Smart Traffic Management System

1.0.0

Generated by Doxygen 1.9.8

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 Accident_roads Class Reference

Class to handle accidents and road closures.

`#include <accidents.h>`

Public Member Functions

- void **blockRoad** (const std::string &start, const std::string &end, Graph &cityGraph)
- Accident_roads ()

  Default constructor for the Accident_roads class.
- ~Accident_roads ()

  Destructor for the Accident_roads class.
- void loadRoadData (Graph &graph)

  Load road closure data from a file and update the graph.
- void markIntersectionsAsBlocked (Graph &graph, const std::string &intersection1, const std::string &intersection2, bool isBlocked)

  Mark intersections as blocked or unblocked in the graph.
- void displayBlockedIntersections (Graph &graph)

  Display the blocked intersections in the graph.
- void displayUnderRepairIntersections (Graph &graph)

  Display the under-repair intersections in the graph.
- void displayBlockedRoads ()

  Display the blocked roads in the network.
- void displayUnderRepairRoads ()

  Display the under-repair roads in the network.

3.1.1 Detailed Description

Class to handle accidents and road closures.

This class manages the data related to accidents or road closures that affect the road network. It maintains a linked list of accident records and provides methods to load, mark, and display blocked intersections.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Accident_roads()

```
Accident_roads::Accident_roads ( )
```

Default constructor for the Accident_roads class.

Initializes an empty list of accident records.

3.1.2.2 ~Accident_roads()

```
Accident_roads::~Accident_roads ( )
```

Destructor for the Accident_roads class.

Frees any memory used by the linked list of accident records.

3.1.3 Member Function Documentation

3.1.3.1 displayBlockedIntersections()

```
void Accident_roads::displayBlockedIntersections (
            Graph & graph )
```

Display the blocked intersections in the graph.

This function prints the names of all intersections that are currently blocked in the graph.

Parameters

graph	The graph containing the intersections and their blocked status.

3.1.3.2 displayBlockedRoads()

```
void Accident_roads::displayBlockedRoads ( )
```

Display the blocked roads in the network.

This function prints the list of roads that are currently blocked.

3.1.3.3 displayUnderRepairIntersections()

```
void Accident_roads::displayUnderRepairIntersections (
            Graph & graph )
```

Display the under-repair intersections in the graph.

This function prints the names of all intersections that are currently under repair in the graph.

Parameters

graph	The graph containing the intersections and their under-repair status.

3.1.3.4 displayUnderRepairRoads()

```
void Accident_roads::displayUnderRepairRoads ( )
```

Display the under-repair roads in the network.

This function prints the list of roads that are currently under repair.

3.1.3.5 loadRoadData()

```
void Accident_roads::loadRoadData (
            Graph & graph )
```

Load road closure data from a file and update the graph.

This function reads road closure data from a file (such as a CSV file) and updates the given graph by marking the affected intersections as blocked or unblocked.

Parameters

graph	The graph containing intersections and roads to be updated with road closures.

3.1.3.6 markIntersectionsAsBlocked()

```
void Accident_roads::markIntersectionsAsBlocked (
            Graph & graph,
            const std::string & intersection1,
            const std::string & intersection2,
            bool isBlocked )
```

Mark intersections as blocked or unblocked in the graph.

This function updates the blocked status of the roads between the specified intersections in the given graph. It ensures that the affected roads are marked as either blocked or unblocked.

Parameters

graph	The graph containing the intersections and roads.
intersection1	The name of the first intersection involved in the road closure.
intersection2	The name of the second intersection involved in the road closure.
isBlocked	The new status of the road (true if blocked, false if unblocked).

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

- accidents.h
- accidents.cpp

3.2 CongestionMaxHeap Class Reference

A class to represent a max heap specifically for managing road congestion data.

`#include <congestionMaxHeap.h>`

Public Member Functions

- CongestionMaxHeap ()
  Construct a new Congestion Max Heap object.
- void makeHeap (RoadNode *hashTableArray, int size)
  Makes a max heap from the roads in the hashtable array.
- void insert (RoadNode *newNode)
  Inserts a new RoadNode into the heap.
- void heapifyUp (RoadNode *&newNode)
  Heapifies up the heap after insertion.
- void printHeap ()
  Prints the heap.
- RoadNode * mostCongested ()
  returns root of the heap without removing it
- void **inorder** ()
  Prints the heap inorder.
- void heapifyDown (RoadNode *node)
  Function to heapify down the heap.

3.2.1 Detailed Description

A class to represent a max heap specifically for managing road congestion data.

This class provides functionalities to create and manage a max heap of RoadNode objects, which are used to monitor and manage road congestion.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 CongestionMaxHeap()

```
CongestionMaxHeap::CongestionMaxHeap ( )
```

Construct a new Congestion Max Heap object.

Initializes root to nullptr

3.2.3 Member Function Documentation

3.2.3.1 heapifyDown()

```
void CongestionMaxHeap::heapifyDown (
            RoadNode * node )
```

Function to heapify down the heap.

Parameters

node	

3.2.3.2 heapifyUp()

```
void CongestionMaxHeap::heapifyUp (
            RoadNode *& newNode )
```

Heapifies up the heap after insertion.

Parameters

newNode	The RoadNode to be heapified up

3.2.3.3 insert()

```
void CongestionMaxHeap::insert (
            RoadNode * newNode )
```

Inserts a new RoadNode into the heap.

inserts preserving the structure property of the heap and heapifies up after insertion

Parameters

newNode	The RoadNode to be inserted

3.2.3.4 makeHeap()

```
void CongestionMaxHeap::makeHeap (
            RoadNode * hashTableArray,
            int size )
```

Makes a max heap from the roads in the hashtable array.

Parameters

hashTableArray	hashTableArray is from the congestion monitoring class
size	size of the hashtable array

3.2.3.5 mostCongested()

```
RoadNode * CongestionMaxHeap::mostCongested ( )
```

returns root of the heap without removing it

Returns

RoadNode*

3.2.3.6 printHeap()

```
void CongestionMaxHeap::printHeap ( )
```

Prints the heap.

Uses BFS to print the heap

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

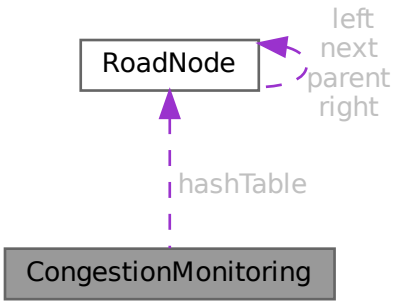
- congestionMaxHeap.h
- congestionMaxHeap.cpp

3.3 CongestionMonitoring Class Reference

A class for tracking the number of vehicles on a road using a hash table.

```
#include <congestionMonitoring.h>
```

Collaboration diagram for CongestionMonitoring:



Public Member Functions

- CongestionMonitoring (Vehicle *vehiclesHead)

  Construct a new Congestion Monitoring object.
- void makeHashTable (Vehicle *vehiclesHead)

  Creates the hash table from the list of vehicles.
- void updateHashTable (Vehicle *prevPos, Vehicle *currentPos)

  Updates the hash table with the new position of a vehicle.
- void **printHashTable** ()

  Prints the contents of the hash table.
- void deleteTable ()

  Deletes the hash table.
- RoadNode * findRoadNode (char start, char end)

  Finds a road segment in the hash table.
- int getTravelTime (char start, char end, int prevTime)

  Get the updated Travel Time in seconds after considering the congestion and the time of the day (time elapsed)
- int getTravelTime (char start, char end, Graph &cityGraph)

  Get the Travel Time in seconds between two points.
- int numberOfCongestionEvents ()

  Returns the performance metric number of congestion events.

Public Attributes

- RoadNode **hashTable** [HASH_TABLE_SIZE]

3.3.1 Detailed Description

A class for tracking the number of vehicles on a road using a hash table.

This class provides functionalities to create, update, and manage a hash table that tracks the number of vehicles on different roads.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 CongestionMonitoring()

```
CongestionMonitoring::CongestionMonitoring (
            Vehicle * vehiclesHead )
```

Construct a new Congestion Monitoring object.

Initializes the hash table and populates it using the provided list of vehicles.

Parameters

vehiclesHead	Pointer to the head of the linked list of vehicles.

3.3.3 Member Function Documentation

3.3.3.1 deleteTable()

```
void CongestionMonitoring::deleteTable ( )
```

Deletes the hash table.

This function clears the hash table, removing all road segments.

3.3.3.2 findRoadNode()

```
RoadNode * CongestionMonitoring::findRoadNode (
            char start,
            char end )
```

Finds a road segment in the hash table.

This function searches for a road segment defined by the start and end points in the hash table.

Parameters

start	The starting point of the road segment.
end	The ending point of the road segment.

Returns

Pointer to the RoadNode representing the road segment, or nullptr if not found.

3.3.3.3 getTravelTime() [1/2]

```
int CongestionMonitoring::getTravelTime (
            char start,
            char end,
            Graph & cityGraph )
```

Get the Travel Time in seconds between two points.

this reutrns the travel time between two points in seconds after considering the congestion and the time of the day

Parameters

start	
end	
cityGraph	

Returns

int

3.3.3.4 getTravelTime() [2/2]

```
int CongestionMonitoring::getTravelTime (
            char start,
            char end,
            int prevTime )
```

Get the updated Travel Time in seconds after considering the congestion and the time of the day (time elapsed)

Parameters

start	
end	
prevTime	

Returns

int

3.3.3.5 makeHashTable()

```
void CongestionMonitoring::makeHashTable (
            Vehicle * vehiclesHead )
```

Creates the hash table from the list of vehicles.

This function populates the hash table with road segments based on the provided list of vehicles.

Parameters

vehiclesHead	Pointer to the head of the linked list of vehicles.

3.3.3.6 numberOfCongestionEvents()

```
int CongestionMonitoring::numberOfCongestionEvents ( )
```

Returns the performance metric number of congestion events.

Returns

int

3.3.3.7 updateHashTable()

```
void CongestionMonitoring::updateHashTable (
            Vehicle * prevPos,
            Vehicle * currentPos )
```

Updates the hash table with the new position of a vehicle.

This function updates the hash table to reflect the new position of a vehicle.

Parameters

prevPos	Pointer to the previous position of the vehicle.
currentPos	Pointer to the current position of the vehicle.

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

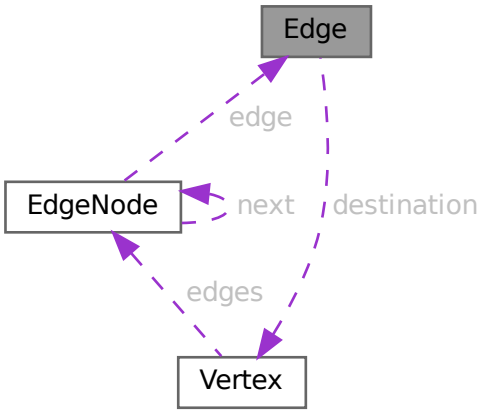
- congestionMonitoring.h
- congestionMonitoring.cpp

3.4 Edge Class Reference

Represents a road between two intersections.

`#include <graph.h>`

Collaboration diagram for Edge:



Public Member Functions

- bool isBlocked () const

  Checks if the edge is blocked.
- bool isUnderRepaired () const

  Checks if the Edge is under repair.
- void setBlocked (bool status)

  Sets the blocked status of the edge.
- void setUnderRepaired (bool status)

  Sets the under-repair status of the edge.
- Edge (Vertex ∗destination, int travelTime)

  Constructs an Edge object.

Public Attributes

- Vertex * destination
- int travelTime
- bool blocked
- bool underRepaired

3.4.1 Detailed Description

Represents a road between two intersections.

An Edge object stores information about a road between two intersections, including the destination intersection and the travel time between them.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Edge()

```
Edge::Edge (
            Vertex * destination,
            int travelTime )
```

Constructs an Edge object.

Parameters

destination	The destination intersection.
travelTime	The travel time to the destination.

3.4.3 Member Function Documentation

3.4.3.1 isBlocked()

```
bool Edge::isBlocked ( ) const
```

Checks if the edge is blocked.

Returns

true if the edge is blocked, false otherwise.

3.4.3.2 isUnderRepaired()

```
bool Edge::isUnderRepaired ( ) const
```

Checks if the Edge is under repair.

Returns

true if the edge is under repair, false otherwise.

3.4.3.3 setBlocked()

```
void Edge::setBlocked (
            bool status )
```

Sets the blocked status of the edge.

Parameters

status	The new blocked status (true for blocked, false for not blocked).

3.4.3.4 setUnderRepaired()

```
void Edge::setUnderRepaired (
            bool status )
```

Sets the under-repair status of the edge.

Parameters

status	The new under-repair status (true for under repair, false for not under repair).

3.4.4 Member Data Documentation

3.4.4.1 blocked

```
bool Edge::blocked
```

Whether the Edge is blocked or not

3.4.4.2 destination

```
Vertex* Edge::destination
```

Destination vertex (intersection)

3.4.4.3 travelTime

```
int Edge::travelTime
```

Travel time to the destination

3.4.4.4 underRepaired

```
bool Edge::underRepaired
```

Whether the Edge is under repair

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

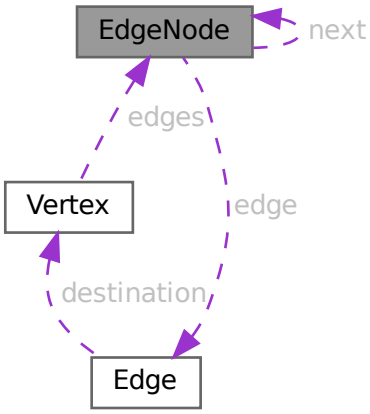
- graph.h
- graph.cpp

3.5 EdgeNode Class Reference

Represents a node in the adjacency list for edges.

```
#include <graph.h>
```

Collaboration diagram for EdgeNode:



Public Member Functions

- EdgeNode (Edge *edge)

  Constructs an EdgeNode object.

Public Attributes

- Edge * edge
- EdgeNode * next

3.5.1 Detailed Description

Represents a node in the adjacency list for edges.

An EdgeNode is used to store an Edge in the adjacency list of a vertex. It also maintains a pointer to the next EdgeNode in the list, allowing multiple edges to be linked together.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 EdgeNode()

```
EdgeNode::EdgeNode (
            Edge * edge )
```

Constructs an EdgeNode object.

Parameters

edge	The edge that this node will represent.

3.5.3 Member Data Documentation

3.5.3.1 edge

```
Edge* EdgeNode::edge
```

The edge

3.5.3.2 next

```
EdgeNode* EdgeNode::next
```

Pointer to the next edge

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

- graph.h
- graph.cpp

3.6 GPS Class Reference

Class for managing GPS navigation and finding all possible paths between two vertices.

```
#include <Route.h>
```

Public Member Functions

- GPS (Graph *graph)

  Constructor for the GPS class.
- void printAllPaths (const std::string &startName, const std::string &endName)

  Function to print all paths between two vertices along with their total weights.
- string rerouteEmergencyVehicle (const string &startName, const string &endName)

  Function to reroute an emergency vehicle around a blocked road.
- string getPathAsString (const string &startName, const string &endName)

  Function to get the path as a string between two vertices.
- void findAllOptimalPaths (Vertex *start, Vertex *end, string path[ ], int pathIndex, string allPaths[ ][MAX_↩ VERTICES], int &allPathsCount, bool visited[ ], int totalWeight[ ], int &totalWeightCount)

  Helper function to find all optimal paths between two vertices.
- void printAllPathsDijkstra (const string &startName, const string &endName)

  Function to print all paths using Dijkstra's algorithm.
- int heuristic (const Vertex *a, const Vertex *b)

  Heuristic function to estimate the distance between two vertices.

3.6.1 Detailed Description

Class for managing GPS navigation and finding all possible paths between two vertices.

This class uses depth-first search (DFS) to find all possible paths between two vertices in a graph, along with calculating and storing the total weight of each path.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 GPS()

```
GPS::GPS (
            Graph * graph )
```

Constructor for the GPS class.

Initializes the GPS object with a reference to a graph object.

Parameters

graph	A pointer to the Graph object.

3.6.3 Member Function Documentation

3.6.3.1 findAllOptimalPaths()

```
void GPS::findAllOptimalPaths (
            Vertex * start,
            Vertex * end,
```

```
            string path[],
            int pathIndex,
            string allPaths[][MAX_VERTICES],
            int & allPathsCount,
            bool visited[],
            int totalWeight[],
            int & totalWeightCount )
```

Helper function to find all optimal paths between two vertices.

This function uses DFS to find all optimal paths between the start and end vertices, considering path weights.

Parameters

start	The starting vertex.
end	The destination vertex.
path	The current path being explored.
pathIndex	The index of the current vertex in the path.
allPaths	The array to store all found paths.
allPathsCount	A counter for the total number of paths found.
visited	An array to track visited vertices.
totalWeight	An array to store the total weights of the paths.
totalWeightCount	A counter for the total number of weights.

3.6.3.2 getPathAsString()

```
string GPS::getPathAsString (
            const string & startName,
            const string & endName )
```

Function to get the path as a string between two vertices.

This function returns a string representation of the path from the start vertex to the end vertex.

Parameters

startName	The name of the starting vertex.
endName	The name of the destination vertex.

Returns

A string representing the path.

3.6.3.3 heuristic()

```
int GPS::heuristic (
            const Vertex * a,
            const Vertex * b )
```

Heuristic function to estimate the distance between two vertices.

This function calculates the heuristic value between two vertices. It is typically used for A$*$ search.

Parameters

a	The first vertex.
b	The second vertex.

Returns

The heuristic value between the two vertices.

3.6.3.4 printAllPaths()

```
void GPS::printAllPaths (
            const std::string & startName,
            const std::string & endName )
```

Function to print all paths between two vertices along with their total weights.

This function initiates the process of finding all paths from the start vertex to the end vertex using depth-first search. It then prints each path and its associated total weight.

Parameters

startName	The name of the starting vertex.
endName	The name of the destination vertex.

3.6.3.5 printAllPathsDijkstra()

```
void GPS::printAllPathsDijkstra (
            const string & startName,
            const string & endName )
```

Function to print all paths using Dijkstra's algorithm.

This function finds and prints the shortest paths between two vertices using Dijkstra's algorithm.

Parameters

startName	The name of the starting vertex.
endName	The name of the destination vertex.

3.6.3.6 rerouteEmergencyVehicle()

```
string GPS::rerouteEmergencyVehicle (
            const string & startName,
            const string & endName )
```

Function to reroute an emergency vehicle around a blocked road.

This function finds an alternate path for an emergency vehicle to reach its destination by avoiding a blocked road. It uses the printAllPaths function to find all possible paths and selects the shortest path that avoids the blocked road.

Parameters

startName	The name of the starting intersection.
endName	The name of the destination intersection.

Returns

A string representing the alternate path.

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

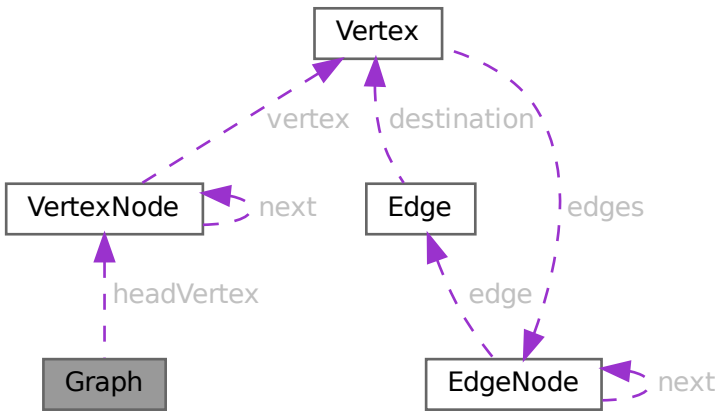
- Route.h
- route.cpp

3.7 Graph Class Reference

Represents the road network as a graph.

`#include <graph.h>`

Collaboration diagram for Graph:



Public Member Functions

- **Graph** ()

  Constructs an empty Graph object.
- ~**Graph** ()

  Destructor for the Graph object.
- void addVertex (const std::string &name)

  Adds a vertex (intersection) to the graph.

- Vertex * findVertex (const std::string &name)

  Finds a vertex by its name.
- void removeVertex (const string &name)

  Removes a vertex (intersection) from the graph.
- void addEdge (const std::string &start, const std::string &end, int travelTime)

  Adds a road (edge) between two intersections.
- void removeEdge (const string &start, const string &end)

  Removes a road (edge) between two intersections.
- void loadRoadData (const std::string &filename="road_network.csv")

  Loads road network data from a file.
- void **displayRoadStatuses** ()

  Displays the current statuses of all roads in the graph.
- void markEdgeAsBlocked (const string &intersection1, const string &intersection2, bool isBlocked)

  Marks the specified intersections as blocked or unblocked.
- void markEdgesAsUnderRepaired (const string &intersection1, const string &intersection2, bool isUnder↩ Repaired)

  Marks the specified Edges as under repair or not.
- void **displayBlockedEdges** ()

  Displays the blocked intersections in the graph.
- void addEdgeToVertex (Vertex *vertex, Edge *edge)

  Adds an edge to the specified vertex.
- void **printAdjacencyList** ()

  Prints the adjacency list representation of the graph.
- bool isBlocked (const std::string &nodeName1, const std::string &nodeName2)

  Checks if a vertex is blocked.
- void getNeighbors (const std::string &nodeName, std::string *neighbors, int &count)

  Gets the neighbors of a specified vertex.
- int getEdgeWeight (const std::string &start, const std::string &end)

  Gets the travel time between two vertices.
- void getVertices (std::string *vertices, int &count)

  Gets all vertices in the graph.
- int getVertexCount ()

  Gets the total number of vertices in the graph.
- void getAllEdges (std::string edges[ ][3], int &count)

  Gets all edges in the graph.

Public Attributes

- VertexNode * headVertex

3.7.1 Detailed Description

Represents the road network as a graph.

The Graph class manages the entire road network, consisting of vertices (intersections) and edges (roads connecting the intersections). It provides methods for adding vertices and edges, displaying the road statuses, and marking intersections as blocked.

3.7.2 Member Function Documentation

3.7.2.1 addEdge()

```
void Graph::addEdge (
            const std::string & start,
            const std::string & end,
            int travelTime )
```

Adds a road (edge) between two intersections.

Parameters

start	The name of the starting intersection.
end	The name of the destination intersection.
travelTime	The travel time between the two intersections.

3.7.2.2 addEdgeToVertex()

```
void Graph::addEdgeToVertex (
            Vertex * vertex,
            Edge * edge )
```

Adds an edge to the specified vertex.

Parameters

vertex	The vertex to which the edge will be added.
edge	The edge to add to the vertex.

3.7.2.3 addVertex()

```
void Graph::addVertex (
            const std::string & name )
```

Adds a vertex (intersection) to the graph.

Parameters

name	The name of the new intersection.

3.7.2.4 findVertex()

```
Vertex * Graph::findVertex (
            const std::string & name )
```

Finds a vertex by its name.

Parameters

name	The name of the vertex to find.

Returns

A pointer to the Vertex object if found, or nullptr if not found.

3.7.2.5 getAllEdges()

```
void Graph::getAllEdges (
            std::string edges[][3],
            int & count )
```

Gets all edges in the graph.

Parameters

vertices	Array to store the names of edges.
count	The number of edges found.

3.7.2.6 getEdgeWeight()

```
int Graph::getEdgeWeight (
            const std::string & start,
            const std::string & end )
```

Gets the travel time between two vertices.

Parameters

start	The starting vertex name.
end	The destination vertex name.

Returns

The travel time between the two vertices, or -1 if no edge exists.

3.7.2.7 getNeighbors()

```
void Graph::getNeighbors (
            const std::string & nodeName,
            std::string * neighbors,
            int & count )
```

Gets the neighbors of a specified vertex.

Parameters

nodeName	The name of the vertex.
neighbors	Array to store the names of neighboring vertices.
count	The number of neighbors found.

3.7.2.8 getVertexCount()

```
int Graph::getVertexCount ( )
```

Gets the total number of vertices in the graph.

Returns

The number of vertices in the graph.

3.7.2.9 getVertices()

```
void Graph::getVertices (
            std::string * vertices,
            int & count )
```

Gets all vertices in the graph.

Parameters

vertices	Array to store the names of vertices.
count	The number of vertices found.

3.7.2.10 isBlocked()

```
bool Graph::isBlocked (
            const std::string & nodeName1,
            const std::string & nodeName2 )
```

Checks if a vertex is blocked.

Parameters

nodeName	The name of the vertex to check.

Returns

true if the vertex is blocked, false otherwise.

3.7.2.11 loadRoadData()

```
void Graph::loadRoadData (
            const std::string & filename = "road_network.csv" )
```

Loads road network data from a file.

Parameters

filename	The name of the CSV file to load road data from (default is "road_network.csv").

3.7.2.12 markEdgeAsBlocked()

```
void Graph::markEdgeAsBlocked (
            const string & intersection1,
            const string & intersection2,
            bool isBlocked )
```

Marks the specified intersections as blocked or unblocked.

Parameters

intersection1	The name of the first intersection.
intersection2	The name of the second intersection.
isBlocked	The new blocked status (true if blocked, false if unblocked).

3.7.2.13 markEdgesAsUnderRepaired()

```
void Graph::markEdgesAsUnderRepaired (
            const string & intersection1,
            const string & intersection2,
            bool isUnderRepaired )
```

Marks the specified Edges as under repair or not.

Parameters

intersection1	The name of the first intersection.
intersection2	The name of the second intersection.
isUnderRepaired	The new under-repair status (true if under repair, false if not).

3.7.2.14 removeEdge()

```
void Graph::removeEdge (
            const string & start,
            const string & end )
```

Removes a road (edge) between two intersections.

Parameters

start	The name of the starting intersection.
end	The name of the destination intersection.

3.7.2.15 removeVertex()

```
void Graph::removeVertex (
            const string & name )
```

Removes a vertex (intersection) from the graph.

Parameters

name	The name of the vertex to remove.

3.7.3 Member Data Documentation

3.7.3.1 headVertex

```
VertexNode* Graph::headVertex
```

Head of the linked list for vertices

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

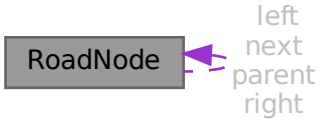
- graph.h
- graph.cpp

3.8 RoadNode Struct Reference

The node for the Hash Table containing a key-value pair and a pointer for chaining.

```
#include <RoadNode.h>
```

Collaboration diagram for RoadNode:



Public Member Functions

- RoadNode (char s='\0', char e='\0', int v=0)

  Constructor to initialize a RoadNode.

Public Attributes

- char **path** [2]
- int **carCount**
- RoadNode * **right**
- RoadNode * **left**
- RoadNode * **parent**
- RoadNode * **next**

3.8.1 Detailed Description

The node for the Hash Table containing a key-value pair and a pointer for chaining.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 RoadNode()

```
RoadNode::RoadNode (
            char s = '\0',
            char e = '\0',
            int v = 0 )  [inline]
```

Constructor to initialize a RoadNode.

Parameters

s	Start intersection
e	End intersection
v	Number of cars on the road

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

- RoadNode.h

3.9 RoadQueue Class Reference

A class to represent a queue specifically for managing RoadNode objects.

```
#include <roadQueue.h>
```

Public Member Functions

- RoadQueue ()
  Construct a new RoadQueue object.
- void enqueue (RoadNode *&newNode)
  Adds a new RoadNode to the end of the queue.
- RoadNode * dequeue ()
  Removes and returns the RoadNode at the front of the queue.
- void printQueue ()
  Prints the contents of the queue.
- bool isEmpty ()
  Checks if the queue is empty.

3.9.1 Detailed Description

A class to represent a queue specifically for managing RoadNode objects.

This class provides functionalities to create and manage a queue of RoadNode objects, which can be used for various purposes such as managing road segments in a traffic system.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 RoadQueue()

```
RoadQueue::RoadQueue ( )
```

Construct a new RoadQueue object.

Initializes the head and tail to nullptr, indicating an empty queue.

3.9.3 Member Function Documentation

3.9.3.1 dequeue()

```
RoadNode * RoadQueue::dequeue ( )
```

Removes and returns the RoadNode at the front of the queue.

This function removes the RoadNode at the head of the queue and returns it.

Returns

The RoadNode at the front of the queue.

3.9.3.2 enqueue()

```
void RoadQueue::enqueue (
            RoadNode *& newNode )
```

Adds a new RoadNode to the end of the queue.

This function inserts a new RoadNode at the tail of the queue.

Parameters

newNode	The RoadNode to be added to the queue.

3.9.3.3 isEmpty()

```
bool RoadQueue::isEmpty ( )
```

Checks if the queue is empty.

This function checks whether the queue is empty.

Returns

true if the queue is empty, false otherwise.

3.9.3.4 printQueue()

```
void RoadQueue::printQueue ( )
```

Prints the contents of the queue.

This function prints the current state of the queue for debugging and visualization purposes.

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

- roadQueue.h
- roadQueue.cpp

3.10 TrafficLightManagement Class Reference

Manages the traffic lights at each intersection.

```
#include <trafficLightManagement.h>
```

Public Member Functions

- TrafficLightManagement ()
  Constructor for TrafficLightManagement class.
- void makeTrafficSignals ()
  Assigns a traffic signal to each intersection.
- void updateTrafficSignals (CongestionMonitoring &ht)
  Updates the traffic signals based on the congestion monitoring data.
- void addSignal (TrafficSignal *signal)
  Adds a signal to the list of traffic signals.
- void **printGreenTimes** ()
  Prints the time the green state is to be maintained of the traffic signals.
- TrafficSignal * getSignal (std::string intersection)
  Returns the signal for a given intersection.

3.10.1 Detailed Description

Manages the traffic lights at each intersection.

The TrafficLightManagement class is responsible for managing the traffic lights at each intersection. It maintains a list of traffic signals and updates them based on the current traffic conditions.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 TrafficLightManagement()

```
TrafficLightManagement::TrafficLightManagement ( )
```

Constructor for TrafficLightManagement class.

Initializes the headSignal pointer to NULL.

3.10.3 Member Function Documentation

3.10.3.1 addSignal()

```
void TrafficLightManagement::addSignal (
            TrafficSignal * signal )
```

Adds a signal to the list of traffic signals.

Parameters

signal	The signal to be added.

3.10.3.2 getSignal()

```
TrafficSignal * TrafficLightManagement::getSignal (
            std::string intersection )
```

Returns the signal for a given intersection.

Parameters

intersection	The intersection for which the signal is to be returned.

Returns

TrafficSignal* The signal for the given intersection.

3.10.3.3 makeTrafficSignals()

```
void TrafficLightManagement::makeTrafficSignals ( )
```

Assigns a traffic signal to each intersection.

Loads the time duration for each signal from the csv file.

3.10.3.4 updateTrafficSignals()

```
void TrafficLightManagement::updateTrafficSignals (
            CongestionMonitoring & ht )
```

Updates the traffic signals based on the congestion monitoring data.

Parameters

congestionMonitoring	The congestion monitoring hash table.

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

- trafficLightManagement.h
- trafficLightsManagement.cpp

3.11 TrafficSignal Class Reference

A class to represent a traffic signal at an intersection.

```
#include <trafficSignal.h>
```

Collaboration diagram for TrafficSignal:

Public Member Functions

- TrafficSignal (std::string state="red", char intersectionId='-', int duration=60, int transitionTime=5)

  Initializes a TrafficSignal instance with the specified state ("red", "yellow", or "green"), Intersection ID, and duration. Defaults to "red" state, -1 Intersection ID, and 60 seconds duration.
- std::string getState ()

  Retrieves the current state of the traffic signal.

- void setState (std::string state)

  Updates the state of the traffic signal.
- void display ()

  Intended for displaying the traffic signal on a graphical user interface (GUI). This method is currently not implemented.
- void print ()

  Outputs the traffic signal's state, Intersection ID, and duration in a human-readable format.
- int getIntersectionId ()

  Retrieve the ID of the intersection for which the signal is.
- int getDuration ()

  Retrieve the duration for which a signal state is maintained.
- int getTransitionTime ()

  Retrieve the time the "yellow" state is maintained.

Public Attributes

- std::string state
- int transitionTime
- char intersectionId
- int duration
- TrafficSignal * **next**
- int **temp**

3.11.1 Detailed Description

A class to represent a traffic signal at an intersection.

This class models a traffic signal at an intersection, providing functionality to manage its state ("red", "yellow", or "green"), the associated Intersection ID, and the duration for which each state is maintained. It also offers methods for displaying and printing the signal's information.

The TrafficSignal class maintains the state of the signal ("red", "yellow", or "green"), the ID of the intersection it belongs to, and the duration for which the signal state is maintained.

Note

The display method is intended for future implementation to show the signal on a graphical interface.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 TrafficSignal()

```
TrafficSignal::TrafficSignal (
            std::string state = "red",
            char intersectionId = '-',
            int duration = 60,
            int transitionTime = 5 )
```

Initializes a TrafficSignal instance with the specified state ("red", "yellow", or "green"), Intersection ID, and duration. Defaults to "red" state, -1 Intersection ID, and 60 seconds duration.

Constructor to initialize the TrafficSignal object with the given state, intersectionId, and duration.

Parameters

state	string "red", "yellow" or "green". Default value is "red"
intersectionId	char ID of the intersection for which the signal is. Default value is '-'
duration	int duration a state is to be maintained in seconds. Default value is 60
transitionTime	int the time the "yellow" state is maintained. Default value is 5
state	The initial state of the signal ("red", "yellow", or "green"). Default value is "red".
intersectionId	The ID of the intersection for which the signal is. Default value is -1.
duration	The duration for which the signal state is maintained, in seconds. Default value is 60.
transitionTime	The time the "yellow" state is maintained, in seconds. Default value is 5.

3.11.3 Member Function Documentation

3.11.3.1 display()

```
void TrafficSignal::display ( )
```

Intended for displaying the traffic signal on a graphical user interface (GUI). This method is currently not implemented.

Displays the signal on the graphical interface.

Note

This method will be implemented in the future.

3.11.3.2 getDuration()

```
int TrafficSignal::getDuration ( )
```

Retrieve the duration for which a signal state is maintained.

Get the duration for which the signal state is maintained.

Returns

int Duration in seconds

The duration in seconds.

3.11.3.3 getIntersectionId()

```
int TrafficSignal::getIntersectionId ( )
```

Retrieve the ID of the intersection for which the signal is.

Get the Intersection ID of the signal.

Returns

int Intersection ID

The Intersection ID as an integer.

3.11.3.4 getState()

```
std::string TrafficSignal::getState ( )
```

Retrieves the current state of the traffic signal.

Get the current state of the signal.

Returns

string "red", "yellow" or "green"

The current state of the signal as a string.

3.11.3.5 getTransitionTime()

```
int TrafficSignal::getTransitionTime ( )
```

Retrieve the time the "yellow" state is maintained.

Get the time the "yellow" state is maintained.

Returns

int Transition time in seconds

The transition time in seconds.

3.11.3.6 print()

```
void TrafficSignal::print ( )
```

Outputs the traffic signal's state, Intersection ID, and duration in a human-readable format.

Print the signal state, intersectionId, and duration on the console.

Example output: "Signal State: red, Intersection ID: 1, Duration: 60s"

3.11.3.7 setState()

```
void TrafficSignal::setState (
            std::string state )
```

Updates the state of the traffic signal.

Set the state of the signal.

Parameters

state	A string representing the new state ("red", "yellow" or "green")
state	The new state of the signal ("red", "yellow", or "green").

3.11.4 Member Data Documentation

3.11.4.1 duration

```
int TrafficSignal::duration
```

The duration for which the signal state is maintained, in seconds.

3.11.4.2 intersectionId

```
int TrafficSignal::intersectionId
```

The ID of the intersection for which the signal is.

3.11.4.3 state

```
std::string TrafficSignal::state
```

The current state of the traffic signal ("red", "yellow", or "green").

3.11.4.4 transitionTime

```
int TrafficSignal::transitionTime
```

The time the "yellow" state is maintained, in seconds.

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

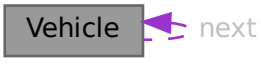
- trafficSignal.h
- trafficSignal.cpp

3.12 Vehicle Struct Reference

A structure representing a vehicle with details about its route and priority.

```
#include <vehicle.h>
```

Collaboration diagram for Vehicle:



Public Member Functions

- Vehicle (std::string vehicleID, std::string startIntersection, std::string endIntersection, std::string priorityLevel)

  Constructs a Vehicle with given attributes.
- void moveForward (std::string nextIntersectionId="")

  updates the currentIntersectionId of the vehicle
- void **printVehicle** ()

  prints the details of the vehicle
- void setPath (std::string path)

  sets the path of the vehicle
- void **printPath** ()

  prints the path of the vehicle

Public Attributes

- std::string **vehicleID**
- const std::string **startIntersection**
- const std::string **endIntersection**
- std::string **priorityLevel**
- std::string * **path**
- int **currentIntersectionInPath**
- int **pathLength**
- bool **presetPath**
- Vehicle * **next**

  Pointer to the next vehicle in a linked list.

3.12.1 Detailed Description

A structure representing a vehicle with details about its route and priority.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 Vehicle()

```
Vehicle::Vehicle (
            std::string vehicleID,
            std::string startIntersection,
            std::string endIntersection,
            std::string priorityLevel )
```

Constructs a Vehicle with given attributes.

Parameters

vehicleID	The unique identifier for the vehicle.
startIntersection	The starting intersection for the vehicle.
endIntersection	The ending intersection for the vehicle.
priorityLevel	The priority level of the vehicle.

3.12.3 Member Function Documentation

3.12.3.1 moveForward()

```
void Vehicle::moveForward (
            std::string nextIntersectionId = "" )
```

updates the currentIntersectionId of the vehicle

Parameters

next↩ IntersectionId	The id of the next intersection the vehicle will move to if the nextIntersectionId is not provided, the vehicle will move to the next intersection in the path only if it is preset

3.12.3.2 setPath()

```
void Vehicle::setPath (
            std::string path )
```

sets the path of the vehicle

Parameters

path	the path the vehicle will take

Note

if the initial intersection of the vehicle is "A" the path must be "ABC.." but it cannot be "BC.."

the path must be preset before the vehicle starts moving

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

- vehicle.h
- vehicle.cpp

3.13 Vehicles Class Reference

A class to manage a linked list of vehicles.

```
#include <vehicles.h>
```

Public Member Functions

- **Vehicles** ()

  Constructor for Vehicles class.
- ∼**Vehicles** ()

  Destructor for Vehicles class.
- void insertAtHead (std::string VehicleID, std::string startIntersection, std::string endIntersection, std::string priorityLevel)

  Inserts a vehicle at the head of the linked list.
- void enqueue (std::string VehicleID, std::string startIntersection, std::string endIntersection, std::string priorityLevel)

  Enqueues a vehicle at the end of the linked list.
- bool insertAfterPosition (int position, std::string VehicleID, std::string startIntersection, std::string end↩ Intersection, std::string priorityLevel)

  Inserts a vehicle after a specific position in the linked list.
- bool insertAfterID (std::string ID, std::string VehicleID, std::string startIntersection, std::string endIntersection, std::string priorityLevel)

  Inserts a vehicle after a specific vehicle ID in the linked list.
- void **deleteAtStart** ()

  Deletes the vehicle at the start of the linked list.
- bool deleteAtEnd ()

  Deletes the vehicle at the end of the linked list.
- bool deleteAtIndex (int position)

  Deletes the vehicle at a specific index in the linked list.
- bool deleteAtID (std::string ID)

  Deletes the vehicle with a specific ID in the linked list.
- bool isEmpty ()

  Checks if the linked list is empty.
- void **printVehicles** ()

  Prints the details of all vehicles in the linked list.
- int findIDInVehicles (std::string vehicleID)

  Finds the position of a vehicle with a specific ID in the linked list.
- void **loadAndReadCSVs** ()

  Loads and reads vehicle data from CSV files.
- Vehicle ∗& getHead ()

  Gets the head of the linked list.
- void addPaths (GPS &gps)

  Adds paths to the vehicles using GPS data.

3.13.1 Detailed Description

A class to manage a linked list of vehicles.

3.13.2 Member Function Documentation

3.13.2.1 addPaths()

```
void Vehicles::addPaths (
            GPS & gps )
```

Adds paths to the vehicles using GPS data.

Parameters

gps	The GPS object containing path data.

3.13.2.2 deleteAtEnd()

```
bool Vehicles::deleteAtEnd ( )
```

Deletes the vehicle at the end of the linked list.

Returns

True if the deletion was successful, false otherwise.

3.13.2.3 deleteAtID()

```
bool Vehicles::deleteAtID (
            std::string ID )
```

Deletes the vehicle with a specific ID in the linked list.

Parameters

ID	The ID of the vehicle to be deleted.

Returns

True if the deletion was successful, false otherwise.

3.13.2.4 deleteAtIndex()

```
bool Vehicles::deleteAtIndex (
            int position )
```

Deletes the vehicle at a specific index in the linked list.

Parameters

position	The index of the vehicle to be deleted.

Returns

True if the deletion was successful, false otherwise.

3.13.2.5 enqueue()

```
void Vehicles::enqueue (
            std::string VehicleID,
            std::string startIntersection,
            std::string endIntersection,
            std::string priorityLevel )
```

Enqueues a vehicle at the end of the linked list.

Parameters

VehicleID	The ID of the vehicle.
startIntersection	The starting intersection of the vehicle.
endIntersection	The ending intersection of the vehicle.
priorityLevel	The priority level of the vehicle.

3.13.2.6 findIDInVehicles()

```
int Vehicles::findIDInVehicles (
            std::string vehicleID )
```

Finds the position of a vehicle with a specific ID in the linked list.

Parameters

vehicleID	The ID of the vehicle to be found.

Returns

The position of the vehicle in the linked list, or -1 if not found.

3.13.2.7 getHead()

```
Vehicle *& Vehicles::getHead ( )
```

Gets the head of the linked list.

Returns

A reference to the pointer to the head of the linked list.

3.13.2.8 insertAfterID()

```
bool Vehicles::insertAfterID (
            std::string ID,
            std::string VehicleID,
            std::string startIntersection,
            std::string endIntersection,
            std::string priorityLevel )
```

Inserts a vehicle after a specific vehicle ID in the linked list.

Parameters

ID	The ID of the vehicle after which the new vehicle will be inserted.
VehicleID	The ID of the new vehicle.
startIntersection	The starting intersection of the new vehicle.
endIntersection	The ending intersection of the new vehicle.
priorityLevel	The priority level of the new vehicle.

Returns

True if the insertion was successful, false otherwise.

3.13.2.9 insertAfterPosition()

```
bool Vehicles::insertAfterPosition (
            int position,
            std::string VehicleID,
            std::string startIntersection,
            std::string endIntersection,
            std::string priorityLevel )
```

Inserts a vehicle after a specific position in the linked list.

Parameters

position	The position after which the vehicle will be inserted.
VehicleID	The ID of the vehicle.
startIntersection	The starting intersection of the vehicle.
endIntersection	The ending intersection of the vehicle.
priorityLevel	The priority level of the vehicle.

Returns

True if the insertion was successful, false otherwise.

3.13.2.10 insertAtHead()

```
void Vehicles::insertAtHead (
            std::string VehicleID,
            std::string startIntersection,
            std::string endIntersection,
            std::string priorityLevel )
```

Inserts a vehicle at the head of the linked list.

Parameters

VehicleID	The ID of the vehicle.
startIntersection	The starting intersection of the vehicle.
endIntersection	The ending intersection of the vehicle.
priorityLevel	The priority level of the vehicle.

3.13.2.11 isEmpty()

```
bool Vehicles::isEmpty ( )
```

Checks if the linked list is empty.

Returns

True if the linked list is empty, false otherwise.

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

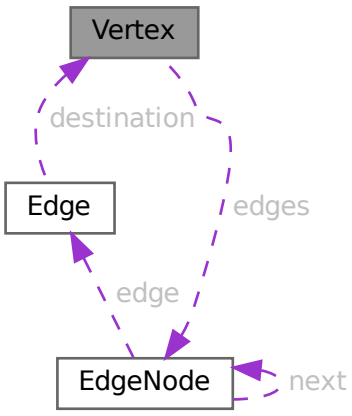
- vehicles.h
- vehicles.cpp

3.14 Vertex Class Reference

Represents an intersection in the road network.

```
#include <graph.h>
```

Collaboration diagram for Vertex:



Public Member Functions

- Vertex (const std::string &name)

  Constructs a Vertex object.

Public Attributes

- std::string name
- struct EdgeNode * edges

3.14.1 Detailed Description

Represents an intersection in the road network.

The Vertex class stores the details of an intersection (such as its name and blockage status) and maintains a list of outgoing edges representing the roads connecting the intersection to others.

3.14.2 Constructor & Destructor Documentation

3.14.2.1 Vertex()

```
Vertex::Vertex (
            const std::string & name )
```

Constructs a Vertex object.

Parameters

name	The name of the intersection.

3.14.3 Member Data Documentation

3.14.3.1 edges

```
struct EdgeNode* Vertex::edges
```

Linked list of edges (adjacency list)

3.14.3.2 name

```
std::string Vertex::name
```

Intersection name

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

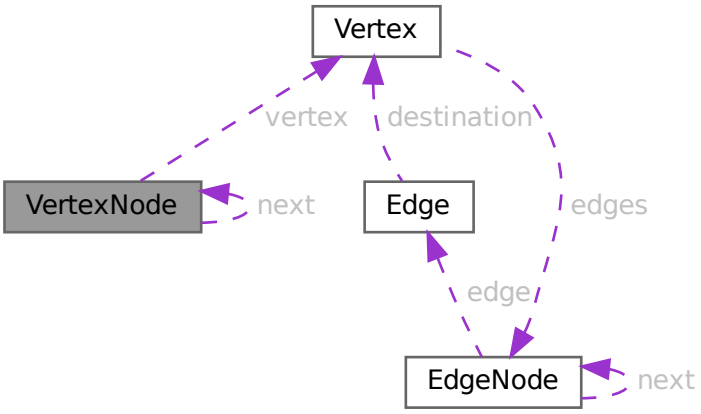
- graph.h
- graph.cpp

3.15 VertexNode Class Reference

Represents a node in the adjacency list for vertices.

`#include <graph.h>`

Collaboration diagram for VertexNode:



Public Member Functions

- VertexNode (Vertex *vertex)

  Constructs a VertexNode object.

Public Attributes

- Vertex * vertex
- VertexNode * next

3.15.1 Detailed Description

Represents a node in the adjacency list for vertices.

A VertexNode is used to store a Vertex in the adjacency list of the graph. It also maintains a pointer to the next VertexNode, allowing multiple vertices to be linked together in the graph.

3.15.2 Constructor & Destructor Documentation

3.15.2.1 VertexNode()

```
VertexNode::VertexNode (
            Vertex * vertex )
```

Constructs a VertexNode object.

Parameters

vertex	The vertex that this node will represent.

3.15.3 Member Data Documentation

3.15.3.1 next

```
VertexNode* VertexNode::next
```

Pointer to the next vertex

3.15.3.2 vertex

```
Vertex* VertexNode::vertex
```

The vertex

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

- graph.h
- graph.cpp

3.16 Visualizer Class Reference

A class to handle the visualization of the traffic management system.

```
#include <visualizer.h>
```

Public Member Functions

- Visualizer ()

  Constructor for the Visualizer class.
- void drawSimulation (Graph &graph, Vehicles &vehicles, TrafficLightManagement &traffic, CongestionMonitoring &ht, Accident_roads &accidentManager)

  Draws the entire simulation graph.
- void drawVehicles (Vehicles &vehicles, const std::string &intersection, const sf::Vector2f &position, sf::↩ RenderWindow &window)

  Draws vehicles at the specified intersection.
- float getElapsedTimeInSeconds ()

  Gets the elapsed time in seconds since the last clock reset.
- sf::Color choseColor (EdgeNode *edge, Vertex *vertex, TrafficLightManagement &traffic, CongestionMonitoring &ht, Accident_roads &accidentManager)

  Chooses a color for the edge and vertex based on traffic conditions.

3.16.1 Detailed Description

A class to handle the visualization of the traffic management system.

This class is responsible for rendering the simulation of the traffic management system, including roads and vehicles, using the SFML library.

3.16.2 Constructor & Destructor Documentation

3.16.2.1 Visualizer()

```
Visualizer::Visualizer ( )
```

Constructor for the Visualizer class.

Initializes the render window, loads textures, and sets up sprites.

3.16.3 Member Function Documentation

3.16.3.1 choseColor()

```
sf::Color Visualizer::choseColor (
            EdgeNode * edge,
            Vertex * vertex,
            TrafficLightManagement & traffic,
            CongestionMonitoring & ht,
            Accident_roads & accidentManager )
```

Chooses a color for the edge and vertex based on traffic conditions.

Parameters

edge	The edge node representing the road segment.
vertex	The vertex representing the intersection.
traffic	The traffic light management system.
ht	The congestion monitoring system.
accidentManager	The accident management system.

Returns

The chosen color.

3.16.3.2 drawSimulation()

```
void Visualizer::drawSimulation (
            Graph & graph,
            Vehicles & vehicles,
```

```
            TrafficLightManagement & traffic,
            CongestionMonitoring & ht,
            Accident_roads & accidentManager )
```

Draws the entire simulation graph.

Parameters

graph	The graph representing the traffic network.
vehicles	The collection of vehicles to be drawn.
traffic	The traffic light management system.
ht	The congestion monitoring system.
accidentManager	The accident management system.

3.16.3.3 drawVehicles()

```
void Visualizer::drawVehicles (
            Vehicles & vehicles,
            const std::string & intersection,
            const sf::Vector2f & position,
            sf::RenderWindow & window )
```

Draws vehicles at the specified intersection.

Parameters

vehicles	The collection of vehicles to be drawn.
intersection	The name of the intersection where vehicles are to be drawn.
position	The position where the vehicles should be drawn.
window	The render window where the vehicles will be drawn.

3.16.3.4 getElapsedTimeInSeconds()

```
float Visualizer::getElapsedTimeInSeconds ( )
```

Gets the elapsed time in seconds since the last clock reset.

Returns

The elapsed time in seconds.

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

- visualizer.h
- visualizer.cpp

Chapter 4

File Documentation

4.1 accidents.h

```
00001 #ifndef ACCIDENTS_H
00002 #define ACCIDENTS_H
00003 #include <string>
00004
00005 // Forward declaration of Graph class
00006 class Graph;
00007
00015 class Accident_roads {
00016 private:
00024     struct AccidentNode {
00025         std::string intersection1;
00026         std::string intersection2;
00027         bool isBlocked;
00028         AccidentNode* next;
00037         AccidentNode(const std::string& i1, const std::string& i2, bool blocked)
00038             : intersection1(i1), intersection2(i2), isBlocked(blocked), next(nullptr) {}
00039     };
00040
00041     AccidentNode* head;
00042     AccidentNode* underRepairHead;
00044 public:
00045     void blockRoad(const std::string& start, const std::string& end, Graph& cityGraph);
00051     Accident_roads();
00052
00058     ~Accident_roads();
00059
00068     void loadRoadData(Graph& graph);
00069
00081     void markIntersectionsAsBlocked(Graph& graph, const std::string& intersection1, const std::string&
      intersection2, bool isBlocked);
00082
00090     void displayBlockedIntersections(Graph& graph);
00091
00099     void displayUnderRepairIntersections(Graph& graph);
00100
00106     void displayBlockedRoads();
00107
00113     void displayUnderRepairRoads();
00114
00115
00116 };
00117
00118 #endif // ACCIDENTS_H
```

4.2 congestionMaxHeap.h

```
00001 #ifndef CONGESTIONMAXHEAP_H
00002 #define CONGESTIONMAXHEAP_H
00003
00004 # include "RoadNode.h"
00005 # include "congestionMonitoring.h"
00006
00015 class CongestionMaxHeap{
00016     private:
```

```
00017         RoadNode* root; //< Pointer to the root of the heap
00018
00026         void inorder(RoadNode* root);
00027     public:
00034         CongestionMaxHeap();
00041         void makeHeap(RoadNode* hashTableArray, int size);
00049         void insert(RoadNode* newNode);
00055         void heapifyUp(RoadNode*& newNode);
00062         void printHeap();
00068         RoadNode* mostCongested();
00074         void inorder();
00080         void heapifyDown(RoadNode* node);
00081
00082 };
00083 #endif  // CONGESTIONMAXHEAP_H
```

4.3 congestionMonitoring.h

```
00001 # ifndef CONGESTION_MONITORING_H
00002 # define CONGESTION_MONITORING_H
00003 # include "vehicles.h"
00004 # include "vehicle.h"
00005 # include "graph.h"
00006 # include "RoadNode.h"
00007 const int HASH_TABLE_SIZE = 100;
00008
00016 class CongestionMonitoring {
00017
00018     private:
00019         const int hashTableSize; //< Hash table array size
00020
00021         // private functions ------------------------------------------------------------
00032         void addToTable(int index, char start, char end, int right = 0);
00033
00043         int hashFunction(char start, char end);
00053         char getStart(int index, int right);
00054
00064         char getEnd(int index, int right);
00065         // ------------------------------------------------------------------------------
00066
00067     public:
00068         RoadNode hashTable[HASH_TABLE_SIZE]; //< RoadNode array
00069
00077         CongestionMonitoring(Vehicle* vehiclesHead);
00085         void makeHashTable(Vehicle* vehiclesHead);
00094         void updateHashTable(Vehicle* prevPos, Vehicle* currentPos);
00095
00100         void printHashTable();
00106         void deleteTable();
00107
00117         RoadNode* findRoadNode(char start, char end);
00118
00127         int getTravelTime(char start, char end, int prevTime);
00138         int getTravelTime(char start, char end, Graph& cityGraph);
00144         int numberOfCongestionEvents();
00145
00146
00147 };
00148
00149
00150 #endif // CONGESTION_MONITORING_H
```

4.4 graph.h

```
00001 #ifndef GRAPH_H
00002 #define GRAPH_H
00003
00004 #include <string>
00005 #include <iostream>
00006 using namespace std;
00007
00008 // Forward declaration of the Accident_roads class
00009 class Accident_roads;
00010
00018 class Vertex {
00019 public:
00020     std::string name;
00022     struct EdgeNode* edges;
00028     Vertex(const std::string& name);
```

```
00029
00030 };
00031
00039 class Edge {
00040 public:
00041     Vertex* destination;
00042     int travelTime;
00043     bool blocked;
00044     bool underRepaired;
00050     bool isBlocked() const;
00051
00056     bool isUnderRepaired() const;
00057
00062     void setBlocked(bool status);
00063
00068     void setUnderRepaired(bool status);
00069
00075     Edge(Vertex* destination, int travelTime);
00076 };
00077
00085 class EdgeNode {
00086 public:
00087     Edge* edge;
00088     EdgeNode* next;
00094     EdgeNode(Edge* edge);
00095 };
00096
00104 class VertexNode {
00105 public:
00106     Vertex* vertex;
00107     VertexNode* next;
00113     VertexNode(Vertex* vertex);
00114 };
00115
00124 class Graph {
00125 public:
00126     VertexNode* headVertex;
00131     Graph();
00132
00136     ~Graph();
00137
00142     void addVertex(const std::string& name);
00143
00149     Vertex* findVertex(const std::string& name);
00150
00155     void removeVertex(const string& name);
00156
00163     void addEdge(const std::string& start, const std::string& end, int travelTime);
00164
00170     void removeEdge(const string& start, const string& end);
00171
00176     void loadRoadData(const std::string& filename = "road_network.csv");
00177
00181     void displayRoadStatuses();
00182
00189     void markEdgeAsBlocked(const string& intersection1, const string& intersection2, bool isBlocked);
00190
00197     void markEdgesAsUnderRepaired(const string& intersection1, const string& intersection2, bool
    isUnderRepaired);
00198
00202     void displayBlockedEdges();
00203
00209     void addEdgeToVertex(Vertex* vertex, Edge* edge);
00210
00214     void printAdjacencyList();
00215
00221     bool isBlocked(const std::string& nodeName1, const std::string& nodeName2);
00222
00229     void getNeighbors(const std::string& nodeName, std::string* neighbors, int& count);
00230
00237     int getEdgeWeight(const std::string& start, const std::string& end);
00238
00244     void getVertices(std::string* vertices, int& count);
00245
00250     int getVertexCount();
00251
00257     void getAllEdges(std::string edges[][3], int& count);
00258 };
00259
00260 #endif // GRAPH_H
```

4.5 RoadNode.h

```
00001 #ifndef ROADNODE_H
```

```
00002 #define ROADNODE_H
00003
00007 struct RoadNode {
00008     char path[2]; //< Start and end Intersection
00009     int carCount; //< The number of cars on the road
00010     RoadNode* right; //< Pointer for chaining in case of collisions in hashtables. Points to right
      child in minheap
00011     RoadNode* left; //< Pointer for the left child in minheap
00012     RoadNode* parent; //< Pointer to the parent node in minheap
00013     RoadNode* next; //< Pointer to the next node in the queue
00020     RoadNode(char s = '\0', char e = '\0', int v = 0) {
00021         path[0] = s;
00022         path[1] = e;
00023         carCount = v;
00024         right = nullptr;
00025         left = nullptr;
00026         parent = nullptr;
00027         next = nullptr;
00028     }
00029 };
00030
00031 #endif
```

4.6 roadQueue.h

```
00001 #ifndef ROADQUEUE_H
00002 #define ROADQUEUE_H
00003 # include "RoadNode.h"
00011 class RoadQueue{
00012       private:
00013             RoadNode* head;
00014             RoadNode* tail;
00015     public:
00021         RoadQueue();
00022
00030         void enqueue(RoadNode*& newNode);
00031
00039         RoadNode* dequeue();
00040
00046         void printQueue();
00047
00055         bool isEmpty();
00056 };
00057
00058 #endif
```

4.7 Route.h

```
00001 #ifndef GPS_H
00002 #define GPS_H
00003
00004 #include <iostream>
00005 #include <string>
00006 #include <cstring>
00007 #include "graph.h"  // Assuming you have a Graph class for managing vertices and edges
00008 using namespace std;
00009
00017 class GPS {
00018 private:
00019     Graph* graph;
00020     static const int MAX_VERTICES = 250;
00021     std::string vertexNames[MAX_VERTICES];
00022
00023     int vertexCount = 0;
00024
00034     int getVertexIndex(const std::string& name);
00035
00052     void findAllPathsDFS(Vertex* start, Vertex* end,
00053                          std::string path[], int pathIndex,
00054                          std::string allPaths[][MAX_VERTICES],
00055                          int& allPathsCount, bool visited[],
00056                          int totalWeight[], int& totalWeightCount);
00057
00058 public:
00066     GPS(Graph* graph);
00067
00077     void printAllPaths(const std::string& startName, const std::string& endName);
00078
00090     string rerouteEmergencyVehicle(const string& startName, const string& endName);
```

```
00091
00101     string getPathAsString(const string& startName, const string& endName);
00102
00118     void findAllOptimalPaths(Vertex* start, Vertex* end,
00119                              string path[], int pathIndex,
00120                              string allPaths[][MAX_VERTICES],
00121                              int& allPathsCount, bool visited[],
00122                              int totalWeight[], int& totalWeightCount);
00123
00132     void printAllPathsDijkstra(const string& startName, const string& endName);
00133
00143     int heuristic(const Vertex* a, const Vertex* b);
00144 };
00145
00146 #endif
```

4.8 trafficLightManagement.h

```
00001 #ifndef TRAFFIC_LIGHT_MANAGEMENT_H
00002 #define TRAFFIC_LIGHT_MANAGEMENT_H
00003 // # include "visualizer.h"
00004 # include "trafficSignal.h"
00005 # include "congestionMonitoring.h"
00006 # include <string>
00015 const std::string SIGNALS_FILE = "dataset/traffic_signals.csv";
00016 class TrafficLightManagement {
00017       private:
00018             TrafficSignal* headSignal;
00019       public:
00025             TrafficLightManagement();
00033             void makeTrafficSignals();
00039             void updateTrafficSignals(CongestionMonitoring& ht);
00045             void addSignal(TrafficSignal* signal);
00050             void printGreenTimes();
00057             TrafficSignal* getSignal(std::string intersection);
00066             // void manageTrafficLights(char mostCongestedIntersection, Visualizer* vs);
00067 };
00068
00069 #endif
```

4.9 trafficSignal.h

```
00001
00002 #ifndef TRAFFIC_SIGNAL_H
00003 #define TRAFFIC_SIGNAL_H
00004 // #include "visualizer.h"
00005 #include <string>
00006 class Visualizer;
00070 class TrafficSignal {
00071       public:
00072             std::string state; //<  "red", "yellow", "green"
00073             int transitionTime; //< the time the "yellow" state is maintained
00074
00075             char intersectionId; //< ID of the intersection for which the signal is
00076             int duration; //< duration a state is to be maintained in seconds
00077             TrafficSignal* next; //< pointer to the next signal in the list
00078             int temp; //< temporary variable to store the duration the current state has been
      maintained
00079
00080
00081
00092             TrafficSignal(std::string state = "red", char intersectionId = '-', int duration = 60, int
      transitionTime = 5);
00093
00098             std::string getState();
00099
00104             void setState(std::string state);
00105
00109             void display();
00110
00117             void print();
00118
00124             int getIntersectionId();
00125
00130             int getDuration();
00131
00136             int getTransitionTime();
00137
00143             // void advanceState(Visualizer* visualizer);
```

```
00144
00148             // void turnGreen(Visualizer* visualizer);
00149 };
00150
00151 #endif
```

4.10 vehicle.h

```
00001 #ifndef VEHICLE_H
00002 #define VEHICLE_H
00003
00004 #include<string>
00005
00010 struct Vehicle {
00011     std::string vehicleID; //<The unique identifier for the vehicle
00012     const std::string startIntersection; //<The starting intersection for the vehicle's route never to
      be changed
00013     const std::string endIntersection; //<The ending intersection for the vehicle's route never to be
      changed
00014     std::string priorityLevel; //<The priority level of the vehicle (e.g., high, low)
00015     std::string* path; //<The path the vehicle will take to reach its destination.
00016     int currentIntersectionInPath; //< an index in the path array that represents the current
      intersection the vehicle is at. the next intersection is at currentIntersectionInPath + 1
00017     int pathLength; //<The length of the path array
00018     bool presetPath; //<A boolean to check if the path is preset or not
00022     Vehicle *next;
00023
00024
00032     Vehicle(std::string vehicleID, std::string startIntersection, std::string endIntersection,
      std::string priorityLevel);
00039     void moveForward(std::string nextIntersectionId = "");
00044     void printVehicle();
00055     void setPath(std::string path);
00060     void printPath();
00061
00062 };
00063
00064 #endif
```

4.11 vehicles.h

```
00001 #ifndef VEHICLES_H
00002 #define VEHICLES_H
00003
00004 #include <string> // Ensure string header is included
00005 #include<limits>
00006 #include "vehicle.h"
00007 #include "Route.h"
00008
00009
00010 // All functions in this class should use std::string as parameter types.
00015 class Vehicles {
00016 private:
00017     Vehicle* head; // Pointer to the head of the linked list
00018
00019 public:
00023 Vehicles();
00024
00028 ~Vehicles();
00029
00037 void insertAtHead(std::string VehicleID, std::string startIntersection, std::string endIntersection,
      std::string priorityLevel);
00038
00046 void enqueue(std::string VehicleID, std::string startIntersection, std::string endIntersection,
      std::string priorityLevel);
00047
00057 bool insertAfterPosition(int position, std::string VehicleID, std::string startIntersection,
      std::string endIntersection, std::string priorityLevel);
00058
00068 bool insertAfterID(std::string ID, std::string VehicleID, std::string startIntersection, std::string
      endIntersection, std::string priorityLevel);
00069
00073 void deleteAtStart();
00074
00079 bool deleteAtEnd();
00080
00086 bool deleteAtIndex(int position);
00087
00093 bool deleteAtID(std::string ID);
```

```
00094
00099 bool isEmpty();
00100
00104 void printVehicles();
00105
00111 int findIDInVehicles(std::string vehicleID);
00112
00116 void loadAndReadCSVs();
00117
00122 Vehicle*& getHead();
00123
00128 void addPaths(GPS& gps);
00129
00130
00131 };
00132
00133 #endif
```

4.12 visualizer.h

```
00001 #ifndef VISUALIZER_H
00002 #define VISUALIZER_H
00003
00004 #include <SFML/Graphics.hpp>
00005 #include <string>
00006 #include <map>
00007 #include "graph.h"
00008 #include "vehicle.h"
00009 #include "vehicles.h"
00010 #include "trafficSignal.h"
00011 #include "accidents.h"
00012 #include "trafficLightManagement.h"
00013 #include "congestionMonitoring.h"
00014
00015 class Visualizer {
00016 private:
00017     sf::RenderWindow window;
00018     sf::Sprite roadSprite;
00019     sf::Sprite vehicleSprite;
00020     sf::Font font;
00021     sf::Texture roadTexture;
00022     sf::Texture vehicleTexture;
00023     sf::Clock clock; //<Measures elapsed time
00024
00025 public:
00039 Visualizer();
00040
00050 void drawSimulation(Graph &graph, Vehicles &vehicles, TrafficLightManagement &traffic,
      CongestionMonitoring &ht, Accident_roads &accidentManager);
00051
00060 void drawVehicles(Vehicles &vehicles, const std::string &intersection, const sf::Vector2f &position,
      sf::RenderWindow &window);
00061
00067 float getElapsedTimeInSeconds();
00068
00079 sf::Color choseColor(EdgeNode *edge, Vertex *vertex, TrafficLightManagement &traffic,
      CongestionMonitoring &ht, Accident_roads &accidentManager);
00080
00081
00082 };
00083 #endif // VISUALIZER_H
```

Index