) {
    a[x].parent = x;
    a[x].myrank = 0;
}

// Find the representative of the set of element x
template <class T> T DisjointSets<T>::find(const T &x) {
    if (a[x].parent != x)
        a[x].parent = find(a[x].parent);

    return a[x].parent;
}

// Merge the sets of elements x and y
template <class T> void DisjointSets<T>::unite(const T &x, const T &y) {
    T xRoot = find(x);
    T yRoot = find(y);

    if (xRoot == yRoot)
        return;

    if (a[xRoot].myrank < a[yRoot].myrank)
        a[xRoot].parent = yRoot;
    else if (a[xRoot].myrank > a[yRoot].myrank)
        a[yRoot].parent = xRoot;
    else {
        a[yRoot].parent = xRoot;
        ++(a[xRoot].myrank);
    }
}

#endif
```

4.2 Graph.h

```
#pragma once
#ifndef BUSNAVIGATIONPROGRAM_GRAPH_H
#include <list>
#include <stack>
#include <vector>
#include <queue>
#include <iostream>
#include <unordered_map>
#include <cmath>
#include "Node.h"
#include "MinHeap.h"
#include "DisjointSets.h"

using namespace std;

class Graph {
    int n;              // Graph size (vertices are numbered from 1 to n)
    double walkingDistance = 0.001; //km
    double swapDistance = 0.0;
    bool hasDir;        // false: undirect; true: directed
    vector<Node> nodes; // The list of nodes being represented
    unordered_map<string, int> positions;


public:
    // Constructor: nr nodes and direction (default: directed)
    Graph(int nodes = 0, bool dir = true);
    Graph(vector<Node> nodes, unordered_map<string, int> positions, bool dir = true);
    vector<Node>& getNodes();
    double getWalkingDistance();
    double getSwapDistance();
    unordered_map<string, int> getPositions();
    void setWalkingDistance(double walkingDist);
    void setSwapDistance(double swapDist);

    // Add edge from source to destination with a certain weight
    void addEdge(string src, string dest, string line);
    void addEdge(int src, int dest, string line);
    void addEdge(int src, int dest, double weight = 1.0);
    double calculateDistance(Coordinates c1, Coordinates c2);
    double calculateDistance(int src, int dest);
    int calculateChangesOfLine(map<string, pair<double, int» lines, int numPrevChangesOfLines, string
    newLine);
    double calculateWeightDijkstraLine(int min, double edgeWeight, string newLine);

    long double prim(int r);
    long double kruskal();

    double bfs(string src, string dest);
    double bfs(int a, int b);
    double dijkstra(string src, string dest);
    double dijkstra(int a, int b);
    double dijkstraLine(string src, string dest);
    double dijkstraLine(int a, int b);

    stack<int> getPathFromGraph(string src, string dest);
    stack<int> getPathFromGraph(int a, int b);

    void printPath(stack<int> path);
    void printPathLinesAlgorithm(stack<int> path);
    void printNodes();
    void insertTemporaryNode(Coordinates c, bool startType);
    void removeTemporaryNodes();
};


#endif //BUSNAVIGATIONPROGRAM_GRAPH_H
```

4.3 Menu.h

```
#ifndef BUSNAVIGATIONPROGRAM_MENU_H
#define BUSNAVIGATIONPROGRAM_MENU_H

#include <string>
#include <vector>
#include <unordered_map>
#include <fstream>
#include <iostream>
#include "Graph.h"

using namespace std;
```

```

class Menu {
    string directory;
    Graph graph;

public:
    Menu(string directory);
    Graph& getGraph();

    void run();
    static void showMenu();
    static void showStopsOrLocation();
    static void showAlgorithmOptions();

    static int readInputMenu();
    static int readInputStopsOrLocation();
    static int readInputBestAlgorithm();
    static double readDouble();
    static int readInt();
    static string readString();
    static void pressEnterToContinue();

    void createGraphStops();
    void createGraphLines();
    void createGraphLine(string line);

    int askStartEnd(string& stopBegin, string& stopEnd, Coordinates& cBegin, Coordinates& cEnd);

    void bestPathDijkstra();
    void askMSTPrim();
    void askMSTKruskal();

    void askSwapWalkDistance();
    void askWalkingDistance();
    void askSwapDistance();

    static vector<string> split(string line, string delimeter);
    static void clear();
};


#endif //BUSNAVIGATIONPROGRAM_MENU_H
```

4.4 MinHeap.h

```
#ifndef BUSNAVIGATIONPROGRAM_MINHEAP_H
#define BUSNAVIGATIONPROGRAM_MINHEAP_H


#include <vector>
#include <unordered_map>

#define LEFT(i) (2*(i))
#define RIGHT(i) (2*(i)+1)
#define PARENT(i) ((i)/2)

using namespace std;

// Binary min-heap to represent integer keys of type K with values (priorities) of type V
template <class K, class V>
class MinHeap {
    struct Node { // An element of the heap: a pair (key, value)
        K key;
        V value;
    };

    int size;                  // Number of elements in heap
    int maxSize;               // Maximum number of elements in heap
    vector<Node> a;            // The heap array
    unordered_map<K, int> pos; // maps a key into its position on the array a
    const K KEY_NOT_FOUND;

    void upHeap(int i);
    void downHeap(int i);
    void swap(int i1, int i2);

public:
    MinHeap(int n, const K& notFound); // Create a min-heap for a max of n pairs (K,V) with notFound
    returned when empty
    int getSize();              // Return number of elements in the heap
    bool hasKey(const K& key);  // Heap has key?
    void insert(const K& key, const V& value);      // Insert (key, value) on the heap
    void decreaseKey(const K& key, const V& value); // Decrease value of key
    K removeMin(); // remove and return key with smaller value
};

// ---------------------------------------------

// Make a value go "up the tree" until it reaches its position
template <class K, class V>
void MinHeap<K,V>::upHeap(int i) {
    while (i>1 && a[i].value < a[PARENT(i)].value) { // while pos smaller than parent, keep swapping to
    upper position
        swap(i, PARENT(i));
        i = PARENT(i);
    }
}

// Make a value go "down the tree" until it reaches its position
template <class K, class V>
void MinHeap<K,V>::downHeap(int i) {
    while (LEFT(i) <= size) { // while within heap limits
        int j = LEFT(i);
        if (RIGHT(i)<=size && a[RIGHT(i)].value < a[j].value) j = RIGHT(i); // choose smaller child
        if (a[i].value < a[j].value) break;   // node already smaller than children, stop
        swap(i, j);                    // otherwise, swap with smaller child
        i = j;
    }
}

// Swap two positions of the heap (update their positions)
template <class K, class V>
void MinHeap<K,V>::swap(int i1, int i2) {
    Node tmp = a[i1]; a[i1] = a[i2]; a[i2] = tmp;
    pos[a[i1].key] = i1;
    pos[a[i2].key] = i2;
}

// ---------------------------------------------

// Create a min-heap for a max of n pairs (K,V) with notFound returned when empty
template <class K, class V>
MinHeap<K,V>::MinHeap(int n, const K& notFound) : KEY_NOT_FOUND(notFound), size(0), maxSize(n), a(n+1) {
}

// Return number of elements in the heap
template <class K, class V>
int MinHeap<K,V>::getSize() {
    return size;
}


// Heap has key?
template <class K, class V>
bool MinHeap<K, V>::hasKey(const K& key) {
    return pos.find(key) != pos.end();
}

// Insert (key, value) on the heap
template <class K, class V>
void MinHeap<K,V>::insert(const K& key, const V& value) {
    if (size == maxSize) return; // heap is full, do nothing
    if (hasKey(key)) return;     // key already exists, do nothing
    a[++size] = {key, value};
    pos[key] = size;
    upHeap(size);
}

// Decrease value of key to the indicated value
template <class K, class V>
void MinHeap<K,V>::decreaseKey(const K& key, const V& value) {
    if (!hasKey(key)) return; // key does not exist, do nothing
    int i = pos[key];
    if (value > a[i].value) return; // value would increase, do nothing
    a[i].value = value;
    upHeap(i);
}

// remove and return key with smaller value
template <class K, class V>
K MinHeap<K,V>::removeMin() {
    if (size == 0) return KEY_NOT_FOUND;
    K min = a[1].key;
    pos.erase(min);
    a[1] = a[size--];
    downHeap(1);
    return min;
}

#endif //BUSNAVIGATIONPROGRAM_MINHEAP_H
```

4.5 Node.h

```
#ifndef BUSNAVIGATIONPROGRAM_NODE_H
#define BUSNAVIGATIONPROGRAM_NODE_H

#include <list>
#include <string>
#include <map>

using namespace std;


struct Coordinates {
    double latitude;
    double longitude;
};

struct Edge {
    int dest;   // Destination node
    double weight; // An integer weight
    string line;

};

struct EdgeKruskal {
    int src;    // Source node
    int dest;   // Destination node
    double weight; // An integer weight

    // For Kruskal's
    bool operator<(const EdgeKruskal &other) const {
        return weight < other.weight;
    }

    bool operator==(const EdgeKruskal &other) const {
        return ((src == other.src && dest == other.dest) ||
                (src == other.dest && dest == other.src)) &&
               weight == other.weight;
    }
};

 struct Node {
    list<Edge> adj; // The list of outgoing edges (to adjacent nodes)
    double distance;
    int parent;  // previous node of the path
    bool visited;
    string code;  //unique code of the bus stop
    string local;
    string zone;
    Coordinates coordinates;
    int changesOfLine;  //used in dijkstraLine algorithm calculation
    map<string, pair<double, int» lines; // used for dijkstraLine algorithm calculations
};


#endif //BUSNAVIGATIONPROGRAM_NODE_H
```

Index