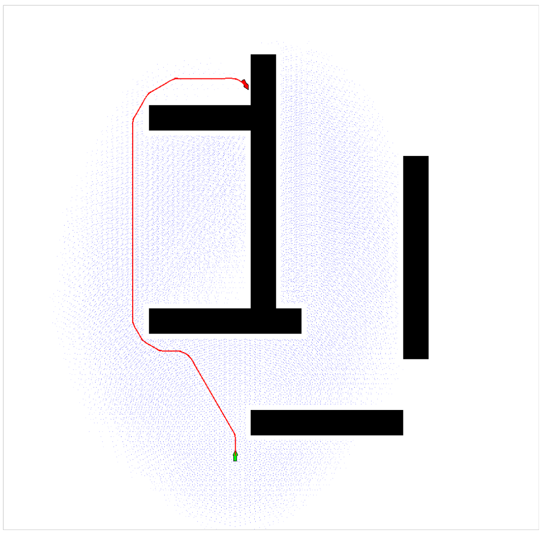
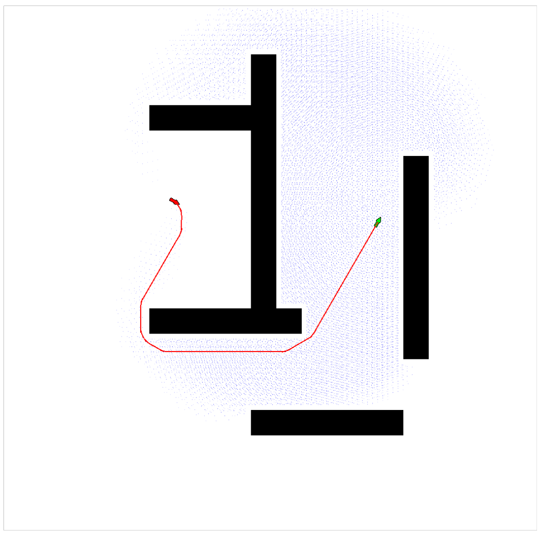
# City Based CBS Documentation

https://github.com/Faywyn/city-CBS-Astar





2024-2025

# 1 Class Index

## 1.1 Class List

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

# 2 File Index

## 2.1 File List

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

# 3 Class Documentation

## 3.1 _aStarConflict Struct Reference

A conflict for the A∗ algorithm.

```
#include <aStar.h>
```

### Public Attributes

- CityGraph::point **point**

*The point in the graph.*

- int **time**

  *The time of the conflict.*

- int **car**

  *The car that caused the conflict.*

### 3.1.1 Detailed Description

A conflict for the A∗ algorithm.
This struct represents a conflict for the A∗ algorithm. It contains the point in the graph, the time of the conflict and the car that caused the conflict.
Definition at line 41 of file aStar.h.
The documentation for this struct was generated from the following file:

- aStar.h

## 3.2 _aStarNode Struct Reference

A node for the A∗ algorithm.
`#include <aStar.h>`

#### Public Attributes

- CityGraph::point **point**

  *The point in the graph.*

- double **speed**

  *The speed of the car.*

- std::pair< CityGraph::point, CityGraph::neighbor > **arcFrom**

  *The arc from which the node was reached.*

### 3.2.1 Detailed Description

A node for the A∗ algorithm.
This struct represents a node for the A∗ algorithm. It contains the point in the graph, the speed of the car and the arc from which the node was reached.
Definition at line 20 of file aStar.h.
The documentation for this struct was generated from the following file:

- aStar.h

## 3.3 _cityGraphNeighbor Struct Reference

A neighbor of a point in the city graph.
`#include <cityGraph.h>`

#### Public Attributes

- _cityGraphPoint **point**

  *The neighbor point.*

- double **maxSpeed**

  *The maximum speed to reach the neighbor point.*

- double **turningRadius**

  *The turning radius to reach the neighbor point.*

- double **distance**

  *The distance to reach the neighbor point.*

- bool **isRightWay**

  *If it is the right way.*

#### 3.3.1 Detailed Description

A neighbor of a point in the city graph.
This struct represents a neighbor of a point in the city graph. It contains the neighbor point, the maximum speed to reach it, the turning radius to reach it, the distance to reach it and if it is the right way.
Definition at line 43 of file cityGraph.h.
The documentation for this struct was generated from the following file:

- cityGraph.h

### 3.4 _cityGraphPoint Struct Reference

A point in the city graph.
`#include <cityGraph.h>`

#### Public Attributes

- sf::Vector2f **position**

  *The position of the point.*
- double **angle**

  *The angle of the point.*

#### 3.4.1 Detailed Description

A point in the city graph.
This struct represents a point in the city graph. It contains the position and the angle of the point.
Definition at line 20 of file cityGraph.h.
The documentation for this struct was generated from the following file:

- cityGraph.h

### 3.5 _cityMapBuilding Struct Reference

A building in the city map.
`#include <cityMap.h>`

#### Public Attributes

- std::vector< sf::Vector2f > **points**

  *The points of the building.*

#### 3.5.1 Detailed Description

A building in the city map.
Definition at line 34 of file cityMap.h.
The documentation for this struct was generated from the following file:

- cityMap.h

### 3.6 _cityMapGreenArea Struct Reference

A green area in the city map.
`#include <cityMap.h>`

#### Public Attributes

- std::vector< sf::Vector2f > **points**

  *The points of the green area.*
- int **type**

  *The type of the green area.*

### 3.6.1 Detailed Description

A green area in the city map.
Definition at line 42 of file cityMap.h.
The documentation for this struct was generated from the following file:

- cityMap.h

## 3.7 _cityMapIntersection Struct Reference

An intersection in the city map.
`#include <cityMap.h>`

### Public Attributes

- int **id**

  *The id of the intersection.*
- sf::Vector2f **center**

  *The center of the intersection.*
- double **radius**

  *The radius of the intersection.*
- std::vector< std::pair< int, int > > **roadSegmentIds**

  *The ids of the road segments (roadId, segmentId). The segments are the same for both directions of the road.*

### 3.7.1 Detailed Description

An intersection in the city map.
Definition at line 59 of file cityMap.h.
The documentation for this struct was generated from the following file:

- cityMap.h

## 3.8 _cityMapRoad Struct Reference

A road in the city map.
`#include <cityMap.h>`

### Public Attributes

- int **id**

  *The id of the road.*
- std::vector< _cityMapSegment > **segments**

  *The segments of the road.*
- double **width**

  *The width of the road.*
- int **numLanes**

  *The number of lanes of the road.*

### 3.8.1 Detailed Description

A road in the city map.
Definition at line 23 of file cityMap.h.
The documentation for this struct was generated from the following file:

- cityMap.h

## 3.9 _cityMapSegment Struct Reference

A segment in the city map.

`#include <cityMap.h>`

### Public Attributes

- sf::Vector2f **p1**

  *The first point of the segment.*
- sf::Vector2f **p2**

  *The second point of the segment.*
- sf::Vector2f **p1_offset**

  *The offset of the first point, used for the intersection.*
- sf::Vector2f **p2_offset**

  *The offset of the second point, used for the intersection.*
- double **angle**

  *The angle of the segment.*

### 3.9.1 Detailed Description

A segment in the city map.
Definition at line 11 of file cityMap.h.
The documentation for this struct was generated from the following file:

- cityMap.h

## 3.10 _cityMapWaterArea Struct Reference

A water area in the city map.

`#include <cityMap.h>`

### Public Attributes

- std::vector$<$ sf::Vector2f $>$ **points**

  *The points of the water area.*

### 3.10.1 Detailed Description

A water area in the city map.
Definition at line 51 of file cityMap.h.
The documentation for this struct was generated from the following file:

- cityMap.h

## 3.11 _data Struct Reference

Data structure.

`#include <dataManager.h>`

### 3.11.1 Detailed Description

Data structure.
This struct represents the data structure.
Definition at line 18 of file dataManager.h.
The documentation for this struct was generated from the following file:

- dataManager.h

## 3.12 _managerCBSNode Struct Reference

A node for the CBS algorithm.
`#include <manager.h>`

### Public Attributes

- std::vector< std::vector< sf::Vector2f > > **paths**

  *The paths for all agents.*
- ConstraintController **constraints**

  *The constraints for all agents.*
- std::vector< double > **costs**

  *The individual path costs.*
- double **cost**

  *The total cost.*
- int **depth**

  *The depth in the CBS tree.*
- bool **hasResolved**

  *If the node has resolved conflicts.*

### 3.12.1 Detailed Description

A node for the CBS algorithm.
This struct represents a node for the CBS algorithm. It contains the paths for all agents, the constraints for all agents, the individual path costs, the total cost, the depth in the CBS tree and if the node has resolved conflicts.
Definition at line 24 of file manager.h.
The documentation for this struct was generated from the following file:

- manager.h

## 3.13 AStar Class Reference

A∗ algorithm.
`#include <aStar.h>`

### Public Member Functions

- AStar (CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph)

  *Constructor.*
- std::vector< node > findPath ()

  *Find the path.*

### 3.13.1 Detailed Description

A∗ algorithm.
This class represents the A∗ algorithm. It is used to find the shortest path between two points in a graph.
Definition at line 74 of file aStar.h.

### 3.13.2 Constructor & Destructor Documentation

#### AStar()

```
AStar::AStar (
            CityGraph::point start,
            CityGraph::point end,
            const CityGraph & cityGraph)
```

Constructor.

**Parameters**

| | |
|---|---|
| *start* | The start point |
| *end* | The end point |
| *cityGraph* | The graph |

Definition at line 21 of file aStar.cpp.

### 3.13.3 Member Function Documentation

#### findPath()

```
std::vector< node > AStar::findPath ()  [inline]
```

Find the path.

**Returns**

The path

Definition at line 91 of file aStar.h.

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

- aStar.h
- aStar.cpp

## 3.14 Car Class Reference

A car in the city.

`#include <car.h>`

### Public Member Functions

- **Car** ()

  *Constructor.*
- void assignStartEnd (CityGraph::point start, CityGraph::point end)

  *Assign the start and end points.*
- void chooseRandomStartEndPath (CityGraph &graph, CityMap &cityMap)

  *Choose a random start and end point in the graph.*
- void assignPath (std::vector< AStar::node > path)

  *Assign a path to the car.*
- void assignExistingPath (std::vector< sf::Vector2f > path)

  *Assign an existing path to the car.*
- void **move** ()

  *Move the car, move to the next point in the path.*
- void render (sf::RenderWindow &window)

  *Render the car.*
- CityGraph::point getStart ()

  *Get the start point.*
- CityGraph::point getEnd ()

  *Get the end point.*
- double getSpeed ()

  *Get the current point in the path.*
- double getSpeedAt (int index)

  *Get the speed at a certain index in the path.*
- double getAverageSpeed (CityGraph &graph)

  *Get the average speed of the car.*

- double getRemainingTime ()

    *Get the remaining time to reach the end point.*
- double getElapsedTime ()

    *Get the elapsed time since the start of the car.*
- double getPathTime ()

    *Get the time to reach the end point from the start point.*
- double getRemainingDistance ()

    *Get the remaining distance to reach the end point.*
- double getElapsedDistance ()

    *Get the elapsed distance since the start of the car.*
- double getPathLength ()

    *Get the distance to reach the end point from the start point.*
- sf::Vector2f getPosition ()

    *Get the position of the car.*
- std::vector< sf::Vector2f > getPath ()

    *Get the path of the car.*
- std::vector< AStar::node > getAStarPath ()

    *Get the path of the car from the A∗ algorithm.*
- void **toggleDebug** ()

    *Toggle the debug mode. In debug mode, the path of the car is rendered and the car is rendered in red.*

### 3.14.1 Detailed Description

A car in the city.
This class represents a car in the city. It contains the start and end points of the car, the path of the car and the current point in the path.
Definition at line 23 of file car.h.

### 3.14.2 Member Function Documentation

#### assignExistingPath()

```
void Car::assignExistingPath (
            std::vector< sf::Vector2f > path)
```

Assign an existing path to the car.

**Parameters**

| | |
|---|---|
| *path* | The path |

Definition at line 87 of file car.cpp.

#### assignPath()

```
void Car::assignPath (
            std::vector< AStar::node > path)
```

Assign a path to the car.

**Parameters**

| | |
|---|---|
| *path* | The path |

Definition at line 76 of file car.cpp.

### assignStartEnd()

```
void Car::assignStartEnd (
            CityGraph::point start,
            CityGraph::point end)  [inline]
```

Assign the start and end points.

**Parameters**

| | |
|---|---|
| *start* | The start point |
| *end* | The end point |

Definition at line 35 of file car.h.

### chooseRandomStartEndPath()

```
void Car::chooseRandomStartEndPath (
            CityGraph & graph,
            CityMap & cityMap)
```

Choose a random start and end point in the graph.

**Parameters**

| | |
|---|---|
| *graph* | The graph |
| *cityMap* | The city map |

Definition at line 142 of file car.cpp.

### getAStarPath()

```
std::vector< AStar::node > Car::getAStarPath ()  [inline]
```

Get the path of the car from the A$*$ algorithm.

**Returns**

The path

Definition at line 154 of file car.h.

### getAverageSpeed()

```
double Car::getAverageSpeed (
            CityGraph & graph)
```

Get the average speed of the car.

**Parameters**

| | |
|---|---|
| *graph* | The graph |

**Returns**

The average speed

Definition at line 172 of file car.cpp.

### getElapsedDistance()

```
double Car::getElapsedDistance ()
```

Get the elapsed distance since the start of the car.

**Returns**

The elapsed distance

Definition at line 122 of file car.cpp.

### getElapsedTime()

`double Car::getElapsedTime ()`

Get the elapsed time since the start of the car.

**Returns**

The elapsed time

Definition at line 109 of file car.cpp.

### getEnd()

`CityGraph::point Car::getEnd () [inline]`

Get the end point.

**Returns**

The end point

Definition at line 80 of file car.h.

### getPath()

`std::vector< sf::Vector2f > Car::getPath () [inline]`

Get the path of the car.

**Returns**

The path

Definition at line 148 of file car.h.

### getPathLength()

`double Car::getPathLength ()`

Get the distance to reach the end point from the start point.

**Returns**

The distance

Definition at line 132 of file car.cpp.

### getPathTime()

`double Car::getPathTime ()`

Get the time to reach the end point from the start point.

**Returns**

The time

Definition at line 110 of file car.cpp.

### getPosition()

`sf::Vector2f Car::getPosition () [inline]`

Get the position of the car.

**Returns**

The position

Definition at line 142 of file car.h.

### getRemainingDistance()

`double Car::getRemainingDistance ()`

Get the remaining distance to reach the end point.

**Returns**

The remaining distance

Definition at line 112 of file car.cpp.

### getRemainingTime()

`double Car::getRemainingTime ()`

Get the remaining time to reach the end point.

**Returns**

The remaining time

Definition at line 108 of file car.cpp.

### getSpeed()

`double Car::getSpeed ()`

Get the current point in the path.

**Returns**

The current point in the path

Definition at line 92 of file car.cpp.

### getSpeedAt()

```
double Car::getSpeedAt (
            int index)
```

Get the speed at a certain index in the path.

**Parameters**

| | |
|---|---|
| *index* | The index |

**Returns**

The speed at the index

Definition at line 100 of file car.cpp.

### getStart()

`CityGraph::point Car::getStart () [inline]`

Get the start point.

**Returns**

The start point

Definition at line 74 of file car.h.

### render()

```
void Car::render (
            sf::RenderWindow & window)
```

Render the car.

**Parameters**

| | |
|---|---|
| *window* | The window |

Definition at line 28 of file car.cpp.
The documentation for this class was generated from the following files:

- car.h
- car.cpp

## 3.15 CityGraph Class Reference

A graph representing the city's streets and intersections using a graph.
`#include <cityGraph.h>`

### Public Member Functions

- void createGraph (const CityMap &cityMap)
    *Create a city graph.*
- std::unordered_map< point, std::vector< neighbor > > getNeighbors () const
    *Get neighbors map.*
- std::unordered_set< point > getGraphPoints () const
    *Get graph points.*
- point getRandomPoint () const
    *Get random point.*
- double getHeight () const
    *Get the height of the city graph.*
- double getWidth () const
    *Get the width of the city graph.*

### 3.15.1 Detailed Description

A graph representing the city's streets and intersections using a graph.
This class represents the city graph. It contains the neighbors of each point in the graph and the graph points.
Definition at line 82 of file cityGraph.h.

### 3.15.2 Member Function Documentation

#### createGraph()

```
void CityGraph::createGraph (
                const CityMap & cityMap)
```

Create a city graph.
This constructor creates a city graph from a city map.

**Parameters**

| | |
|---|---|
| *cityMap* | The city map |

Definition at line 23 of file cityGraph.cpp.

#### getGraphPoints()

```
std::unordered_set< point > CityGraph::getGraphPoints () const  [inline]
```

Get graph points.

**Returns**

Graph points

Definition at line 106 of file cityGraph.h.

### getHeight()

`double CityGraph::getHeight () const [inline]`

Get the height of the city graph.

**Returns**

The height of the city graph

Definition at line 118 of file cityGraph.h.

### getNeighbors()

`std::unordered_map< point, std::vector< neighbor > > CityGraph::getNeighbors () const [inline]`

Get neighbors map.

**Returns**

Neighbors map

Definition at line 100 of file cityGraph.h.

### getRandomPoint()

`CityGraph::point CityGraph::getRandomPoint () const`

Get random point.

**Returns**

Random point

Definition at line 274 of file cityGraph.cpp.

### getWidth()

`double CityGraph::getWidth () const [inline]`

Get the width of the city graph.

**Returns**

The width of the city graph

Definition at line 124 of file cityGraph.h.

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

- cityGraph.h
- cityGraph.cpp

## 3.16 CityMap Class Reference

A city map.

`#include <cityMap.h>`

### Public Member Functions

- **CityMap** ()

  *Constructor.*
- void loadFile (const std::string &filename)

  *Load a city map from a file.*
- bool isCityMapLoaded () const

  *Check if the city map is loaded.*
- std::vector< road > getRoads () const

  *Get the roads.*

- std::vector< intersection > getIntersections () const
  *Get the intersections.*
- std::vector< building > getBuildings () const
  *Get the buildings.*
- std::vector< greenArea > getGreenAreas () const
  *Get the green areas.*
- std::vector< waterArea > getWaterAreas () const
  *Get the water areas.*
- sf::Vector2f getMinLatLon () const
  *Get the minimum latitude and longitude.*
- sf::Vector2f getMaxLatLon () const
  *Get the maximum latitude and longitude.*
- int getWidth () const
  *Get the width of the city map.*
- int getHeight () const
  *Get the height of the city map.*

### 3.16.1 Detailed Description

A city map.
This class represents the city map. It contains the roads, intersections, buildings, green areas and water areas of the city.
Definition at line 74 of file cityMap.h.

### 3.16.2 Member Function Documentation

#### getBuildings()

`std::vector< building > CityMap::getBuildings () const [inline]`
Get the buildings.

**Returns**

The buildings

Definition at line 116 of file cityMap.h.

#### getGreenAreas()

`std::vector< greenArea > CityMap::getGreenAreas () const [inline]`
Get the green areas.

**Returns**

The green areas

Definition at line 122 of file cityMap.h.

#### getHeight()

`int CityMap::getHeight () const [inline]`
Get the height of the city map.

**Returns**

The height of the city map

Definition at line 152 of file cityMap.h.

#### getIntersections()

`std::vector< intersection > CityMap::getIntersections () const [inline]`

Get the intersections.

**Returns**

The intersections

Definition at line 110 of file cityMap.h.

#### getMaxLatLon()

`sf::Vector2f CityMap::getMaxLatLon () const [inline]`

Get the maximum latitude and longitude.

**Returns**

The maximum latitude and longitude

Definition at line 140 of file cityMap.h.

#### getMinLatLon()

`sf::Vector2f CityMap::getMinLatLon () const [inline]`

Get the minimum latitude and longitude.

**Returns**

The minimum latitude and longitude

Definition at line 134 of file cityMap.h.

#### getRoads()

`std::vector< road > CityMap::getRoads () const [inline]`

Get the roads.

**Returns**

The roads

Definition at line 104 of file cityMap.h.

#### getWaterAreas()

`std::vector< waterArea > CityMap::getWaterAreas () const [inline]`

Get the water areas.

**Returns**

The water areas

Definition at line 128 of file cityMap.h.

#### getWidth()

`int CityMap::getWidth () const [inline]`

Get the width of the city map.

**Returns**

The width of the city map

Definition at line 146 of file cityMap.h.

#### isCityMapLoaded()

```
bool CityMap::isCityMapLoaded () const  [inline]
```

Check if the city map is loaded.

**Returns**

True if the city map is loaded, false otherwise

Definition at line 98 of file cityMap.h.

#### loadFile()

```
void CityMap::loadFile (
            const std::string & filename)
```

Load a city map from a file.

**Parameters**

| | |
|---|---|
| *filename* | The filename |

Definition at line 23 of file cityMap.cpp.

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

- cityMap.h
- cityMap.cpp

## 3.17 ConstraintController Class Reference

Controller for constraints.

`#include <aStar.h>`

### Public Member Functions

- **ConstraintController** ()

  *Constructor.*
- ConstraintController copy ()

  *Copy constructor.*
- ConstraintController copy (std::vector< int > cars)

  *Copy constructor.*
- void addConstraint (AStar::conflict constraints)

  *Add a constraint.*
- bool hasConstraint (AStar::conflict constraint)

  *Check if a constraint exists.*
- bool checkConstraints (int car, double speed, double newSpeed, double time, CityGraph::point from, CityGraph::neighbor to)

  *Check if a car can move to a certain point in the graph at a certain time.*

### 3.17.1 Detailed Description

Controller for constraints.

This class is used to control the constraints of the A∗ algorithm. It is used to check if a car can move to a certain point in the graph at a certain time.

Definition at line 114 of file aStar.h.

### 3.17.2 Member Function Documentation

#### addConstraint()

```
void ConstraintController::addConstraint (
            AStar::conflict constraints)
```

Add a constraint.

**Parameters**

| | |
|---|---|
| *constraints* | The constraint to add |

Definition at line 15 of file constraintController.cpp.

### checkConstraints()

```
bool ConstraintController::checkConstraints (
            int car,
            double speed,
            double newSpeed,
            double time,
            CityGraph::point from,
            CityGraph::neighbor to)
```

Check if a car can move to a certain point in the graph at a certain time.

**Parameters**

| | |
|---|---|
| *car* | The car |
| *speed* | The speed of the car |
| *newSpeed* | The new speed of the car |
| *time* | The time |
| *from* | The point from which the car is moving |
| *to* | The point to which the car is moving |

**Returns**

True if the car can move to the point, false otherwise

Definition at line 74 of file constraintController.cpp.

### copy() [1/2]

```
ConstraintController ConstraintController::copy ()
```

Copy constructor.

**Returns**

A copy of the object

Definition at line 52 of file constraintController.cpp.

### copy() [2/2]

```
ConstraintController ConstraintController::copy (
            std::vector< int > cars)
```

Copy constructor.

**Parameters**

| | |
|---|---|
| *cars* | The cars to copy |

**Returns**

A copy of the object

Definition at line 60 of file constraintController.cpp.

#### hasConstraint()

```
bool ConstraintController::hasConstraint (
            AStar::conflict constraint)
```

Check if a constraint exists.

**Parameters**

| | |
|---|---|
| *constraint* | The constraint to check |

**Returns**

True if the constraint exists, false otherwise

Definition at line 32 of file constraintController.cpp.

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

- aStar.h
- constraintController.cpp

## 3.18 DataManager Class Reference

Data manager.

`#include <dataManager.h>`

#### Public Member Functions

- DataManager (std::string filename)

  *Constructor.*
- void createData (int numData, int numCarsMin, int numCarsMax, std::string mapName)

  *Create data. It launches multiple simulations with different number of cars and car densities. Then, it calculates different statistics and stores them in a file.*

### 3.18.1 Detailed Description

Data manager.

This class represents the data manager. It creates data and stores it in a file.

Definition at line 30 of file dataManager.h.

### 3.18.2 Constructor & Destructor Documentation

#### DataManager()

```
DataManager::DataManager (
            std::string filename)
```

Constructor.

**Parameters**

| | |
|---|---|
| *filename* | The filename |

Definition at line 20 of file dataManager.cpp.

### 3.18.3 Member Function Documentation

#### createData()

```
void DataManager::createData (
            int numData,
            int numCarsMin,
```

```
int numCarsMax,
std::string mapName)
```

Create data. It launches multiple simulations with different number of cars and car densities. Then, it calculates different statistics and stores them in a file.

**Parameters**

| | |
|---|---|
| *numData* | The number of data |
| *numCarsMin* | The minimum number of cars |
| *numCarsMax* | The maximum number of cars |
| *mapName* | The map name |

Definition at line 28 of file dataManager.cpp.

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

- dataManager.h
- dataManager.cpp

## 3.19 Dubins Class Reference

Dubins path used to calculate the path between two points in the city graph.

`#include <dubins.h>`

### Public Member Functions

- Dubins (CityGraph::point start, CityGraph::neighbor end)

    *Constructor with start and end points.*
- Dubins (CityGraph::point start, CityGraph::neighbor end, double startSpeed)

    *Constructor with start point, end point and start speed.*
- Dubins (CityGraph::point start, CityGraph::neighbor end, double startSpeed, double endSpeed)

    *Constructor with start point, end point, start speed and end speed.*
- ∼**Dubins** ()

    *Destructor.*
- double distance ()

    *Get the distance to reach the end point.*
- double time ()

    *Get the time to reach the end point.*
- CityGraph::point point (double time)

    *Get the point at a certain time in the path using interpolation.*
- std::vector< CityGraph::point > path ()

    *Get the path using interpolation.*

### 3.19.1 Detailed Description

Dubins path used to calculate the path between two points in the city graph.

This class represents a Dubins path used to calculate the path between two points in the city graph. Given the start and end points, it calculates the path, the distance and the time to reach the end point.

Definition at line 26 of file dubins.h.

### 3.19.2 Constructor & Destructor Documentation

**Dubins()** `[1/3]`

```
Dubins::Dubins (
        CityGraph::point start,
        CityGraph::neighbor end)
```

Constructor with start and end points.

The class will be initialized with the start and end points. The car will run without speed limits.

**Parameters**

| | |
|---|---|
| *start* | The start point |
| *end* | The end point |

Definition at line 11 of file dubins.cpp.

#### Dubins() [2/3]

```
Dubins::Dubins (
        CityGraph::point start,
        CityGraph::neighbor end,
        double startSpeed)
```

Constructor with start point, end point and start speed.
The class will be initialized with the start and end points and the start speed. The car will accelerate to the maximum speed.

**Parameters**

| | |
|---|---|
| *start* | The start point |
| *end* | The end point |
| *startSpeed* | The start speed |

Definition at line 14 of file dubins.cpp.

#### Dubins() [3/3]

```
Dubins::Dubins (
        CityGraph::point start,
        CityGraph::neighbor end,
        double startSpeed,
        double endSpeed)
```

Constructor with start point, end point, start speed and end speed.
The class will be initialized with the start and end points, the start and end speeds. The car will accelerate uniformly to the maximum speed.

**Parameters**

| | |
|---|---|
| *start* | The start point |
| *end* | The end point |
| *startSpeed* | The start speed |
| *endSpeed* | The end speed |

Definition at line 34 of file dubins.cpp.

### 3.19.3 Member Function Documentation

#### distance()

```
double Dubins::distance ()   [inline]
```

Get the distance to reach the end point.

**Returns**

The distance

Definition at line 72 of file dubins.h.

#### path()

```
std::vector< CityGraph::point > Dubins::path ()
```

Get the path using interpolation.

**Returns**

The path

Definition at line 85 of file dubins.cpp.

#### point()

```
CityGraph::point Dubins::point (
                double time)
```

Get the point at a certain time in the path using interpolation.

**Parameters**

| | |
|---|---|
| *time* | The time |

**Returns**

The point

Definition at line 64 of file dubins.cpp.

#### time()

```
double Dubins::time ()
```

Get the time to reach the end point.

**Returns**

The time

Definition at line 62 of file dubins.cpp.
The documentation for this class was generated from the following files:

- dubins.h
- dubins.cpp

## 3.20 DubinsPath Class Reference

Dubins path used to calculate the path between two points in the city graph.

```
#include <dubins.h>
```

### Public Member Functions

- DubinsPath (std::vector< AStar::node > path)

  *Constructor with path.*
- std::vector< CityGraph::point > **path** ()

  *Get the path.*

### 3.20.1 Detailed Description

Dubins path used to calculate the path between two points in the city graph.
This class represents a Dubins path used to calculate the path between two points in the city graph. Given the start and end points, it calculates the path, the distance and the time to reach the end point.
Definition at line 112 of file dubins.h.

### 3.20.2 Constructor & Destructor Documentation

#### DubinsPath()

```
DubinsPath::DubinsPath (
            std::vector< AStar::node > path)
```

Constructor with path.
The class will be initialized with the path.

**Parameters**

| | |
|---|---|
| *path* | The path |

Definition at line 95 of file dubins.cpp.
The documentation for this class was generated from the following files:

- dubins.h
- dubins.cpp

## 3.21 FileSelector Class Reference

A file selector.
`#include <fileSelector.h>`

### 3.21.1 Detailed Description

A file selector.
This class represents a file selector. It allows the user to select a file from a folder.
Definition at line 19 of file fileSelector.h.
The documentation for this class was generated from the following files:

- fileSelector.h
- fileSelector.cpp

## 3.22 Manager Class Reference

A manager for the cars.
`#include <manager.h>`

#### Public Member Functions

- Manager (const CityGraph &cityGraph, const CityMap &CityMap, bool log)

  *Constructor.*
- Manager (const CityGraph &cityGraph, const CityMap &CityMap, std::vector< Car > cars, bool log)

  *Constructor.*
- void createCarsAStar (int numCars)

  *Create cars using A∗ pathfinding, no collision avoidance.*
- std::pair< bool, DataManager::data > createCarsCBS (int numCars)

  *Create cars using CBS pathfinding.*
- CBSNode createSubCBS (CBSNode &node, int subNodeDepth)

  *Create a sub-CBS node.*
- CBSNode processCBS (ConstraintController constraints, int subNodeDepth)

  *Process a CBS node.*
- bool hasConflict (std::vector< std::vector< sf::Vector2f > > paths, int ∗car1, int ∗car2, sf::Vector2f ∗p1, sf↩ ::Vector2f ∗p2, double ∗a1, double ∗a2, int ∗time)

  *Check if two cars have a conflict.*
- void **moveCars** ()

*Move the cars to the next point in the path.*

- void renderCars (sf::RenderWindow &window)

  *Render the cars.*

- void toggleCarDebug (sf::Vector2f mousePos)

  *Toggle the debug of one car.*

- int getNumCars ()

  *Get the number of cars.*

- std::vector< Car > getCars ()

  *Get the cars.*

### 3.22.1 Detailed Description

A manager for the cars.
The manager class is used to manage the cars during the CBS pathfinding. It creates the cars and resolves conflicts using the CBS algorithm.
Definition at line 45 of file manager.h.

### 3.22.2 Constructor & Destructor Documentation

#### Manager() [1/2]

```
Manager::Manager (
            const CityGraph & cityGraph,
            const CityMap & CityMap,
            bool log)  [inline]
```

Constructor.

**Parameters**

| | |
|---|---|
| *cityGraph* | The city graph |
| *CityMap* | The city map |
| *log* | If the manager should log |

Definition at line 55 of file manager.h.

#### Manager() [2/2]

```
Manager::Manager (
            const CityGraph & cityGraph,
            const CityMap & CityMap,
            std::vector< Car > cars,
            bool log)  [inline]
```

Constructor.

**Parameters**

| | |
|---|---|
| *cityGraph* | The city graph |
| *CityMap* | The city map |
| *cars* | The cars |
| *log* | If the manager should log |

Definition at line 66 of file manager.h.

### 3.22.3 Member Function Documentation

#### createCarsAStar()

```
void Manager::createCarsAStar (
             int numCars)
```
Create cars using A∗ pathfinding, no collision avoidance.

**Parameters**

| | |
|---|---|
| *numCars* | The number of cars |

Definition at line 13 of file manager.cpp.

#### createCarsCBS()

```
std::pair< bool, DataManager::data > Manager::createCarsCBS (
             int numCars)
```
Create cars using CBS pathfinding.

**Parameters**

| | |
|---|---|
| *numCars* | The number of cars |

**Returns**

The data for the cars (success, data)

Definition at line 16 of file managerCBS.cpp.

#### createSubCBS()

```
Manager::CBSNode Manager::createSubCBS (
             CBSNode & node,
             int subNodeDepth)
```
Create a sub-CBS node.

**Parameters**

| | |
|---|---|
| *node* | The parent CBS node |
| *subNodeDepth* | The depth of the sub-CBS node |

**Returns**

The sub-CBS node

This function creates a sub-CBS node from a parent CBS node. It creates a new node with the same paths and constraints as the parent node, but with less agents.
Definition at line 90 of file managerCBS.cpp.

#### getCars()

```
std::vector< Car > Manager::getCars ()  [inline]
```
Get the cars.

**Returns**

The cars

Definition at line 150 of file manager.h.

### getNumCars()

```
int Manager::getNumCars ()  [inline]
```

Get the number of cars.

**Returns**

The number of cars

Definition at line 144 of file manager.h.

### hasConflict()

```
bool Manager::hasConflict (
            std::vector< std::vector< sf::Vector2f > > paths,
            int * car1,
            int * car2,
            sf::Vector2f * p1,
            sf::Vector2f * p2,
            double * a1,
            double * a2,
            int * time)
```

Check if two cars have a conflict.

**Parameters**

| | |
|---|---|
| *paths* | The paths of the cars |
| *car1* | The first car |
| *car2* | The second car |
| *p1* | The position of the first car |
| *p2* | The position of the second car |
| *a1* | The angle of the first car |
| *a2* | The angle of the second car |
| *time* | The time of the conflict |

**Returns**

If the cars have a conflict

Definition at line 325 of file managerCBS.cpp.

### processCBS()

```
Manager::CBSNode Manager::processCBS (
            ConstraintController constraints,
            int subNodeDepth)
```

Process a CBS node.

**Parameters**

| | |
|---|---|
| *constraints* | The constraints |
| *subNodeDepth* | The depth of the sub-CBS node |

**Returns**

The processed CBS node

This function processes a CBS node. It resolves conflicts and returns a new CBS node with the resolved conflicts.
Definition at line 167 of file managerCBS.cpp.

#### renderCars()

```
void Manager::renderCars (
             sf::RenderWindow & window)
```

Render the cars.

**Parameters**

| | |
|---|---|
| *window* | The window |

Definition at line 37 of file manager.cpp.

#### toggleCarDebug()

```
void Manager::toggleCarDebug (
             sf::Vector2f mousePos)
```

Toggle the debug of one car.

**Parameters**

| | |
|---|---|
| *mousePos* | The mouse position |

This function toggles the debug of a car. If the mouse is over a car, the debug of the car is toggled.
Definition at line 43 of file manager.cpp.
The documentation for this class was generated from the following files:

- manager.h
- manager.cpp
- managerCBS.cpp

## 3.23 PriorityQueue< T > Class Template Reference

Priority Queue.
`#include <priorityQueue.h>`

### Public Member Functions

- PriorityQueue (int size)

    *Constructor.*
- ~**PriorityQueue** ()

    *Destructor.*
- void push (T e, double p)

    *Push an element with a priority.*
- T pop ()

    *Pop the element with the lowest priority.*
- bool empty ()

    *Get the element with the lowest priority.*

### 3.23.1 Detailed Description

**template<class T>**
**class PriorityQueue< T >**
Priority Queue.

**Template Parameters**

| | |
|---|---|
| *T* | Type of the elements |

---

This class is a simple priority queue implementation. With a fixed size, it will keep the elements sorted by their priority. The elements with the lowest priority will be at the front of the queue.
Definition at line 20 of file priorityQueue.h.

### 3.23.2 Constructor & Destructor Documentation

#### PriorityQueue()

```
template<class T>
PriorityQueue< T >::PriorityQueue (
                int size) [inline]
```

Constructor.

**Parameters**

| | |
|---|---|
| *size* | The size of the queue |

Definition at line 26 of file priorityQueue.h.

### 3.23.3 Member Function Documentation

#### empty()

```
template<class T>
bool PriorityQueue< T >::empty () [inline]
```

Get the element with the lowest priority.

**Returns**

The element

Definition at line 88 of file priorityQueue.h.

#### pop()

```
template<class T>
T PriorityQueue< T >::pop () [inline]
```

Pop the element with the lowest priority.

**Returns**

The element

Definition at line 69 of file priorityQueue.h.

#### push()

```
template<class T>
void PriorityQueue< T >::push (
                T e,
                double p) [inline]
```

Push an element with a priority.

**Parameters**

| | |
|---|---|
| *e* | The element |
| *p* | The priority |

Definition at line 44 of file priorityQueue.h.
The documentation for this class was generated from the following file:

- priorityQueue.h

## 3.24 Renderer Class Reference

A renderer for the city.

```
#include <renderer.h>
```

### Public Member Functions

- void **startRender** (const CityMap &cityMap, const CityGraph &cityGraph, Manager &manager)

  *Start the rendering.*
- void renderCityMap (const CityMap &cityMap)

  *Render the city map.*
- void renderCityGraph (const CityGraph &cityGraph, const sf::View &view)

  *Render the city graph.*
- void renderManager (Manager &manager)

  *Render the cars.*
- void **renderTime** ()

  *Render the time.*
- void **setConflicts** (const std::vector< AStar::conflict > &conflicts)

  *Render the conflicts.*

### 3.24.1 Detailed Description

A renderer for the city.
The renderer class is used to render the city map, the city graph and the cars.
Definition at line 19 of file renderer.h.

### 3.24.2 Member Function Documentation

#### renderCityGraph()

```
void Renderer::renderCityGraph (
            const CityGraph & cityGraph,
            const sf::View & view)
```

Render the city graph.

**Parameters**

| | |
|---|---|
| *cityGraph* | The city graph |
| *view* | The view |

Definition at line 250 of file renderer.cpp.

#### renderCityMap()

```
void Renderer::renderCityMap (
            const CityMap & cityMap)
```

Render the city map.

**Parameters**

| | |
|---|---|
| *cityMap* | The city map |

Definition at line 147 of file renderer.cpp.

#### renderManager()

```
void Renderer::renderManager (
            Manager & manager)
```

Render the cars.

**Parameters**

| | |
|---|---|
| *manager* | The manager |

Definition at line 341 of file renderer.cpp.
The documentation for this class was generated from the following files:

- renderer.h
- renderer.cpp

## 3.25 Test Class Reference

A class for testing the project.
`#include <test.h>`

### Public Member Functions

- void **runTests** ()

  *Run the tests.*

### 3.25.1 Detailed Description

A class for testing the project.
This class is used to test the project.
Definition at line 13 of file test.h.
The documentation for this class was generated from the following files:

- test.h
- test.cpp

## 3.26 TimedAStar Class Reference

Timed A∗ algorithm.
`#include <aStar.h>`

### Public Member Functions

- TimedAStar (CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph, ConstraintController ∗constraints, int carIndex)

  *Constructor.*
- std::vector< AStar::node > findPath ()

  *Find the path.*

### 3.26.1 Detailed Description

Timed A∗ algorithm.
This class represents the timed A∗ algorithm. It is used to find the shortest path between two points in a graph while taking into account the constraints of the cars.
Definition at line 171 of file aStar.h.

### 3.26.2 Constructor & Destructor Documentation

**TimedAStar()**

```
TimedAStar::TimedAStar (
            CityGraph::point start,
            CityGraph::point end,
            const CityGraph & cityGraph,
```

```
ConstraintController * constraints,
int carIndex)
```

Constructor.

**Parameters**

| | |
|---|---|
| *start* | The start point |
| *end* | The end point |
| *cityGraph* | The graph |
| *constraints* | The constraints |
| *carIndex* | The car index |

Definition at line 16 of file timedAStar.cpp.

### 3.26.3 Member Function Documentation

**findPath()**

```
std::vector< AStar::node > TimedAStar::findPath ()  [inline]
```

Find the path.

**Returns**

The path

Definition at line 188 of file aStar.h.
The documentation for this class was generated from the following files:

- aStar.h
- timedAStar.cpp

# 4 File Documentation

## 4.1 aStar.h File Reference

A∗ algorithm.

```
#include "cityGraph.h"
```

### Classes

- struct _aStarNode

  *A node for the A∗ algorithm.*
- struct _aStarConflict

  *A conflict for the A∗ algorithm.*
- class AStar

  *A∗ algorithm.*
- class ConstraintController

  *Controller for constraints.*
- class TimedAStar

  *Timed A∗ algorithm.*

### 4.1.1 Detailed Description

A∗ algorithm.
This file contains the A∗ algorithm. It is used to find the shortest path between two points in a graph. It also contains the timed A∗ algorithm, which is used to find the shortest path between two points in a graph while taking into account the constraints of the cars.
Definition in file aStar.h.

## 4.2 aStar.h

Go to the documentation of this file.

```
#pragma once

#include "cityGraph.h"

typedef struct _aStarNode {
  CityGraph::point point;
  double speed;
  std::pair<CityGraph::point, CityGraph::neighbor> arcFrom;

  bool operator==(const _aStarNode &other) const {
    double s = std::round(speed / SPEED_RESOLUTION);
    double oS = std::round(other.speed / SPEED_RESOLUTION);

    return point == other.point && s == oS && arcFrom.first == other.arcFrom.first &&
           arcFrom.second == other.arcFrom.second;
  }
} _aStarNode;

typedef struct _aStarConflict {
  CityGraph::point point;
  int time;
  int car;

  bool operator==(const _aStarConflict &other) const {
    return point == other.point && time == other.time && car == other.car;
  }
} _aStarConflict;

namespace std {
template <> struct hash<_aStarNode> {
  std::size_t operator()(const _aStarNode &point) const {
    double s = std::round(point.speed / SPEED_RESOLUTION);

    return std::hash<CityGraph::point>()(point.point) ^ std::hash<double>()(s) ^
           std::hash<CityGraph::point>()(point.arcFrom.first) ^ std::hash<CityGraph::neighbor>()(point.arcFrom.second);
  }
};
template <> struct hash<_aStarConflict> {
  std::size_t operator()(const _aStarConflict &conflict) const {
    return std::hash<CityGraph::point>()(conflict.point) ^ std::hash<int>()(conflict.time) ^
           std::hash<int>()(conflict.car);
  }
};
} // namespace std

class AStar {
public:
  using node = _aStarNode;
  using conflict = _aStarConflict;

  AStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph);

  std::vector<node> findPath() {
    if (!processed)
      process();
    return path;
  }

private:
  bool processed = false;
  node start;
  node end;
  std::vector<node> path;
  CityGraph graph;

  void process();
};

class ConstraintController {
public:
  ConstraintController() { this->constraints.clear(); }

  ConstraintController copy();

  ConstraintController copy(std::vector<int> cars);

  void addConstraint(AStar::conflict constraints);

  bool hasConstraint(AStar::conflict constraint);

  bool checkConstraints(int car, double speed, double newSpeed, double time, CityGraph::point from,
```

```
                        CityGraph::neighbor to);

private:
  std::vector<std::vector<std::vector<AStar::conflict>>> constraints; // [car][time][constraints]
};

class TimedAStar {
public:
  TimedAStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph,
             ConstraintController *constraints, int carIndex);

  std::vector<AStar::node> findPath() {
    if (!processed)
      process();
    return path;
  }

private:
  bool processed = false;
  AStar::node start;
  AStar::node end;
  std::vector<AStar::node> path;
  ConstraintController *conflicts;
  int carIndex;
  CityGraph graph;

  void process();
};
```

## 4.3 car.h File Reference

A car in the city.
`#include <SFML/Graphics.hpp>`
`#include <vector>`
`#include "aStar.h"`
`#include "cityGraph.h"`

### Classes

- class Car

  *A car in the city.*

### 4.3.1 Detailed Description

A car in the city.
This file contains the declaration of the Car class. This class represents a car in the city. It contains the start and end points of the car, the path of the car and the current point in the path.
Definition in file car.h.

## 4.4 car.h

Go to the documentation of this file.

```

#pragma once

#include <SFML/Graphics.hpp>
#include <vector>

#include "aStar.h"
#include "cityGraph.h"

class Car {
public:
  Car();

  void assignStartEnd(CityGraph::point start, CityGraph::point end) {
    this->start = start;
    this->end = end;
  }

  void chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap);

  void assignPath(std::vector<AStar::node> path);
```

```

  void assignExistingPath(std::vector<sf::Vector2f> path);

  void move();

  void render(sf::RenderWindow &window);

  CityGraph::point getStart() { return start; }

  CityGraph::point getEnd() { return end; }

  double getSpeed();

  double getSpeedAt(int index);

  double getAverageSpeed(CityGraph &graph);

  double getRemainingTime();

  double getElapsedTime();

  double getPathTime();

  double getRemainingDistance();

  double getElapsedDistance();

  double getPathLength();

  sf::Vector2f getPosition() { return path[currentPoint]; }

  std::vector<sf::Vector2f> getPath() { return path; }

  std::vector<AStar::node> getAStarPath() { return aStarPath; }

  void toggleDebug() { debug = !debug; }

private:
  CityGraph::point start;
  CityGraph::point end;
  std::vector<sf::Vector2f> path;
  std::vector<AStar::node> aStarPath;
  int currentPoint = 0;
  bool debug = false;
  sf::Color color;
};
```

## 4.5 cityGraph.h File Reference

A graph representing the city's streets and intersections using a graph.

```
#include <unordered_set>
#include "cityMap.h"
#include "config.h"
#include "utils.h"
```

### Classes

- struct _cityGraphPoint

  *A point in the city graph.*
- struct _cityGraphNeighbor

  *A neighbor of a point in the city graph.*
- class CityGraph

  *A graph representing the city's streets and intersections using a graph.*

### 4.5.1 Detailed Description

A graph representing the city's streets and intersections using a graph.
This file contains the definition of the CityGraph class.
Definition in file cityGraph.h.

## 4.6 cityGraph.h

Go to the documentation of this file.

```
#pragma once
#include <unordered_set>

#include "cityMap.h"
#include "config.h"
#include "utils.h"

typedef struct _cityGraphPoint {
  sf::Vector2f position;
  double angle;

  bool operator==(const _cityGraphPoint &other) const {
    int x = std::round(position.x / CELL_SIZE);
    int y = std::round(position.y / CELL_SIZE);
    int a = std::round(normalizeAngle(angle) / ANGLE_RESOLUTION);
    int oX = std::round(other.position.x / CELL_SIZE);
    int oY = std::round(other.position.y / CELL_SIZE);
    int oA = std::round(normalizeAngle(other.angle) / ANGLE_RESOLUTION);

    return x == oX && y == oY && a == oA;
  }
} _cityGraphPoint;

typedef struct _cityGraphNeighbor {
  _cityGraphPoint point;
  double maxSpeed;
  double turningRadius;
  double distance;
  bool isRightWay;

  bool operator==(const _cityGraphNeighbor &other) const {
    return point == other.point && maxSpeed == other.maxSpeed && turningRadius == other.turningRadius &&
           distance == other.distance && isRightWay == other.isRightWay;
  }

} _cityGraphNeighbor;

namespace std {
template <> struct hash<_cityGraphPoint> {
  std::size_t operator()(const _cityGraphPoint &point) const {
    int x = std::round(point.position.x / CELL_SIZE);
    int y = std::round(point.position.y / CELL_SIZE);
    int a = std::round(normalizeAngle(point.angle) / ANGLE_RESOLUTION);

    return std::hash<int>()(x) ^ std::hash<int>()(y) ^ std::hash<int>()(a);
  }
};
template <> struct hash<_cityGraphNeighbor> {
  std::size_t operator()(const _cityGraphNeighbor &neighbor) const {
    return std::hash<_cityGraphPoint>()(neighbor.point) ^ std::hash<double>()(neighbor.maxSpeed) ^
           std::hash<double>()(neighbor.turningRadius) ^ std::hash<double>()(neighbor.distance) ^
           std::hash<bool>()(neighbor.isRightWay);
  }
};
} // namespace std

class CityGraph {
public:
  using point = _cityGraphPoint;
  using neighbor = _cityGraphNeighbor;

  void createGraph(const CityMap &cityMap);

  std::unordered_map<point, std::vector<neighbor» getNeighbors() const { return neighbors; }

  std::unordered_set<point> getGraphPoints() const { return graphPoints; }

  point getRandomPoint() const;

  double getHeight() const { return height; }

  double getWidth() const { return width; }

private:
  std::unordered_map<point, std::vector<neighbor» neighbors;
  std::unordered_set<point> graphPoints;

  void linkPoints(const point &point1, const point &point2, int direction,
                  bool subPoints); // direction: 0 -> point1 to point2, 1 -> point2 to point1, 2 -> both
  bool canLink(const point &point1, const point &point2, double speed, double *distance) const;
```

```

    double width;
    double height;
};
```

## 4.7 cityMap.h

```
#pragma once

#include <SFML/Graphics.hpp>
#include <string>
#include <vector>

typedef struct {
    sf::Vector2f p1;
    sf::Vector2f p2;
    sf::Vector2f p1_offset;
    sf::Vector2f p2_offset;
    double angle;
} _cityMapSegment;

typedef struct {
    int id;
    std::vector<_cityMapSegment> segments;
    double width;
    int numLanes;
} _cityMapRoad;

typedef struct {
    std::vector<sf::Vector2f> points;
} _cityMapBuilding;

typedef struct {
    std::vector<sf::Vector2f> points;
    int type;
} _cityMapGreenArea;

typedef struct {
    std::vector<sf::Vector2f> points;
} _cityMapWaterArea;

typedef struct {
    int id;
    sf::Vector2f center;
    double radius;
    std::vector<std::pair<int, int» roadSegmentIds;
} _cityMapIntersection;

class CityMap {
public:
    using segment = _cityMapSegment;
    using road = _cityMapRoad;
    using building = _cityMapBuilding;
    using greenArea = _cityMapGreenArea;
    using waterArea = _cityMapWaterArea;
    using intersection = _cityMapIntersection;

    CityMap();

    void loadFile(const std::string &filename);

    bool isCityMapLoaded() const { return isLoaded; }

    std::vector<road> getRoads() const { return roads; }

    std::vector<intersection> getIntersections() const { return intersections; }

    std::vector<building> getBuildings() const { return buildings; }

    std::vector<greenArea> getGreenAreas() const { return greenAreas; }

    std::vector<waterArea> getWaterAreas() const { return waterAreas; }

    sf::Vector2f getMinLatLon() const { return minLatLon; }

    sf::Vector2f getMaxLatLon() const { return maxLatLon; }

    int getWidth() const { return width; }

    int getHeight() const { return height; }

private:
    bool isLoaded = false;

    std::vector<road> roads;
```

```
    std::vector<intersection> intersections;
    std::vector<building> buildings;
    std::vector<greenArea> greenAreas;
    std::vector<waterArea> waterAreas;

    sf::Vector2f minLatLon;
    sf::Vector2f maxLatLon;
    double width;  // in meters
    double height; // in meters
};
```

## 4.8 config.h File Reference

Configuration file.

`#include <string>`

### 4.8.1 Detailed Description

Configuration file.

Definition in file config.h.

## 4.9 config.h

Go to the documentation of this file.

```

#pragma once

#include <string>

constexpr int ENVIRONMENT = 0; // 0 = development, 1 = production
constexpr int SCREEN_WIDTH = 2880;
constexpr int SCREEN_HEIGHT = 1864;
constexpr double LOG_CBS_REFRESHRATE = 0.3; // in seconds

constexpr int EARTH_RADIUS = 6371000; // in meters

constexpr double DEFAULT_ROAD_WIDTH = 7.0; // in meters
constexpr double DEFAULT_LANE_WIDTH = 3.5; // in meters
constexpr double MIN_ROAD_WIDTH = 4.0;     // in meters
constexpr bool ROAD_ENABLE_RIGHT_HAND_TRAFFIC = false;

constexpr double ZOOM_SPEED = 0.1;
constexpr double MOVE_SPEED = 0.01;

constexpr double SIM_STEP_TIME = 0.15;  // in seconds
constexpr int CBS_PRECISION_FACTOR = 1; // CBS_PRECISION_FACTOR * SIM_STEP_TIME must not be to high
constexpr double CBS_MAX_SUB_TIME = 30; // in seconds
constexpr double CBS_MAX_OPENSET_SIZE = 5;

// For hash functions (to reduce items that are really close to each other)
constexpr double CELL_SIZE = 0.5;                 // in meters
constexpr double SPEED_RESOLUTION = 0.3;          // in m/s
constexpr double ANGLE_RESOLUTION = 0.1;          // in radians
constexpr double TIME_RESOLUTION = SIM_STEP_TIME; // in seconds

constexpr double CAR_MIN_TURNING_RADIUS = 1.5;               // in meters
constexpr double CAR_MAX_SPEED_KM = 30.0;                    // in km/h
constexpr double CAR_MAX_SPEED_MS = CAR_MAX_SPEED_KM / 3.6;  // in m/s
constexpr double CAR_MAX_G_FORCE = 5.0;                      // in m/s^2
constexpr double CAR_ACCELERATION = 3.0;                     // in m/s^2
constexpr double CAR_DECELERATION = 4.0;                     // in m/s^2
constexpr double CAR_LENGTH = 4.2;                           // in meters
constexpr double CAR_WIDTH = 1.6;                            // in meters
```

## 4.10 dataManager.h File Reference

Data manager.

`#include <string>`
`#include <vector>`

#### Classes

- struct _data

    *Data structure.*
- class DataManager

    *Data manager.*

### 4.10.1 Detailed Description

Data manager.
This file contains the data manager class.
Definition in file dataManager.h.

## 4.11 dataManager.h

Go to the documentation of this file.

```

#pragma once

#include <string>
#include <vector>

struct _data {
  double numCars;
  double carDensity;
  std::vector<double> carAvgSpeed;
};

class DataManager {
public:
  using data = _data;

  DataManager(std::string filename);

  void createData(int numData, int numCarsMin, int numCarsMax, std::string mapName);

private:
};
```

## 4.12 dubins.h File Reference

Dubins path.
`#include "aStar.h"`
`#include "cityGraph.h"`
`#include <ompl/base/State.h>`
`#include <ompl/base/StateSpace.h>`
`#include <ompl/base/spaces/DubinsStateSpace.h>`

#### Classes

- class Dubins

    *Dubins path used to calculate the path between two points in the city graph.*
- class DubinsPath

    *Dubins path used to calculate the path between two points in the city graph.*

### 4.12.1 Detailed Description

Dubins path.
This file contains the Dubins class. It is used to calculate the path between two points in the city graph. It will be used to render cars in the city and check for collisions.
Definition in file dubins.h.

## 4.13 dubins.h

Go to the documentation of this file.

```
#pragma once

#include "aStar.h"
#include "cityGraph.h"

#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>

namespace ob = ompl::base;

class Dubins {
public:
  Dubins(CityGraph::point start, CityGraph::neighbor end);

  Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed);

  Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed, double endSpeed);

  ~Dubins();

  double distance() { return endPoint.distance; }

  double time();

  CityGraph::point point(double time);

  std::vector<CityGraph::point> path();

private:
  ob::DubinsStateSpace *space;
  ob::State *start;
  ob::State *end;

  CityGraph::point startPoint;
  CityGraph::neighbor endPoint;
  double startSpeed;
  double endSpeed;
  double avgSpeed;
};

class DubinsPath {
public:
  DubinsPath(std::vector<AStar::node> path);

  std::vector<CityGraph::point> path();

private:
  void process();

  std::vector<AStar::node> path_;
  std::vector<CityGraph::point> pathProcessed_;
};
```

## 4.14 fileSelector.h File Reference

File selector.

```
#include <iostream>
#include <termios.h>
#include <unistd.h>
#include <vector>
```

### Classes

- class FileSelector

  *A file selector.*

### 4.14.1 Detailed Description

File selector.

This file contains the FileSelector class. It is used to select a file from a folder.

Definition in file fileSelector.h.

## 4.15 fileSelector.h

Go to the documentation of this file.

```
#pragma once
#include <iostream>
#include <termios.h>
#include <unistd.h>
#include <vector>

class FileSelector {
private:
  std::string folderPath;
  std::vector<std::string> files;
  int selectedIndex;

  void loadFiles();
  char getKeyPress();
  void moveCursorUp();
  void moveCursorDown();
  void displayFiles();

public:
  FileSelector(const std::string &path) : folderPath(path), selectedIndex(0) { loadFiles(); }
  ~FileSelector() { std::cout « "\033[?25h"; }

  std::string selectFile();
};
```

## 4.16 manager.h File Reference

Manager for the cars.

```
#include <SFML/Graphics.hpp>
#include <vector>
#include "car.h"
#include "cityGraph.h"
#include "dataManager.h"
```

### Classes

- struct _managerCBSNode

  *A node for the CBS algorithm.*

- class Manager

  *A manager for the cars.*

### 4.16.1 Detailed Description

Manager for the cars.

This file contains the declaration of the Manager class. This class is used to manage the cars during the CBS pathfinding. It creates the cars and resolves conflicts using the CBS algorithm.

Definition in file manager.h.

## 4.17 manager.h

Go to the documentation of this file.

```
#pragma once

#include <SFML/Graphics.hpp>
#include <vector>

#include "car.h"
#include "cityGraph.h"
#include "dataManager.h"

typedef struct _managerCBSNode {
  std::vector<std::vector<sf::Vector2f» paths;
  ConstraintController constraints;
  std::vector<double> costs;
  double cost;
```

```
  int depth;
  bool hasResolved;

  bool operator<(const _managerCBSNode &other) const {
    return cost > other.cost || (cost == other.cost && depth > other.depth);
  }

} _managerCBSNode;

class Manager {
public:
  using CBSNode = _managerCBSNode;

  Manager(const CityGraph &cityGraph, const CityMap &CityMap, bool log) : graph(cityGraph),
map(CityMap) {
    this->log = log;
  }

  Manager(const CityGraph &cityGraph, const CityMap &CityMap, std::vector<Car> cars, bool log)
      : graph(cityGraph), map(CityMap), cars(cars) {
    this->numCars = cars.size();
    this->log = log;
  }

  void createCarsAStar(int numCars);

  std::pair<bool, DataManager::data> createCarsCBS(int numCars);

  CBSNode createSubCBS(CBSNode &node, int subNodeDepth);

  CBSNode processCBS(ConstraintController constraints, int subNodeDepth);

  bool hasConflict(std::vector<std::vector<sf::Vector2f» paths, int *car1, int *car2, sf::Vector2f
*p1,
                   sf::Vector2f *p2, double *a1, double *a2, int *time);

  void moveCars();

  void renderCars(sf::RenderWindow &window);

  void toggleCarDebug(sf::Vector2f mousePos);

  int getNumCars() { return numCars; }

  std::vector<Car> getCars() { return cars; }

private:
  int numCars;
  std::vector<Car> cars;
  CityGraph graph;
  CityMap map;
  bool log;
};
```

## 4.18 priorityQueue.h File Reference

Priority Queue.

`#include <iostream>`

### Classes

- class PriorityQueue< T >

  *Priority Queue.*

### 4.18.1 Detailed Description

Priority Queue.

This file contains the declaration of the PriorityQueue class. This class is a simple priority queue implementation. With a fixed size, it will keep the elements sorted by their priority. The elements with the lowest priority will be at the front of the queue.

Definition in file priorityQueue.h.

## 4.19 priorityQueue.h

Go to the documentation of this file.

```
#pragma once
#include <iostream>

template <class T> class PriorityQueue {
public:
  PriorityQueue(int size) {
    this->size = size;
    elements = (T *)malloc(size * sizeof(T));
    priorities = (double *)malloc(size * sizeof(double));
  }
  ~PriorityQueue() {
    free(elements);
    free(priorities);
  }

  void push(T e, double p) {
    if (count < size) {
      elements[count] = e;
      priorities[count] = p;
      count++;
      return;
    }

    for (int i = 0; i < size; i++) {
      if (p < priorities[i]) {
        for (int j = size - 1; j > i; j--) {
          elements[j] = elements[j - 1];
          priorities[j] = priorities[j - 1];
        }
        elements[i] = e;
        priorities[i] = p;
        return;
      }
    }
  }

  T pop() {
    if (count == 0) {
      std::cerr « "PriorityQueue is empty" « std::endl;
      exit(1);
    }

    T e = elements[0];
    for (int i = 1; i < size; i++) {
      elements[i - 1] = elements[i];
      priorities[i - 1] = priorities[i];
    }
    count--;
    return e;
  }

  bool empty() { return count == 0; }

private:
  T *elements;
  double *priorities;
  int size;
  int count;
};
```

## 4.20 renderer.h File Reference

A renderer for the city.

```
#include <SFML/Graphics.hpp>
#include "cityGraph.h"
#include "cityMap.h"
#include "manager.h"
```

### Classes

- class Renderer

    *A renderer for the city.*

### Functions

- void drawArrow (sf::RenderWindow &window, sf::Vector2f position, double rotation, double length, double thickness, sf::Color color=sf::Color::Red, bool outline=false)

  *Draw an arrow.*

### 4.20.1 Detailed Description

A renderer for the city.
Definition in file renderer.h.

### 4.20.2 Function Documentation

#### drawArrow()

```
void drawArrow (
            sf::RenderWindow & window,
            sf::Vector2f position,
            double rotation,
            double length,
            double thickness,
            sf::Color color = sf::Color::Red,
            bool outline = false)   [inline]
```

Draw an arrow.

**Parameters**

| | |
|---|---|
| *window* | The window |
| *position* | The position |
| *rotation* | The rotation |
| *length* | The length |
| *thickness* | The thickness |
| *color* | The color |
| *outline* | If the arrow should have an outline |

Definition at line 74 of file renderer.h.

## 4.21 renderer.h

Go to the documentation of this file.

```

#pragma once

#include <SFML/Graphics.hpp>

#include "cityGraph.h"
#include "cityMap.h"
#include "manager.h"

class Renderer {
public:
  void startRender(const CityMap &cityMap, const CityGraph &cityGraph, Manager &manager);

  void renderCityMap(const CityMap &cityMap);

  void renderCityGraph(const CityGraph &cityGraph, const sf::View &view);

  void renderManager(Manager &manager);

  void renderTime();

  void setConflicts(const std::vector<AStar::conflict> &conflicts) { this->conflicts = conflicts; }

private:
  sf::RenderWindow window;
  double time;

```

```
std::vector<AStar::conflict> conflicts;

  bool debug = false;
};

inline void drawArrow(sf::RenderWindow &window, sf::Vector2f position, double rotation, double length,
double thickness,
                      sf::Color color = sf::Color::Red, bool outline = false) {
  sf::ConvexShape arrow;

  arrow.setFillColor(color);
  arrow.setOrigin(-length / 2, 0);
  arrow.setPosition(position);
  arrow.setRotation(rotation);

  arrow.setPointCount(7);
  arrow.setPoint(0, sf::Vector2f(0, 0));
  arrow.setPoint(1, sf::Vector2f(-2 * length / 5, thickness));
  arrow.setPoint(2, sf::Vector2f(-2 * length / 5, thickness / 2));
  arrow.setPoint(3, sf::Vector2f(-length, thickness / 2));
  arrow.setPoint(4, sf::Vector2f(-length, -thickness / 2));
  arrow.setPoint(5, sf::Vector2f(-2 * length / 5, -thickness / 2));
  arrow.setPoint(6, sf::Vector2f(-2 * length / 5, -thickness));

  if (outline) {
    arrow.setOutlineThickness(thickness / 10);
    arrow.setOutlineColor(sf::Color::Black);
  }

  window.draw(arrow);
}
```

## 4.22 test.h File Reference

A header file for the Test class.

### Classes

- class Test

  *A class for testing the project.*

### 4.22.1 Detailed Description

A header file for the Test class.
Definition in file test.h.

## 4.23 test.h

Go to the documentation of this file.

```

#pragma once

class Test {
public:
  void runTests();

private:
  void testSpdlog();
  void testTinyXML2();
  void testSFML();
};
```

## 4.24 utils.h File Reference

Utility functions.

```
#include "config.h"
#include <SFML/Graphics.hpp>
```

### Functions

- sf::Vector2f latLonToXY (double lat, double lon)

*Convert latitude and longitude to x and y.*

- double distance (sf::Vector2f p1, sf::Vector2f p2)

  *Get the distance between two points.*
- double normalizeAngle (double angle)

  *Normalize an angle to -PI to PI.*
- double turningRadius (double speed)

  *Get the turning radius from the speed.*
- double turningRadiusToSpeed (double radius)

  *Get the speed from the turning radius.*
- bool carsCollided (Car car1, Car car2, int time)
- bool carConflict (sf::Vector2f carPos, double carAngle, sf::Vector2f confPos, double confAngle)

  *Check if two cars have a conflict.*
- sf::Font loadFont ()

  *Load a font.*

### 4.24.1 Detailed Description

Utility functions.
Definition in file utils.h.

### 4.24.2 Function Documentation

#### carConflict()

```
bool carConflict (
            sf::Vector2f carPos,
            double carAngle,
            sf::Vector2f confPos,
            double confAngle)
```

Check if two cars have a conflict.

**Parameters**

| | |
|---|---|
| *carPos* | The position of the car |
| *carAngle* | The angle of the car |
| *confPos* | The position of the conflicting car |
| *confAngle* | The angle of the conflicting car |

**Returns**

If the cars have a conflict

Definition at line 49 of file utils.cpp.

#### carsCollided()

```
bool carsCollided (
            Car car1,
            Car car2,
            int time)
```

@bref Check if two cars collided

**Parameters**

| | |
|---|---|
| *car1* | The first car |
| *car2* | The second car |

Definition at line 23 of file utils.cpp.

### distance()

```
double distance (
            sf::Vector2f p1,
            sf::Vector2f p2)  [inline]
```

Get the distance between two points.

**Parameters**

| | |
|---|---|
| *p1* | The first point |
| *p2* | The second point |

Definition at line 29 of file utils.h.

### latLonToXY()

```
sf::Vector2f latLonToXY (
            double lat,
            double lon)  [inline]
```

Convert latitude and longitude to x and y.

**Parameters**

| | |
|---|---|
| *lat* | The latitude |
| *lon* | The longitude |

**Returns**

The x and y

Definition at line 17 of file utils.h.

### loadFont()

```
sf::Font loadFont ()
```

Load a font.

**Returns**

The font

Definition at line 13 of file utils.cpp.

### normalizeAngle()

```
double normalizeAngle (
            double angle)  [inline]
```

Normalize an angle to -PI to PI.

**Parameters**

| | |
|---|---|
| *angle* | The angle |

Definition at line 37 of file utils.h.

### turningRadius()

```
double turningRadius (
            double speed)  [inline]
```

Get the turning radius from the speed.

**Parameters**

| | |
|---|---|
| *speed* | The speed |

**Returns**

The turning radius

Definition at line 52 of file utils.h.

### turningRadiusToSpeed()

```
double turningRadiusToSpeed (
            double radius)  [inline]
```

Get the speed from the turning radius.

**Parameters**

| | |
|---|---|
| *radius* | The turning radius |

**Returns**

The speed

Definition at line 59 of file utils.h.

## 4.25 utils.h

Go to the documentation of this file.

```
00001
00005 #pragma once
00006 #include "config.h"
00007 #include <SFML/Graphics.hpp>
00008
00009 class Car;
00010
00017 inline sf::Vector2f latLonToXY(double lat, double lon) {
00018   sf::Vector2f xy;
00019   xy.x = EARTH_RADIUS * lon * M_PI / 180;
00020   xy.y = EARTH_RADIUS * std::log(std::tan((90.0f + lat) * M_PI / 360.0f));
00021   return xy;
00022 }
00023
00029 inline double distance(sf::Vector2f p1, sf::Vector2f p2) {
00030   return std::sqrt(std::pow(p2.x - p1.x, 2) + std::pow(p2.y - p1.y, 2));
00031 }
00032
00037 inline double normalizeAngle(double angle) { // -PI to PI
00038   while (angle > M_PI) {
00039     angle -= 2 * M_PI;
00040   }
00041   while (angle <= -M_PI) {
00042     angle += 2 * M_PI;
00043   }
00044   return angle;
00045 }
00046
00052 inline double turningRadius(double speed) { return speed * speed / CAR_MAX_G_FORCE; }
00053
00059 inline double turningRadiusToSpeed(double radius) { return std::sqrt(radius * CAR_MAX_G_FORCE); }
00060
00066 bool carsCollided(Car car1, Car car2, int time);
00067
00076 bool carConflict(sf::Vector2f carPos, double carAngle, sf::Vector2f confPos, double confAngle);
00077
00082 sf::Font loadFont();
```

## 4.26 index.py

```
import sys
```

```python
import os
import matplotlib.pyplot as plt
import numpy as np
from collections import defaultdict

# =======================
# User-Configurable Parameters
# =======================

# Parameters for the vertical bars representing individual data points
BAR_COLOR = 'blue'              # Color of the vertical bars
BAR_WIDTH = 1                   # Width of the bars (in data units)
BAR_VERTICAL_OFFSET = 0.3       # Vertical offset: each bar spans from (y - offset) to (y + offset)
BAR_ALPHA = 0.1                 # Opacity of the bars (0.0 to 1.0)

# Parameters for the mean speed trend line
MEAN_LINE_COLOR = 'red'         # Color of the mean speed line
MEAN_LINE_STYLE = '-'           # Style of the mean speed line
MEAN_LINE_WIDTH = 2             # Thickness of the mean speed line

# Parameters for the trend line (interpolation)
TREND_LINE_COLOR = 'green'      # Color of the trend line
TREND_LINE_STYLE = '-'          # Style of the trend line
TREND_LINE_WIDTH = 2            # Thickness of the trend line
TREND_DEGREE = 1                # Degree of the polynomial for the trend line (1 = linear)

# Parameters for the standard deviation bands
STD_LINE_COLOR = 'purple'       # Color of the standard deviation lines
STD_LINE_STYLE = '--'           # Style of the standard deviation lines
STD_LINE_WIDTH = 1.5            # Thickness of the standard deviation lines

# Parameters for the x-axis labels
X_LABEL_STEP = 4

# =======================
# Main Code
# =======================

def main():
    # Check if a filename is provided as a command-line argument
    if len(sys.argv) < 2:
        print("Usage: python script.py <filename>")
        sys.exit(1)

    filename = sys.argv[1]

    # Validate that the file exists
    if not os.path.isfile(filename):
        print(f"Error: File '{filename}' does not exist.")
        sys.exit(1)

    # Lists to store data points
    x_points = []               # Number of vehicles (numCar)
    y_points = []               # Converted speeds (km/h)
    speed_data = defaultdict(list)  # Dictionary mapping numCar to a list of speeds
    density_mapping = {}         # Store density values for each numCar

    # Read the file
    with open(filename, 'r', encoding='utf-8') as file:
        for line_number, line in enumerate(file, start=1):
            line = line.strip()
            if not line:
                continue

            # Split the line using ';' as the delimiter and remove empty tokens
            tokens = [token.strip() for token in line.split(';') if token.strip()]

            if len(tokens) < 2:
                continue

            # Parse numCar (the number of vehicles) and density
            try:
                num_car = int(tokens[0])
                density = float(tokens[1])  # Store density for the x-axis label
                density_mapping[num_car] = density  # Ensure each numCar has a unique density mapping
            except ValueError:
                print(f"Error on line {line_number}: Cannot parse numCar or density '{tokens[:2]}'.")
                continue

            expected_token_count = 2 + num_car
            if len(tokens) < expected_token_count:
                print(f"Error on line {line_number}: Expected {expected_token_count} values, found {len(tokens)}.")
                continue

            # Process each speed value (tokens from index 2 onward), converting from m/s to km/h
            for token in tokens[2:]:
```

```
                try:
                    speed_kmh = float(token) * 3.6
                    x_points.append(num_car)           # Use numCar as the x-value
                    y_points.append(speed_kmh)         # Store the speed (km/h)
                    speed_data[num_car].append(speed_kmh)  # Group speeds by numCar for averaging
                except ValueError:
                    print(f"Error on line {line_number}: Cannot convert '{token}' to float.")
                    continue

    if not x_points:
        print("No valid data found. Exiting.")
        sys.exit(1)

    # Compute the mean speed and standard deviation for each unique numCar
    unique_x = sorted(speed_data.keys())
    mean_y = [np.mean(speed_data[num]) for num in unique_x]
    std_y = [np.std(speed_data[num]) for num in unique_x]  # Compute standard deviation

    # Compute upper and lower bounds (±1 standard deviation)
    upper_y = [mean + std for mean, std in zip(mean_y, std_y)]
    lower_y = [mean - std for mean, std in zip(mean_y, std_y)]

    # Fit a linear trend line (degree 1)
    trend_poly = np.polyfit(unique_x, mean_y, TREND_DEGREE)
    trend_func = np.poly1d(trend_poly)

    # Generate smooth x values for plotting the trend curve
    x_smooth = np.linspace(min(unique_x), max(unique_x), 300)
    y_smooth = trend_func(x_smooth)

    # Create the plot
    fig, ax = plt.subplots(figsize=(10, 6))

    # Plot the individual data points as vertical bars using plt.bar
    bottoms = [y - BAR_VERTICAL_OFFSET for y in y_points]
    heights = [2 * BAR_VERTICAL_OFFSET for _ in y_points]
    ax.bar(x_points, heights, width=BAR_WIDTH, bottom=bottoms, color=BAR_COLOR,
           alpha=BAR_ALPHA, align='center')

    # Plot the mean speed as a continuous red line
    ax.plot(unique_x, mean_y, color=MEAN_LINE_COLOR, linestyle=MEAN_LINE_STYLE,
            linewidth=MEAN_LINE_WIDTH, label="Mean Speed")

    # Plot the trend (interpolation) curve
    ax.plot(x_smooth, y_smooth, color=TREND_LINE_COLOR, linestyle=TREND_LINE_STYLE,
            linewidth=TREND_LINE_WIDTH, label="Trend Curve")

    # Plot ±1 standard deviation lines
    ax.plot(unique_x, upper_y, color=STD_LINE_COLOR, linestyle=STD_LINE_STYLE,
            linewidth=STD_LINE_WIDTH, label="+1 Std Dev")
    ax.plot(unique_x, lower_y, color=STD_LINE_COLOR, linestyle=STD_LINE_STYLE,
            linewidth=STD_LINE_WIDTH, label="-1 Std Dev")

    # Set x-axis labels with "numCar (density)"
    x_labels = [f"{num} ({density_mapping[num]:.0f})" if i % X_LABEL_STEP == 0 else ""
            for i, num in enumerate(unique_x)]
    ax.set_xticks(unique_x)
    ax.set_xticklabels(x_labels, rotation=45, ha='right')  # Rotate for better readability
    ax.set_xlim(min(unique_x)-0.5, max(unique_x)+0.5)

    ax.set_xlabel("Number of Vehicles (Density)")
    ax.set_ylabel("Average Speed (km/h)")
    ax.set_title("Number of Vehicles vs Average Speeds with Std Deviation")
    ax.legend()

    # Display the plot (grid is not added)
    plt.show()

if __name__ == '__main__':
    main()
```

## 4.27 aStar.cpp File Reference

A∗ algorithm implementation.

```
#include "aStar.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
```

```
#include <spdlog/spdlog.h>
#include <unordered_set>
```

### 4.27.1 Detailed Description

A∗ algorithm implementation.

This file contains the implementation of the A∗ algorithm. It is used to find the shortest path between two points in a graph.

Definition in file aStar.cpp.

## 4.28 aStar.cpp

Go to the documentation of this file.

```

#include "aStar.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"

#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <spdlog/spdlog.h>
#include <unordered_set>

namespace ob = ompl::base;

AStar::AStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph) {
  this->start.point = start;
  this->start.speed = 0;
  this->end.point = end;
  this->end.speed = 0;
  this->graph = cityGraph;
}

void AStar::process() {
  path.clear();

  std::unordered_map<AStar::node, AStar::node> cameFrom;
  std::unordered_map<AStar::node, double> gScore;
  std::unordered_map<AStar::node, double> fScore;

  auto heuristic = [&](const AStar::node &a) {
    CityGraph::neighbor end_;
    end_.point = end.point;
    end_.maxSpeed = 0;
    end_.turningRadius = CAR_MIN_TURNING_RADIUS;
    Dubins dubins(a.point, end_);
    return dubins.distance();
  };
  auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };

  std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSet(compare);
  std::unordered_set<AStar::node> isInOpenSet;

  openSet.push(start);
  gScore[start] = 0;
  fScore[start] = heuristic(start);

  auto neighbors = graph.getNeighbors();

  int nbIterations = 0;
  while (!openSet.empty() && nbIterations++ < 1e5) {
    AStar::node current = openSet.top();
    openSet.pop();
    isInOpenSet.erase(current);

    if (current.point == end.point) {
      AStar::node currentCopy = current;

      while (!(currentCopy == start)) {
        path.push_back(currentCopy);
        currentCopy = cameFrom[currentCopy];
      }
      path.push_back(currentCopy);
      std::reverse(path.begin(), path.end());
      break;
    }

```

```
      for (const auto &neighborGraphPoint : neighbors[current.point]) {
        AStar::node neighbor;
        neighbor.point = neighborGraphPoint.point;
        neighbor.speed = neighborGraphPoint.maxSpeed;
        neighbor.arcFrom = {current.point, neighborGraphPoint};

        double tentativeGScore = gScore[current] + neighborGraphPoint.distance;

        if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
          cameFrom[neighbor] = current;
          gScore[neighbor] = tentativeGScore;
          fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);

          if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
            openSet.push(neighbor);
            isInOpenSet.insert(neighbor);
          }
        }
      }
    }
}
```

## 4.29 car.cpp File Reference

Car class implementation.

```
#include "car.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"
#include <iostream>
#include <random>
```

### 4.29.1 Detailed Description

Car class implementation.

This file contains the implementation of the Car class.

Definition in file car.cpp.

## 4.30 car.cpp

Go to the documentation of this file.

```

#include "car.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"

#include <iostream>
#include <random>

Car::Car() {
  std::vector<sf::Color> colors = {sf::Color(50, 120, 190), sf::Color(183, 132, 144), sf::Color(105,
    101, 89),
                                   sf::Color(182, 18, 34),  sf::Color(24, 25, 24),    sf::Color(17,
    86, 122)};
  color = colors[rand() % colors.size()];
}

void Car::move() {
  if (currentPoint >= (int)path.size())
    return;

  currentPoint++;
}

void Car::render(sf::RenderWindow &window) {
  if (1 + currentPoint >= (int)path.size())
    return;

  sf::Vector2f point = path[currentPoint];
  sf::Vector2f nextPoint = path[currentPoint + 1];

  sf::RectangleShape shape(sf::Vector2f(CAR_LENGTH, CAR_WIDTH));
  shape.setOrigin(CAR_LENGTH / 2.0f, CAR_WIDTH / 2.0f);
  shape.setPosition(point);
```

```
  shape.setRotation(atan2(nextPoint.y - point.y, nextPoint.x - point.x) * 180.0f / M_PI);
  if (debug)
    shape.setFillColor(sf::Color(255, 0, 0));
  else
    shape.setFillColor(color);
  window.draw(shape);

  if (!debug)
    return;

  // Render speed, elapsed time, remaining time, and distance
  int speed = (int)(getSpeed() * 3.6f);
  int dSpeed = (getSpeed() * 3.6f - (double)speed) * 100;
  sf::Font font = loadFont();
  sf::Text text;
  text.setFont(font);
  text.setCharacterSize(24);
  text.setFillColor(sf::Color::White);
  text.setPosition(getPosition());
  text.setString(std::to_string(speed) + "." + std::to_string(dSpeed) + " km/h" + "\n" +
                 std::to_string((int)getElapsedTime()) + "s / " +
std::to_string((int)getRemainingTime()) + "s" + "\n" +
                 std::to_string((int)getElapsedDistance()) + "m / " +
std::to_string((int)getRemainingDistance()) +
                 "m");
  text.setOutlineColor(sf::Color::Black);
  text.setOutlineThickness(1.0f);
  text.scale(0.1f, 0.1f);
  text.setOrigin(text.getLocalBounds().width / 2.0f, text.getLocalBounds().height / 2.0f);
  window.draw(text);

  // Render path
  for (int i = currentPoint; i < (int)path.size() - 1; i++) {
    sf::Vertex line[] = {sf::Vertex(path[i]), sf::Vertex(path[i + 1])};
    line[0].color = sf::Color(255, 255, 255);
    line[1].color = sf::Color(255, 255, 255);
    window.draw(line, 2, sf::Lines);
  }
}

void Car::assignPath(std::vector<AStar::node> path) {
  this->path.clear();
  this->aStarPath = path;
  DubinsPath dubins(path);
  std::vector<CityGraph::point> dubinsPath_ = dubins.path();
  for (CityGraph::point point : dubinsPath_) {
    this->path.push_back(point.position);
  }
  currentPoint = 0;
}

void Car::assignExistingPath(std::vector<sf::Vector2f> path) {
  this->path = path;
  currentPoint = 0;
}

double Car::getSpeed() {
  if (currentPoint >= (int)path.size() - 1)
    return 0;

  sf::Vector2f diff = path[currentPoint + 1] - path[currentPoint];
  return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
}

double Car::getSpeedAt(int index) {
  if (index >= (int)path.size() - 1)
    return 0;

  sf::Vector2f diff = path[index + 1] - path[index];
  return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
}

double Car::getRemainingTime() { return (double)(path.size() - currentPoint) * SIM_STEP_TIME; }
double Car::getElapsedTime() { return (double)currentPoint * SIM_STEP_TIME; }
double Car::getPathTime() { return (double)path.size() * SIM_STEP_TIME; }

double Car::getRemainingDistance() {
  double dist = 0;
  for (int i = currentPoint; i < (int)path.size() - 1; i++) {
    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
  }

  return dist;
}

double Car::getElapsedDistance() {
```

```
  double dist = 0;
  for (int i = 0; i < currentPoint; i++) {
    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
  }

  return dist;
}

double Car::getPathLength() {
  double dist = 0;
  for (int i = 0; i < (int)path.size() - 1; i++) {
    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
  }

  return dist;
}

void Car::chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap) {
  CityGraph::point start;
  CityGraph::point end;

  double minDistance = std::max(graph.getWidth(), graph.getHeight()) / 2.0;
  std::vector<AStar::node> path;

  do {
    path.clear();
    start = graph.getRandomPoint();
    end = graph.getRandomPoint();

    if (std::sqrt(std::pow(start.position.x - end.position.x, 2) + std::pow(start.position.y -
end.position.y, 2)) <
        minDistance)
      continue;

    AStar aStar(start, end, graph);
    path = aStar.findPath();

    if (!path.empty() && (int)path.size() >= 3) {
      TimedAStar timedAStar(start, end, graph, nullptr, 0);
      path.clear();
      path = timedAStar.findPath();
    }
  } while (path.empty() || (int)path.size() < 3);

  this->assignStartEnd(start, end);
  this->assignPath(path);
}

double Car::getAverageSpeed(CityGraph &graph) {
  double dist = 0;
  double time = 0;
  auto outOfBounds = [&](sf::Vector2f p) {
    return p.x < 0 || p.y < 0 || p.x > graph.getWidth() || p.y > graph.getWidth();
  };

  for (int i = 0; i < (int)path.size() - 1; i++) {
    if (outOfBounds(path[i]) || outOfBounds(path[i + 1]))
      continue;

    sf::Vector2f diff = path[i + 1] - path[i];
    dist += sqrt(diff.x * diff.x + diff.y * diff.y);
    time += SIM_STEP_TIME;
  }

  if (time == 0)
    return 0;

  return dist / time;
}
```

## 4.31 cityGraph.cpp File Reference

City graph implementation.

```
#include <iostream>
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <ompl/geometric/SimpleSetup.h>
#include <ompl/geometric/planners/rrt/RRT.h>
#include <random>
```

```
#include <spdlog/spdlog.h>
#include "cityGraph.h"
#include "config.h"
#include "utils.h"
```

### 4.31.1 Detailed Description

City graph implementation.

This file contains the implementation of the CityGraph class. This class represents the graph of the city. It contains the points of the graph and the neighbors of each point.

Definition in file cityGraph.cpp.

## 4.32 cityGraph.cpp

Go to the documentation of this file.

```

#include <iostream>
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <ompl/geometric/SimpleSetup.h>
#include <ompl/geometric/planners/rrt/RRT.h>
#include <random>
#include <spdlog/spdlog.h>

#include "cityGraph.h"
#include "config.h"
#include "utils.h"

namespace ob = ompl::base;

void CityGraph::createGraph(const CityMap &cityMap) {
  auto roads = cityMap.getRoads();
  auto intersections = cityMap.getIntersections();

  this->height = cityMap.getHeight();
  this->width = cityMap.getWidth();

  // Graph's points are evenly distributed along a road segment
  for (const auto &road : roads) {
    if (road.segments.empty()) {
      continue;
    }

    int numSeg = 0;
    for (const auto &segment : road.segments) {
      if (numSeg > 0) { // Link to the previous one
        for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
          double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
          offset += road.width / (2 * road.numLanes);

          point point1;
          point1.angle = road.segments[numSeg - 1].angle;
          point1.position =
              sf::Vector2f(road.segments[numSeg - 1].p2_offset.x + offset * sin(road.segments[numSeg - 1].angle),
                           road.segments[numSeg - 1].p2_offset.y + offset * -cos(road.segments[numSeg - 1].angle));

          point point2;
          point2.angle = road.segments[numSeg].angle;
          point2.position =
              sf::Vector2f(road.segments[numSeg].p1_offset.x + offset * sin(road.segments[numSeg].angle),
                           road.segments[numSeg].p1_offset.y + offset * -cos(road.segments[numSeg].angle));

          linkPoints(point1, point2, 2, true);
        }
      }
      numSeg++;

      double segmentLength =
          sqrt(pow(segment.p2_offset.x - segment.p1_offset.x, 2) + pow(segment.p2_offset.y - segment.p1_offset.y, 2));
      double pointDistance = 15;
      int numPoints = segmentLength / pointDistance;
```

```
      double dx_s = (segment.p2_offset.x - segment.p1_offset.x) / numPoints;
      double dy_s = (segment.p2_offset.y - segment.p1_offset.y) / numPoints;
      double dx_a = sin(segment.angle);
      double dy_a = -cos(segment.angle);

      if (dx_a < 0) {
        dx_a = -dx_a;
        dy_a = -dy_a;
      }

      for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
        double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
        offset += road.width / (2 * road.numLanes);

        if (numPoints == 0) {
          point point1;
          point1.angle = segment.angle;
          point1.position = sf::Vector2f(segment.p1_offset.x + offset * dx_a, segment.p1_offset.y + offset * dy_a);

          point point2;
          point2.angle = segment.angle;
          point2.position = sf::Vector2f(segment.p2_offset.x + offset * dx_a, segment.p2_offset.y + offset * dy_a);

          linkPoints(point1, point2, 2, true);
          continue;
        }

        for (int i = 0; i <= numPoints; i++) {
          point point1;
          point1.position = sf::Vector2f(segment.p1_offset.x + i * dx_s + offset * dx_a,
                                         segment.p1_offset.y + i * dy_s + offset * dy_a);
          point1.angle = segment.angle;

          if (i > 0) {
            for (int i2_lane = 0; i2_lane < road.numLanes; i2_lane++) {
              double offset2 = ((double)i2_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
              offset2 += road.width / (2 * road.numLanes);

              point point2;
              point2.position = sf::Vector2f(segment.p1_offset.x + (i - 1) * dx_s + offset2 * dx_a,
                                             segment.p1_offset.y + (i - 1) * dy_s + offset2 * dy_a);
              point2.angle = segment.angle;

              int direction = 2;
              double a = normalizeAngle(atan2(dy_a, dx_a));
              if (offset == offset2 || (offset >= 0 && offset2 >= 0)) {
                if (dy_s >= 0) {
                  direction = offset > 0 ? 0 : 1;
                } else {
                  direction = offset > 0 ? 1 : 0;
                }
                linkPoints(point1, point2, direction, offset == offset2);
              } else {
                if (!ROAD_ENABLE_RIGHT_HAND_TRAFFIC) {
                  linkPoints(point1, point2, 2, true);
                }
              }
            }
          }
        }
      }
    }
  }

  // Connect the intersections
  for (const auto &intersection : intersections) {
    for (const auto &roadSegmentId1 : intersection.roadSegmentIds) {
      for (const auto &roadSegmentId2 : intersection.roadSegmentIds) {
        const auto &road1 = roads[roadSegmentId1.first];
        const auto &road2 = roads[roadSegmentId2.first];
        const auto &segment1 = road1.segments[roadSegmentId1.second];
        const auto &segment2 = road2.segments[roadSegmentId2.second];

        // Find the point of the segment2 closest to the intersection
        point point1;
        point1.angle = segment1.angle;
        point1.position = (distance(segment1.p1, intersection.center) < distance(segment1.p2, intersection.center))
                              ? segment1.p1_offset
                              : segment1.p2_offset;

        point point2;
        point2.angle = segment2.angle;
        point2.position = (distance(segment2.p1, intersection.center) < distance(segment2.p2,
```

```
intersection.center))
                              ? segment2.p1_offset
                              : segment2.p2_offset;

        for (int iL_1 = 0; iL_1 < road1.numLanes; iL_1++) {
          double offset1 = ((double)iL_1 - (double)road1.numLanes / 2.0f) * road1.width / road1.numLanes;
          offset1 += road1.width / (2 * road1.numLanes);

          for (int iL_2 = 0; iL_2 < road2.numLanes; iL_2++) {
            double offset2 = ((double)iL_2 - (double)road2.numLanes / 2.0f) * road2.width / road2.numLanes;
            offset2 += road2.width / (2 * road2.numLanes);

            point point1_offset;
            point1_offset.angle = segment1.angle;
            point1_offset.position = sf::Vector2f(point1.position.x + offset1 * sin(segment1.angle),
                                                  point1.position.y + offset1 * -cos(segment1.angle));

            point point2_offset;
            point2_offset.angle = segment2.angle;
            point2_offset.position = sf::Vector2f(point2.position.x + offset2 * sin(segment2.angle),
                                                  point2.position.y + offset2 * -cos(segment2.angle));

            linkPoints(point1_offset, point2_offset, 2, true);
          }
        }
      }
    }
  }

  spdlog::info("Graph created with {} points", graphPoints.size());

  // Remove all the neighbors that need to turn too much
  for (auto &point : graphPoints) {
    std::vector<neighbor> newNeighbors;
    double distance;
    for (auto &neighbor : neighbors[point]) {
      double speed = turningRadiusToSpeed(CAR_MIN_TURNING_RADIUS);
      bool can = canLink(point, neighbor.point, speed, &distance);

      if (!can)
        continue;

      while (canLink(point, neighbor.point, speed + 0.1, &distance)) {
        speed += 0.1;
        if (speed >= CAR_MAX_SPEED_MS) {
          speed = CAR_MAX_SPEED_MS;
          break;
        }
      }

      if (can) {
        neighbor.maxSpeed = speed - 0.1;
        neighbor.distance = std::sqrt(std::pow(neighbor.point.position.x - point.position.x, 2) +
                                      std::pow(neighbor.point.position.y - point.position.y, 2));

        neighbor.turningRadius = turningRadius(speed);
        newNeighbors.push_back(neighbor);
      }
    }

    neighbors[point].clear();
    for (const auto &neighbor : newNeighbors) {
      neighbors[point].push_back(neighbor);
    }
  }
}

void CityGraph::linkPoints(const point &p, const point &n, int direction, bool subPoints) {
  std::vector<double> anglesPoint = {normalizeAngle(p.angle), normalizeAngle(p.angle + M_PI)};
  std::vector<double> anglesNeighbor = {normalizeAngle(n.angle), normalizeAngle(n.angle + M_PI)};

  point copyPoint = p;
  point copyNeighbor = n;

  bool isRiP = direction == 2 || direction == 0;
  bool isRiN = direction == 2 || direction == 1;
  bool isStraight = direction != 2;
  isStraight &= (anglesPoint[0] == anglesNeighbor[0] || anglesPoint[0] == anglesNeighbor[1] ||
                 anglesPoint[1] == anglesNeighbor[0] || anglesPoint[1] == anglesNeighbor[1]);
  isStraight &= subPoints;

  if (!isStraight) {
    for (const auto &anglePoint : anglesPoint) {
      for (const auto &angleNeighbor : anglesNeighbor) {
        copyPoint.angle = anglePoint;
```

```
        copyNeighbor.angle = angleNeighbor;

        neighbors[copyPoint].push_back({copyNeighbor, 0, 0, 0, isRiP}); // This fields will be updated later
        neighbors[copyNeighbor].push_back({copyPoint, 0, 0, 0, isRiN});

        graphPoints.insert(copyPoint);
        graphPoints.insert(copyNeighbor);
      }
    }
    return;
  }

  // Link adding points in the middle
  double pointDistance = 3;
  double distance = std::sqrt(std::pow(n.position.x - p.position.x, 2) + std::pow(n.position.y - p.position.y, 2));
  int numPoints = distance / pointDistance;
  double dx = (n.position.x - p.position.x) / numPoints;
  double dy = (n.position.y - p.position.y) / numPoints;

  for (const auto &anglePoint : anglesPoint) {
    for (const auto &angleNeighbor : anglesNeighbor) {
      point previousPoint = p;
      previousPoint.angle = anglePoint;

      for (int i = 1; i <= numPoints; i++) {
        point newPoint;
        newPoint.position = sf::Vector2f(p.position.x + i * dx, p.position.y + i * dy);
        newPoint.angle = anglePoint;

        neighbors[previousPoint].push_back({newPoint, 0, 0, 0, isRiP}); // This fields will be updated later
        neighbors[newPoint].push_back({previousPoint, 0, 0, 0, isRiN});

        previousPoint = newPoint;

        graphPoints.insert(newPoint);
      }

      // Add the last point
      neighbors[previousPoint].push_back({n, 0, 0, 0, isRiP}); // This fields will be updated later
    }
  }
}

CityGraph::point CityGraph::getRandomPoint() const {
  std::vector<point> graphPointsOut;
  for (const auto &point : graphPoints) {
    if (point.position.x + CAR_LENGTH < 0 || point.position.x - CAR_LENGTH > width ||
        point.position.y + CAR_LENGTH < 0 || point.position.y - CAR_LENGTH > height)
      graphPointsOut.push_back(point);
  }

  auto it = graphPointsOut.begin();
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> dis(0, graphPointsOut.size() - 1);

  std::advance(it, dis(gen));

  return *it;
}

bool CityGraph::canLink(const point &point1, const point &point2, double speed, double *distance) const {
  double radius = turningRadius(speed);

  ob::DubinsStateSpace space(radius);

  ob::State *start = space.allocState();
  ob::State *end = space.allocState();

  start->as<ob::DubinsStateSpace::StateType>()->setXY(point1.position.x, point1.position.y);
  start->as<ob::DubinsStateSpace::StateType>()->setYaw(point1.angle);

  end->as<ob::DubinsStateSpace::StateType>()->setXY(point2.position.x, point2.position.y);
  end->as<ob::DubinsStateSpace::StateType>()->setYaw(point2.angle);

  double total = 0;

  // Extract the path
  ob::DubinsStateSpace::DubinsPath path = space.dubins(start, end);
  for (unsigned int i = 0; i < 3; ++i) // Dubins path has up to 3 segments
  {
    auto type = path.type_[i];
    if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_LEFT) {
```

```
      total += std::abs(path.length_[i]);
    } else if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_RIGHT) {
      total += std::abs(path.length_[i]);
    }
  }

  *distance = space.distance(start, end);
  return total < M_PI * 0.75f;
}
```

## 4.33 cityMap.cpp File Reference

CityMap class implementation.

```
#include <iostream>
#include <math.h>
#include <set>
#include "spdlog/spdlog.h"
#include "tinyxml2.h"
#include "cityMap.h"
#include "utils.h"
```

### 4.33.1 Detailed Description

CityMap class implementation.
This file contains the implementation of the CityMap class.
Definition in file cityMap.cpp.

## 4.34 cityMap.cpp

Go to the documentation of this file.

```
#include <iostream>
#include <math.h>
#include <set>

#include "spdlog/spdlog.h"
#include "tinyxml2.h"

#include "cityMap.h"
#include "utils.h"

CityMap::CityMap() {
  roads.clear();
  intersections.clear();
  minLatLon.x = minLatLon.y = maxLatLon.x = maxLatLon.y = 0;
}

void CityMap::loadFile(const std::string &filename) {
  spdlog::info("Loading file: {}", filename);

  tinyxml2::XMLDocument doc;
  // Load the XML file
  if (doc.LoadFile(filename.c_str()) != tinyxml2::XML_SUCCESS) {
    spdlog::error("Failed to load file: {}", filename);
    return;
  }

  // Extract the bounds of the map
  tinyxml2::XMLElement *bounds = doc.FirstChildElement("osm")->FirstChildElement("bounds");
  if (!bounds) {
    spdlog::error("Failed to extract bounds from file: {}", filename);
    return;
  }

  minLatLon.x = bounds->FloatAttribute("minlon");
  minLatLon.y = bounds->FloatAttribute("minlat");
  maxLatLon.x = bounds->FloatAttribute("maxlon");
  maxLatLon.y = bounds->FloatAttribute("maxlat");

  // Define the width and height of the map
  width = latLonToXY(minLatLon.y, minLatLon.x).x - latLonToXY(maxLatLon.y, maxLatLon.x).x;
  height = latLonToXY(minLatLon.y, minLatLon.x).y - latLonToXY(maxLatLon.y, maxLatLon.x).y;
  width = std::abs(width);
  height = std::abs(height);

```

```
  std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
  spdlog::info("Loading roads and buildings ...");

  // List of highway types to exclude
  std::set<std::string> excludedHighways = {"footway", "path",  "pedestrian", "cycleway",
                                            "steps",   "track", "bridleway",  "service"};

  // List of highway types to include
  std::set<std::string> includedHighways = {
      "motorway",      "trunk",         "primary",    "secondary",    "tertiary",
"unclassified", "residential",
      "living_street", "motorway_link", "trunk_link", "primary_link", "secondary_link",
"tertiary_link"};

  // Extract the roads
  tinyxml2::XMLElement *way = doc.FirstChildElement("osm")->FirstChildElement("way");
  int roadId = 0;
  while (way) {
    road r;
    building b;
    greenArea g;
    waterArea w;
    r.width = DEFAULT_ROAD_WIDTH;
    r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    r.id = roadId;

    tinyxml2::XMLElement *nd = way->FirstChildElement("nd");
    while (nd) {
      tinyxml2::XMLElement *node = doc.FirstChildElement("osm")->FirstChildElement("node");
      while (node) {
        if (node->IntAttribute("id") == nd->IntAttribute("ref")) {
          sf::Vector2f p;
          p.x = node->FloatAttribute("lon");
          p.y = node->FloatAttribute("lat");

          if (r.segments.size() > 0) {
            segment s;
            s.p1 = r.segments.back().p2;
            s.p2 = p;
            r.segments.push_back(s);
          } else {
            segment s;
            s.p1 = p;
            s.p2 = p;
            r.segments.push_back(s);
          }

          b.points.push_back(p);
          g.points.push_back(p);
          w.points.push_back(p);
          break;
        }
        node = node->NextSiblingElement("node");
      }
      nd = nd->NextSiblingElement("nd");
    }

    // Remove the first segment (it has the same p1 and p2)
    r.segments.erase(r.segments.begin());

    std::string highwayType;
    bool isHighway = false;
    bool isBuilding = false;
    bool isUnderground = false;
    bool isGreenArea = false;
    bool isWaterArea = false;
    bool widthSet = false;
    bool lanesSet = false;
    tinyxml2::XMLElement *tag = way->FirstChildElement("tag");
    while (tag) {
      if (strcmp(tag->Attribute("k"), "width") == 0) {
        r.width = tag->FloatAttribute("v");
        widthSet = true;
      } else if (strcmp(tag->Attribute("k"), "lanes") == 0) {
        r.numLanes = tag->IntAttribute("v");
        lanesSet = true;
      } else if (strcmp(tag->Attribute("k"), "highway") == 0) {
        highwayType = tag->Attribute("v");
        isHighway = true;
      } else if (strcmp(tag->Attribute("k"), "building") == 0) {
        isBuilding = true;
      } else if (strcmp(tag->Attribute("k"), "layer") == 0) {
        int layerValue = tag->IntAttribute("v");
        if (layerValue < 0) {
          isUnderground = true;
        }
      } else if (strcmp(tag->Attribute("k"), "landuse") == 0) {
```

```
        if (strcmp(tag->Attribute("v"), "forest") == 0 || strcmp(tag->Attribute("v"), "grass") == 0 ||
            strcmp(tag->Attribute("v"), "meadow") == 0) {
          isGreenArea = true;
          g.type = 0;
        }
      } else if (strcmp(tag->Attribute("k"), "leisure") == 0) {
        if (strcmp(tag->Attribute("v"), "park") == 0 || strcmp(tag->Attribute("v"), "garden") == 0) {
          isGreenArea = true;
          g.type = 1;
        }
      } else if (strcmp(tag->Attribute("k"), "waterway") == 0 &&
                 (strcmp(tag->Attribute("v"), "river") == 0 || strcmp(tag->Attribute("v"), "stream") == 0 ||
                  strcmp(tag->Attribute("v"), "canal") == 0)) {
        isWaterArea = true;
      } else if (strcmp(tag->Attribute("k"), "natural") == 0 &&
                 (strcmp(tag->Attribute("v"), "water") == 0 || strcmp(tag->Attribute("v"), "wetland") == 0)) {
        isWaterArea = true;
      } else if (strcmp(tag->Attribute("k"), "water") == 0 &&
                 (strcmp(tag->Attribute("v"), "lake") == 0 || strcmp(tag->Attribute("v"), "pond") == 0 ||
                  strcmp(tag->Attribute("v"), "river") == 0)) {
        isWaterArea = true;
      }
      tag = tag->NextSiblingElement("tag");
    }
    if (!widthSet && !lanesSet) {
      r.width = DEFAULT_ROAD_WIDTH;
      r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    } else if (!widthSet) {
      r.width = r.numLanes * DEFAULT_LANE_WIDTH;
    } else if (!lanesSet) {
      r.numLanes = r.width / DEFAULT_LANE_WIDTH;
    }
    r.width = std::max(r.width, MIN_ROAD_WIDTH);
    r.numLanes = std::max(r.numLanes, 1);

    if (isUnderground) {
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isBuilding) {
      buildings.push_back(b);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isGreenArea) {
      greenAreas.push_back(g);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (isWaterArea) {
      waterAreas.push_back(w);
      way = way->NextSiblingElement("way");
      continue;
    }
    if (!isHighway || excludedHighways.find(highwayType) != excludedHighways.end()) {
      way = way->NextSiblingElement("way");
      continue;
    }
    if (includedHighways.find(highwayType) != includedHighways.end()) {
      roads.push_back(r);
      roadId++;
    }

    way = way->NextSiblingElement("way");
  }

  // Convert lat/lon to meters (using the upper-left corner as origin)
  sf::Vector2f minXY = latLonToXY(minLatLon.y, minLatLon.x);
  sf::Vector2f maxXY = latLonToXY(maxLatLon.y, maxLatLon.x);
  for (auto &r : roads) {
    for (auto &s : r.segments) {
      s.p1 = latLonToXY(s.p1.y, s.p1.x);
      s.p2 = latLonToXY(s.p2.y, s.p2.x);

      s.p1.x -= minXY.x;
      s.p1.y -= minXY.y;
      s.p2.x -= minXY.x;
      s.p2.y -= minXY.y;

      // Symetri to the x-axis
      s.p1.y = maxXY.y - minXY.y - s.p1.y;
      s.p2.y = maxXY.y - minXY.y - s.p2.y;

      s.p1_offset = s.p1;
```

```
      s.p2_offset = s.p2;

      s.angle = std::atan2(s.p2.y - s.p1.y, s.p2.x - s.p1.x);
    }
  }
  for (auto &b : buildings) {
    for (auto &p : b.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }
  for (auto &g : greenAreas) {
    for (auto &p : g.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }
  for (auto &w : waterAreas) {
    for (auto &p : w.points) {
      p = latLonToXY(p.y, p.x);

      p.x -= minXY.x;
      p.y -= minXY.y;

      // Symetri to the x-axis
      p.y = maxXY.y - minXY.y - p.y;
    }
  }

  std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
  spdlog::info("Roads and buildings loaded ({} ms)",
               std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count());

  spdlog::info("Loading intersections ...");

  // Intersections are at any roads' points if they are near another one
  // First add the intersections for each node point
  // Then merge the intersections that are close to each other
  intersections.clear();
  int intersectionId = 0;

  // Add the intersections for each road segment
  spdlog::debug("Adding intersections ...");
  for (auto r : roads) {
    for (int s_id = 0; s_id < (int)r.segments.size(); s_id++) {
      segment s = r.segments[s_id];
      std::vector<sf::Vector2f> points = {s.p1, s.p2};
      for (auto p : points) {
        intersection i = {intersectionId++, p, r.width / 2, {}};
        i.roadSegmentIds.push_back({r.id, s_id});
        intersections.push_back(i);
      }
    }
  }
  spdlog::debug("Intersections added");

  // Merge the intersections that are close to each other
  spdlog::debug("Merging intersections ...");
  for (int distCoef = 5; distCoef > 0; distCoef -= 1) {
    for (int i = 0; i < (int)intersections.size(); i++) {
      for (int j = i + 1; j < (int)intersections.size(); j++) {
        bool is_i = intersections[i].roadSegmentIds.size() > intersections[j].roadSegmentIds.size();

        if (intersections[i].roadSegmentIds.size() == intersections[j].roadSegmentIds.size()) {
          is_i = intersections[i].id < intersections[j].id;
        }

        double minSpace = intersections[i].radius + intersections[j].radius;
        minSpace /= distCoef;

        if (distance(intersections[i].center, intersections[j].center) < minSpace) {
          // Merge the intersections to i or j (depending on is_i)
          int index_from = is_i ? j : i;
          int index_to = is_i ? i : j;

          for (auto &r : intersections[index_from].roadSegmentIds) {
            intersections[index_to].roadSegmentIds.push_back(r);
```

```
          }

          intersections.erase(intersections.begin() + index_from);
          i -= 1;
          break;
        }
      }
    }
  }
  spdlog::debug("Intersections merged");

  // Make the road point to be outside the intersection
  spdlog::debug("Adding offsets to the roads ...");
  for (auto &i : intersections) {
    for (auto &roadInfo : i.roadSegmentIds) {
      double dx =
          roads[roadInfo.first].segments[roadInfo.second].p2.x - roads[roadInfo.first].segments[roadInfo.second].p1.x;
      double dy =
          roads[roadInfo.first].segments[roadInfo.second].p2.y - roads[roadInfo.first].segments[roadInfo.second].p1.y;
      double dd = distance({0, 0}, {(float)dx, (float)dy});
      dx /= dd;
      dy /= dd;

      double radius = i.radius;

      if (distance(roads[roadInfo.first].segments[roadInfo.second].p1, i.center) <
          distance(roads[roadInfo.first].segments[roadInfo.second].p2, i.center)) {
        roads[roadInfo.first].segments[roadInfo.second].p1_offset.x = i.center.x + dx * radius;
        roads[roadInfo.first].segments[roadInfo.second].p1_offset.y = i.center.y + dy * radius;
      } else {
        dx = -dx;
        dy = -dy;
        roads[roadInfo.first].segments[roadInfo.second].p2_offset.x = i.center.x + dx * radius;
        roads[roadInfo.first].segments[roadInfo.second].p2_offset.y = i.center.y + dy * radius;
      }
    }
  }
  spdlog::debug("Offsets added");

  // Remove the intersections that link the same road
  spdlog::debug("Removing intersections that link the same road ...");
  for (int i = 0; i < (int)intersections.size(); i++) {
    if (intersections[i].roadSegmentIds.size() != 2)
      continue;

    if (intersections[i].roadSegmentIds[0].first == intersections[i].roadSegmentIds[1].first) {
      intersections.erase(intersections.begin() + i);
      i -= 1;
    }
  }
  spdlog::debug("Intersections removed");

  // Log all the intersections and roads
  for (auto r : roads) {
    spdlog::debug("Road: id={}, width={}, numLanes={}, segments={}", r.id, r.width, r.numLanes, r.segments.size());
  }
  for (auto i : intersections) {
    spdlog::debug("Intersection: id={}, center=({}, {}), radius={}, roadSegmentIds={}", i.id, i.center.x, i.center.y,
                  i.radius, i.roadSegmentIds.size());
  }

  std::chrono::steady_clock::time_point end2 = std::chrono::steady_clock::now();
  spdlog::info("Intersections loaded ({} ms)",
               std::chrono::duration_cast<std::chrono::milliseconds>(end2 - end).count());

  spdlog::info("Number of roads: {}", roads.size());
  spdlog::info("Number of buildings: {}", buildings.size());
  spdlog::info("Number of intersections: {}", intersections.size());

  spdlog::info("Width: {} m", width);
  spdlog::info("Height: {} m", height);

  isLoaded = true;
}
```

## 4.35 constraintController.cpp File Reference

ConstraintController class implementation.

```
#include <iostream>
#include <spdlog/spdlog.h>
```

```
#include "aStar.h"
#include "dubins.h"
```

### 4.35.1 Detailed Description

ConstraintController class implementation.
This file contains the implementation of the ConstraintController class. This class is used to control the constraints of the A∗ algorithm. It is used to check if a car can move to a certain point in the graph at a certain time.
Definition in file constraintController.cpp.

## 4.36 constraintController.cpp

Go to the documentation of this file.

```

#include <iostream>
#include <spdlog/spdlog.h>

#include "aStar.h"
#include "dubins.h"

void ConstraintController::addConstraint(AStar::conflict constraint) {
  if (constraint.car < 0) {
    spdlog::error("Invalid car index for constraint");
    throw std::runtime_error("Invalid car index for constraint");
  }

  while (constraints.size() <= constraint.car) {
    constraints.push_back(std::vector<std::vector<AStar::conflict»());
  }

  while (constraints[constraint.car].size() <= constraint.time) {
    constraints[constraint.car].push_back(std::vector<AStar::conflict>());
  }

  constraints[constraint.car][constraint.time].push_back(constraint);
}

bool ConstraintController::hasConstraint(AStar::conflict constraint) {
  if (constraint.car < 0) {
    spdlog::error("Invalid car index for constraint");
    throw std::runtime_error("Invalid car index for constraint");
  }
  if (constraints.size() <= constraint.car) {
    return false;
  }
  for (int t = std::max(0, constraint.time - 1);
       t < std::min((int)constraints[constraint.car].size(), constraint.time + 1); t++) {
    for (auto c : constraints[constraint.car][t]) {
      if (c == constraint) {
        return true;
      }
    }
  }

  return false;
}

ConstraintController ConstraintController::copy() {
  ConstraintController cc;

  cc.constraints = constraints;

  return cc;
}

ConstraintController ConstraintController::copy(std::vector<int> cars) {
  ConstraintController cc;

  for (int car : cars) {
    if (constraints.size() > car) {
      cc.constraints.push_back(constraints[car]);
    } else {
      cc.constraints.push_back(std::vector<std::vector<AStar::conflict»());
    }
  }

  return cc;
}

```

```
bool ConstraintController::checkConstraints(int car, double speed, double newSpeed, double time, CityGraph::point from,
                                             CityGraph::neighbor to) {

  Dubins dubin = Dubins(from, to, speed, newSpeed);
  float duration = 2 * to.distance / (speed + newSpeed);

  int tMin = std::round(time / SIM_STEP_TIME);
  tMin = tMin < 0 ? 0 : tMin;
  int tMax = std::round((time + duration) / SIM_STEP_TIME);
  if (constraints.size() > car && constraints[car].size() < tMax) {
    tMax = constraints[car].size();
  }

  if (constraints.size() <= car || constraints[car].size() <= tMin) {
    return false;
  }

  double distance = to.distance;
  double acc = (std::pow(newSpeed, 2) - std::pow(speed, 2)) / (2 * distance);
  auto xFun = [&](double t) { return (0.5 * acc * t * t + speed * t) / distance; };

  for (int t = tMin; t < tMax; t += 1) {
    bool precise = false;
    sf::Vector2f pos = from.position + (to.point.position - from.position) * (float)xFun(t * SIM_STEP_TIME - time);
    CityGraph::point point;
    if (constraints[car].size() <= t)
      continue;

    for (auto c : constraints[car][t]) {
      if (precise) {
        if (carConflict(point.position, point.angle, c.point.position, c.point.angle)) {
          return true;
        }
      } else {
        sf::Vector2f diff = pos - c.point.position;
        double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
        if (dist < 2 * CAR_LENGTH) {
          precise = true;
          point = dubin.point(t * SIM_STEP_TIME - time);
          if (carConflict(point.position, point.angle, c.point.position, c.point.angle)) {
            return true;
          }
        }
      }
    }
  }

  return false;
}
```

## 4.37 dataManager.cpp File Reference

Data manager.

```
#include <filesystem>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <spdlog/spdlog.h>
#include "cityGraph.h"
#include "cityMap.h"
#include "dataManager.h"
#include "manager.h"
```

### 4.37.1 Detailed Description

Data manager.
This file contains the implementation of the DataManager class.
Definition in file dataManager.cpp.

## 4.38 dataManager.cpp

Go to the documentation of this file.

```

#include <filesystem>
#include <fstream>
#include <iostream>
#include <random>
#include <string>

#include <spdlog/spdlog.h>

#include "cityGraph.h"
#include "cityMap.h"
#include "dataManager.h"
#include "manager.h"

DataManager::DataManager(std::string filename) {
  // Create /data folder if it doesn’t exist
  if (!std::filesystem::exists("data")) {
    spdlog::debug("Creating data folder");
    std::filesystem::create_directory("data");
  }
}

void DataManager::createData(int numData, int numCarsMin, int numCarsMax, std::string mapName) {
  // If numData is less than 1, default to a very high number (as in your original code).
  numData = numData < 1 ? INT_MAX : numData;

  // Remove file extension from mapName to construct the output filename.
  std::string mapNameNoExt = mapName.substr(0, mapName.find_last_of("."));
  std::string filename = "data/" + mapNameNoExt + "_" + std::to_string((int)CBS_MAX_SUB_TIME) +
                         (ROAD_ENABLE_RIGHT_HAND_TRAFFIC ? "_RHT" : "") + "_data.csv";

  // Load the city map.
  CityMap cityMap;
  cityMap.loadFile("assets/map/" + mapName);

  // Create the city graph.
  CityGraph cityGraph;
  cityGraph.createGraph(cityMap);

  // Open the output file in append mode.
  std::ofstream file;
  file.open(filename, std::ios::app);
  if (!file.is_open()) {
    spdlog::error("Failed to open file {}", filename);
    return;
  }

  std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
  std::uniform_int_distribution<int> dist(numCarsMin, numCarsMax);

  for (int i = 0; i < numData; i += 1) {
    int numCars = dist(rng);

    Manager manager(cityGraph, cityMap, false);
    auto resData = manager.createCarsCBS(numCars);
    if (!resData.first) {
      spdlog::warn("Data {}: CBS failed (numCars: {})", i + 1, numCars);
      i--;
      continue;
    }

    data validResData = resData.second;

    file « validResData.numCars « ";" « validResData.carDensity;
    for (auto speed : validResData.carAvgSpeed) {
      file « ";" « speed;
    }
    file « std::endl;

    if (numData == INT_MAX) {
      spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, validResData.numCars, validResData.carDensity);
    } else {
      spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, numData, validResData.numCars,
                   validResData.carDensity);
    }
  }

  file.close();
}
```

## 4.39 dubins.cpp File Reference

Dubins path implementation.

```
#include "dubins.h"
#include "utils.h"
```

### 4.39.1 Detailed Description

Dubins path implementation.

This file contains the implementation of the Dubins class. It is used to calculate the path between two points in the city graph. It will be used to render cars in the city and check for collisions.

Definition in file dubins.cpp.

## 4.40 dubins.cpp

Go to the documentation of this file.

```

#include "dubins.h"
#include "utils.h"

Dubins::Dubins(CityGraph::point start, CityGraph::neighbor end)
    : Dubins(start, end, CAR_MAX_SPEED_MS, CAR_MAX_SPEED_MS) {}

Dubins::Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed)
    : Dubins(start, end, startSpeed, CAR_MAX_SPEED_MS) {

  // The distance needed to reach the maximum speed
  double distanceToMaxSpeed = (std::pow(CAR_MAX_SPEED_MS, 2) - std::pow(startSpeed, 2)) / (2 * CAR_ACCELERATION);

  double dist = distance();
  if (dist == 0) {
    this->avgSpeed = CAR_MAX_SPEED_MS;
    return;
  }

  if (dist < distanceToMaxSpeed) {
    this->avgSpeed = CAR_MAX_SPEED_MS;
  } else {
    double avg = (startSpeed + CAR_MAX_SPEED_MS) / 2;
    this->avgSpeed = (avg * distanceToMaxSpeed + CAR_MAX_SPEED_MS * (dist - distanceToMaxSpeed)) / dist;
  }
}

Dubins::Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed, double endSpeed) {
  this->startPoint = start;
  this->endPoint = end;
  this->startSpeed = startSpeed;
  this->endSpeed = endSpeed;
  this->avgSpeed = (startSpeed + endSpeed) / 2;

  this->space = new ob::DubinsStateSpace(this->endPoint.turningRadius);

  ob::RealVectorBounds bounds(2);
  space->setBounds(bounds);

  this->start = space->allocState();
  this->end = space->allocState();

  this->start->as<ob::DubinsStateSpace::StateType>()->setXY(start.position.x, start.position.y);
  this->start->as<ob::DubinsStateSpace::StateType>()->setYaw(start.angle);

  this->end->as<ob::DubinsStateSpace::StateType>()->setXY(end.point.position.x, end.point.position.y);
  this->end->as<ob::DubinsStateSpace::StateType>()->setYaw(end.point.angle);
}

Dubins::~Dubins() {
  space->freeState(start);
  space->freeState(end);
  delete space;
}

double Dubins::time() { return this->distance() / avgSpeed; }

CityGraph::point Dubins::point(double time) {
  double distance = this->distance();
  double acc = (std::pow(endSpeed, 2) - std::pow(startSpeed, 2)) / (2 * distance);
```

```
  auto xFun = [distance, acc, this](double t) { return (0.5 * acc * t * t + this->startSpeed * t) / distance; };

  ob::State *state = space->allocState();
  space->interpolate(start, end, xFun(time), state);

  double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
  double y = state->as<ob::DubinsStateSpace::StateType>()->getY();
  double yaw = state->as<ob::DubinsStateSpace::StateType>()->getYaw();

  space->freeState(state);

  CityGraph::point point;
  point.position = {(float)x, (float)y};
  point.angle = yaw;

  return point;
}

std::vector<CityGraph::point> Dubins::path() {
  std::vector<CityGraph::point> path;
  double time = this->time();
  for (double t = 0; t < time; t += SIM_STEP_TIME) {
    path.push_back(this->point(t));
  }

  return path;
}

DubinsPath::DubinsPath(std::vector<AStar::node> path) : path_(path) {}

std::vector<CityGraph::point> DubinsPath::path() {
  if (pathProcessed_.empty())
    process();

  return pathProcessed_;
}

void DubinsPath::process() {
  pathProcessed_.clear();
  double t = 0;
  double prevTime = 0;

  for (int i = 1; i < (int)path_.size(); i++) {
    AStar::node prevNode = path_[i - 1];
    AStar::node node = path_[i];

    CityGraph::point start = node.arcFrom.first;
    CityGraph::neighbor end = node.arcFrom.second;

    Dubins dubins(start, end, prevNode.speed, node.speed);
    double time = dubins.time();

    if (t >= prevTime + time) {
      continue;
    }

    while (t < prevTime + time) {
      pathProcessed_.push_back(dubins.point(t - prevTime));
      t += SIM_STEP_TIME;
    }

    prevTime += time;
  }
}
```

## 4.41 fileSelector.cpp File Reference

File selector implementation.

```
#include "fileSelector.h"
#include <filesystem>
#include <iostream>
#include <spdlog/spdlog.h>
#include <termios.h>
#include <unistd.h>
#include <vector>
```

### 4.41.1 Detailed Description

File selector implementation.

This file contains the implementation of the FileSelector class. It is used to select a file from a folder.

Definition in file fileSelector.cpp.

## 4.42 fileSelector.cpp

Go to the documentation of this file.

```
#include "fileSelector.h"

#include <filesystem>
#include <iostream>
#include <spdlog/spdlog.h>
#include <termios.h>
#include <unistd.h>
#include <vector>

namespace fs = std::filesystem;

void FileSelector::loadFiles() {
  files.clear();
  for (const auto &entry : fs::directory_iterator(folderPath)) {
    if (entry.is_regular_file() && entry.path().extension() == ".osm") {
      files.push_back(entry.path().filename().string());
    }
  }
  std::sort(files.begin(), files.end());
}

char FileSelector::getKeyPress() {
  struct termios oldt, newt;
  char ch;
  tcgetattr(STDIN_FILENO, &oldt);
  newt = oldt;
  newt.c_lflag &= ~(ICANON | ECHO);
  tcsetattr(STDIN_FILENO, TCSANOW, &newt);
  ch = getchar();
  tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
  return ch;
}

void FileSelector::moveCursorUp() {
  if (selectedIndex > 0) {
    std::cout « "\033[2K\r  " « files[selectedIndex] « std::flush;
    selectedIndex--;
    std::cout « "\033[A\033[2K\r> " « files[selectedIndex] « std::flush;
  }
}

void FileSelector::moveCursorDown() {
  if (selectedIndex < files.size() - 1) {
    std::cout « "\033[2K\r  " « files[selectedIndex] « std::flush;
    selectedIndex++;
    std::cout « "\033[B\033[2K\r> " « files[selectedIndex] « std::flush;
  }
}

void FileSelector::displayFiles() {
  std::cout « "Use UP/DOWN arrow keys to navigate, ENTER to select:\n";
  for (size_t i = 0; i < files.size(); i++) {
    if (i == selectedIndex) {
      std::cout « "> " « files[i] « "\n";
    } else {
      std::cout « "  " « files[i] « "\n";
    }
  }
  std::cout « "\033[" « files.size() « "A";
}

std::string FileSelector::selectFile() {
  std::cout « "\033[?25l";
  if (files.empty()) {
    spdlog::error("No .osm files found in the folder: {}", folderPath);
    return "";
  }

  displayFiles();

  while (true) {
    char key = getKeyPress();
    if (key == 27) {
```

```cpp
      if (getKeyPress() == '[') {
        switch (getKeyPress()) {
        case 'A':
          moveCursorUp();
          break;
        case 'B':
          moveCursorDown();
          break;
        }
      }
    } else if (key == '\n') {
      std::cout « "\033[" « selectedIndex + 1 « "A\033[2K\r" « std::flush;
      std::cout « "\033[?25h";
      spdlog::info("Selected file: {}", files[selectedIndex]);
      return files[selectedIndex];
    }
  }
}
```

## 4.43 main.cpp File Reference

Main file.

```cpp
#include "spdlog/spdlog.h"
#include <SFML/Graphics.hpp>
#include "cityMap.h"
#include "config.h"
#include "dataManager.h"
#include "fileSelector.h"
#include "manager.h"
#include "renderer.h"
#include "test.h"
```

### 4.43.1 Detailed Description

Main file.

This file contains the main function of the project. It is used to run the simulation and create data.

Definition in file main.cpp.

## 4.44 main.cpp

Go to the documentation of this file.

```cpp

#include "spdlog/spdlog.h"
#include <SFML/Graphics.hpp>

#include "cityMap.h"
#include "config.h"
#include "dataManager.h"
#include "fileSelector.h"
#include "manager.h"
#include "renderer.h"
#include "test.h"

int main(int nArgs, char **args) {
  srand(time(NULL));
  spdlog::set_pattern("[%d-%m-%C %H:%M:%S.%e] [%^%l%$] [thread %t] %v");

  if (nArgs < 1) {
    spdlog::error("Usage: {} \"data\" [numCarsMin] [numCarsMax] [numData] || {} \"run\" [numCars]",
args[0]);
    return 1;
  }

  bool data = args[1] == std::string("data");
  int runNumCars = 10;
  int dataNumCarsMin = 10;
  int dataNumCarsMax = 15;
  int dataNumData = -1;

  if (nArgs > 2) {
    runNumCars = std::stoi(args[2]);
    dataNumCarsMin = std::stoi(args[2]);
  }
  if (nArgs > 3) {
```

```
    dataNumCarsMax = std::stoi(args[3]);
  }
  if (nArgs > 4) {
    dataNumData = std::stoi(args[4]);
  }

  FileSelector fileSelector("assets/map");
  std::string mapFile = fileSelector.selectFile();
  // std::string mapFile = "small01.osm";

  if (ENVIRONMENT == 0 && false) {
    spdlog::set_level(spdlog::level::debug);
    Test test;
    test.runTests();
  } else {
    spdlog::set_level(spdlog::level::info);
  }

  if (data) {
    spdlog::info("Creating data for map {}, numData: {}, numCarsMin: {}, numCarsMax: {}", mapFile,
dataNumData,
                 dataNumCarsMin, dataNumCarsMax);

    DataManager dataManager(mapFile);
    dataManager.createData(dataNumData, dataNumCarsMin, dataNumCarsMax, mapFile);
  } else {
    spdlog::info("Running simulation for map {}, numCars: {}", mapFile, runNumCars);

    CityMap cityMap;
    cityMap.loadFile("assets/map/" + mapFile);

    CityGraph cityGraph;
    cityGraph.createGraph(cityMap);

    Manager manager(cityGraph, cityMap, true);
    manager.createCarsCBS(runNumCars);

    Renderer renderer;
    renderer.startRender(cityMap, cityGraph, manager);
  }

  return 0;
}
```

## 4.45 manager.cpp File Reference

Implementation of the Manager class.

```
#include "manager.h"
#include "aStar.h"
#include <iostream>
#include <spdlog/spdlog.h>
```

### 4.45.1 Detailed Description

Implementation of the Manager class.

This file contains the implementation of the Manager class.

Definition in file manager.cpp.

## 4.46 manager.cpp

Go to the documentation of this file.

```

#include "manager.h"
#include "aStar.h"

#include <iostream>
#include <spdlog/spdlog.h>

void Manager::createCarsAStar(int numCars) {
  if (log)
    spdlog::info("Creating {} AStar cars", numCars);
  for (int i = 0; i < numCars; i++) {
    Car car;
    cars.push_back(car);
  }

```

```
  // Create a path for each car (random start and end points)
  for (int i = 0; i < numCars; i++) {
    bool valid = false;
    cars[i].chooseRandomStartEndPath(graph, map);

    if (log)
      spdlog::info("Car {} assigned path with {} points", i, cars[i].getPath().size());
  }
}

void Manager::moveCars() {
  for (Car &car : cars) {
    car.move();
  }
}

void Manager::renderCars(sf::RenderWindow &window) {
  for (Car &car : cars) {
    car.render(window);
  }
}

void Manager::toggleCarDebug(sf::Vector2f mousePos) {
  for (Car &car : cars) {
    sf::Vector2f carPos = car.getPosition();
    double distance = sqrt(pow(mousePos.x - carPos.x, 2) + pow(mousePos.y - carPos.y, 2));
    if (distance < 5.0f) {
      car.toggleDebug();
    }
  }
}
```

## 4.47 managerCBS.cpp File Reference

CBS algorithm implementation.

```
#include "manager.h"
#include "priorityQueue.h"
#include "renderer.h"
#include "utils.h"
#include <iostream>
#include <numeric>
#include <spdlog/spdlog.h>
```

### 4.47.1 Detailed Description

CBS algorithm implementation.

This file contains the implementation of the CBS algorithm. It is used to resolve conflicts between cars.

Definition in file managerCBS.cpp.

## 4.48 managerCBS.cpp

Go to the documentation of this file.

```

#include "manager.h"
#include "priorityQueue.h"
#include "renderer.h"
#include "utils.h"

#include <iostream>
#include <numeric>
#include <spdlog/spdlog.h>

std::pair<bool, DataManager::data> Manager::createCarsCBS(int numCars) {
  this->createCarsAStar(numCars);
  this->numCars = numCars;
  bool valid = true;

  ConstraintController constraints;

  if (log)
    spdlog::info("Creating {} CBS cars", numCars);

  CBSNode node = processCBS(constraints, 0);
  if (!node.hasResolved) {
    if (log)
```

```
      spdlog::error("CBS could not resolve all conflicts");
    return std::make_pair(false, DataManager::data());
  } else {
    if (log)
      spdlog::info("CBS resolved all conflicts");
  }

  // Check if conflicts remain
  for (int i = 0; i < numCars; i++) {
    for (int j = i + 1; j < numCars; j++) {
      int tMin = std::min(cars[i].getPath().size(), cars[j].getPath().size());
      for (int t = 0; t < tMin; t++) {
        sf::Vector2f diff = cars[i].getPath()[t] - cars[j].getPath()[t];

        double width = graph.getWidth();
        double height = graph.getHeight();
        auto outOfBounds = [&](sf::Vector2f p) {
          return p.x + CAR_LENGTH < 0 || p.y + CAR_LENGTH < 0 || p.x > width + CAR_LENGTH || p.y > height + CAR_LENGTH;
        };

        if (outOfBounds(cars[i].getPath()[t]) || outOfBounds(cars[j].getPath()[t])) {
          continue;
        }

        if (std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1) {
          if (log)
            spdlog::error("Cars {} and {} still have a conflict at time {} ({}, {})", i, j, t * SIM_STEP_TIME,
                          cars[i].getPath()[t].x, cars[i].getPath()[t].y);
          valid = false;
        }
      }
    }
  }

  if (!valid) {
    return std::make_pair(false, DataManager::data());
  }

  DataManager::data data;
  data.numCars = 0;
  data.carAvgSpeed.clear();

  for (int i = 0; i < numCars; i++) {
    double avgSpeed = cars[i].getAverageSpeed(graph);
    if (avgSpeed <= 0.01)
      continue;

    data.carAvgSpeed.push_back(avgSpeed);
    data.numCars++;
  }

  if (data.numCars == 0) {
    return std::make_pair(false, DataManager::data());
  }

  data.carDensity = 1000000 * data.numCars / (graph.getWidth() * graph.getHeight());

  return std::make_pair(true, data);
}

// Split the node into 2 subnodes
Manager::CBSNode Manager::createSubCBS(CBSNode &node, int subNodeDepth) {
  int numCars = (int)node.paths.size();
  int numCars1 = numCars / 2;
  int numCars2 = numCars - numCars1;

  std::vector<Car> cars1;
  std::vector<Car> cars2;

  std::vector<int> cars1Index;
  std::vector<int> cars2Index;

  for (int i = 0; i < numCars1; i++) {
    cars1.push_back(cars[i]);
    cars1Index.push_back(i);
  }
  for (int i = numCars1; i < numCars; i++) {
    cars2.push_back(cars[i]);
    cars2Index.push_back(i);
  }

  ConstraintController constraints1 = node.constraints.copy(cars1Index);
  ConstraintController constraints2 = node.constraints.copy(cars2Index);

  Manager manager1(graph, map, cars1, log);
```

```
  Manager manager2(graph, map, cars2, log);

  CBSNode node1 = manager1.processCBS(constraints1, subNodeDepth + 1);
  if (!node1.hasResolved) {
    return node1;
  }

  // Push all manager1 cars pos to manager2 constraints
  for (int i = 0; i < numCars1; i++) {
    std::vector<sf::Vector2f> path = node1.paths[i];
    for (int j = 0; j < (int)path.size(); j += CBS_PRECISION_FACTOR) {
      AStar::conflict conflict;
      conflict.point.position = path[j];
      conflict.point.angle = 0;
      conflict.time = j;

      if (conflict.point.position.x < -CAR_LENGTH || conflict.point.position.y < -CAR_LENGTH ||
          conflict.point.position.x > graph.getWidth() + CAR_LENGTH ||
          conflict.point.position.y > graph.getHeight() + CAR_LENGTH) {
        continue;
      }

      for (int k = 0; k < numCars2; k++) {
        conflict.car = k;
        constraints2.addConstraint(conflict);
      }
    }
  }

  CBSNode node2 = manager2.processCBS(constraints2, subNodeDepth + 1);
  if (!node2.hasResolved) {
    return node2;
  }

  // Merge the 2 managers
  for (int i = 0; i < numCars1; i++) {
    node.costs[i] = node1.costs[i];
    node.paths[i] = node1.paths[i];
    cars[i].assignExistingPath(node1.paths[i]);
  }
  for (int i = numCars1; i < numCars; i++) {
    node.costs[i] = node2.costs[i - numCars1];
    node.paths[i] = node2.paths[i - numCars1];
    cars[i].assignExistingPath(node2.paths[i - numCars1]);
  }

  node.cost = node1.cost + node2.cost;
  node.depth = std::max(node1.depth, node2.depth);
  node.hasResolved = node1.hasResolved && node2.hasResolved;

  return node;
}

Manager::CBSNode Manager::processCBS(ConstraintController constraints, int subNodeDepth) {
  PriorityQueue<CBSNode> openSet = PriorityQueue<CBSNode>(CBS_MAX_OPENSET_SIZE);

  CBSNode startNode;
  startNode.paths.resize(numCars);
  startNode.constraints = constraints;
  startNode.costs.clear();
  startNode.costs.resize(numCars);
  startNode.cost = 0;
  startNode.depth = 0;
  startNode.hasResolved = false;

  double maxCarCost = 0;

  for (int i = 0; i < numCars; i++) {
    TimedAStar aStar(cars[i].getStart(), cars[i].getEnd(), graph, &constraints, i);
    std::vector<AStar::node> newPath = aStar.findPath();

    cars[i].assignPath(newPath);

    startNode.paths[i] = cars[i].getPath();

    double carCost = cars[i].getPathTime();
    startNode.costs[i] = carCost;
    startNode.cost += carCost;

    maxCarCost = std::max(maxCarCost, carCost);
  }

  openSet.push(startNode, startNode.cost);

  // For logs
  std::vector<double> meanCosts;
  std::vector<double> meanDepths;
```

```
  std::vector<double> meanTimes;
  auto start = std::chrono::system_clock::now();
  double clockLastRefresh = 0;
  int numNodeProcessed = 0;

  // While there are conflicts in the paths, resolve them
  while (!openSet.empty()) {
    auto duration =
        std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now() - start).count() /
        1000.0;

    numNodeProcessed++;
    CBSNode node = openSet.pop();

    // if (duration > CBS_MAX_SUB_TIME) {
    //   CBSNode resSub = createSubCBS(node, subNodeDepth);
    //   if (resSub.hasResolved) {
    //     return resSub;
    //   }
    // }

    std::vector<std::vector<sf::Vector2f» paths = node.paths;
    double cost = node.cost;
    int depth = node.depth;

    int car1, car2;
    sf::Vector2f p1, p2;

    double a1, a2;
    int time;
    bool conflict = hasConflict(paths, &car1, &car2, &p1, &p2, &a1, &a2, &time);

    if (!conflict) {
      for (int i = 0; i < numCars; i++) {
        cars[i].assignExistingPath(node.paths[i]);
      }
      node.hasResolved = true;
      return node;
    }

    meanCosts.push_back(cost);
    meanDepths.push_back(depth);
    meanTimes.push_back(time);

    if (clockLastRefresh + LOG_CBS_REFRESHRATE < duration) {
      double meanCost = std::accumulate(meanCosts.begin(), meanCosts.end(), 0.0) / meanCosts.size();
      double meanDepth = std::accumulate(meanDepths.begin(), meanDepths.end(), 0.0) / meanDepths.size();
      double meanTime = std::accumulate(meanTimes.begin(), meanTimes.end(), 0.0) / meanTimes.size();
      meanTime = meanTime * SIM_STEP_TIME;

      double remainingTime = (maxCarCost - meanTime) * (duration / meanTime);
      double processPerSecond = numNodeProcessed / (duration - clockLastRefresh);

      if (log) {
        spdlog::info("Node C: {:0>6.5} | D: {:0>6.5} | CT: {:0>6.5} | SD: {} | ET: {}s | "
                     "ETR: ~{}s | Processed nodes: ~{:0>4.5}/s",
                     meanCost, meanDepth, meanTime, subNodeDepth, (int)duration, (int)remainingTime, processPerSecond);
        std::cout « "\033[A\033[2K\r";
      }

      clockLastRefresh = duration;
      numNodeProcessed = 0;

      meanCosts.clear();
      meanDepths.clear();
      meanTimes.clear();
    }

    // Resolve conflict
    for (int iCar = 0; iCar < 2; iCar++) {
      int car = iCar == 0 ? car1 : car2;

      AStar::conflict newConflict;
      newConflict.point.position = iCar == 0 ? p2 : p1;
      newConflict.point.angle = iCar == 0 ? a2 : a1;
      newConflict.time = time;
      newConflict.car = iCar == 0 ? car1 : car2;

      // If already in constraints, skip
      if (node.constraints.hasConstraint(newConflict)) {
        continue;
      }

      ConstraintController newConstraints = node.constraints.copy();
```

```
      newConstraints.addConstraint(newConflict);

      TimedAStar aStar(cars[car].getStart(), cars[car].getEnd(), graph, &newConstraints, car);
      std::vector<AStar::node> newPath = aStar.findPath();

      if (newPath.empty()) {
        continue;
      }

      cars[car].assignPath(newPath);
      double carOldCost = node.costs[car];
      double carNewCost = cars[car].getPathTime();

      CBSNode newNode;
      newNode.paths = paths;
      newNode.paths[car] = cars[car].getPath();
      newNode.constraints = newConstraints;
      newNode.costs = node.costs;
      newNode.costs[car] = carNewCost;
      newNode.cost = cost - carOldCost + carNewCost;
      newNode.depth = depth + 1;
      newNode.hasResolved = false;

      newNode.cost = 0;
      for (int i = 0; i < numCars; i++) {
        newNode.cost += newNode.costs[i];
      }

      // hasConflict(newNode.paths, &car1, &car2, &p1, &p2, &a1, &a2, &time);
      // newNode.cost /= time;

      // newNode.cost = 1 / (double)time;

      openSet.push(newNode, newNode.cost);
    }
  }

  return startNode;
}

bool Manager::hasConflict(std::vector<std::vector<sf::Vector2f» paths, int *car1, int *car2,
sf::Vector2f *p1,
                            sf::Vector2f *p2, double *a1, double *a2, int *time) {
  int maxPathLength = 0;
  int numCars = (int)paths.size();
  for (int i = 0; i < numCars; i++) {
    maxPathLength = std::max(maxPathLength, (int)paths[i].size());
  }

  double width = graph.getWidth();
  double height = graph.getHeight();
  auto outOfBounds = [&](sf::Vector2f p) {
    return p.x + CAR_LENGTH < 0 || p.y + CAR_LENGTH < 0 || p.x - CAR_LENGTH > width || p.y -
CAR_LENGTH > height;
  };

  for (int t = 0; t < maxPathLength; t += CBS_PRECISION_FACTOR) {
    for (int i = 0; i < numCars; i++) {
      if (t >= (int)paths[i].size() - 1 || outOfBounds(paths[i][t]))
        continue;
      for (int j = i + 1; j < numCars; j++) {
        if (t >= (int)paths[j].size() - 1 || outOfBounds(paths[j][t]))
          continue;

        sf::Vector2f diff = paths[i][t] - paths[j][t];
        if (std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1) {
          *car1 = i;
          *car2 = j;
          *p1 = paths[i][t];
          *p2 = paths[j][t];
          *a1 = std::atan2(paths[i][t + 1].y - paths[i][t].y, paths[i][t + 1].x - paths[i][t].x);
          *a2 = std::atan2(paths[j][t + 1].y - paths[j][t].y, paths[j][t + 1].x - paths[j][t].x);
          *time = t;
          return true;
        }
      }
    }
  }

  return false;
}
```

## 4.49 renderer.cpp File Reference

Implementation of the Renderer class.

```
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <ompl/geometric/SimpleSetup.h>
#include <ompl/geometric/planners/rrt/RRT.h>
#include <spdlog/spdlog.h>
#include "aStar.h"
#include "config.h"
#include "renderer.h"
#include "utils.h"
```

### 4.49.1 Detailed Description

Implementation of the Renderer class.
This file contains the implementation of the Renderer class.
Definition in file renderer.cpp.

## 4.50 renderer.cpp

Go to the documentation of this file.

```

#include <algorithm>
#include <iostream>
#include <random>
#include <vector>

#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <ompl/geometric/SimpleSetup.h>
#include <ompl/geometric/planners/rrt/RRT.h>
#include <spdlog/spdlog.h>

#include "aStar.h"
#include "config.h"
#include "renderer.h"
#include "utils.h"

namespace ob = ompl::base;

void Renderer::startRender(const CityMap &cityMap, const CityGraph &cityGraph, Manager &manager) {
  window.create(sf::VideoMode(SCREEN_WIDTH, SCREEN_HEIGHT), "City Map");

  // Set the view to the center of the city map, allowing some basic camera movement
  // Arrow to move the camera, + and - to zoom in and out
  double height = cityMap.getHeight();
  double width = cityMap.getWidth();
  sf::View view(sf::FloatRect(0, 0, width, height));
  // Reset view function
  auto resetView = [&]() {
    double screenRatio = window.getSize().x / (double)window.getSize().y;
    double cityRatio = width / height;
    view.setCenter(width / 2, height / 2);
    if (screenRatio > cityRatio) {
      view.setSize(height * screenRatio, height);
    } else {
      view.setSize(width, width / screenRatio);
    }
    window.setView(view);
  };

  resetView();
  time = 0;

  sf::Clock clockCars;
  bool speedUp = false;
  bool pause = true;

  while (true) {
    sf::Event event;
```

```
    while (window.pollEvent(event)) {
      if (event.type == sf::Event::Closed) {
        window.close();
        return;
      }

      if (event.type == sf::Event::MouseButtonPressed) {
        if (event.mouseButton.button == sf::Mouse::Left) {
          sf::Vector2f mousePos = window.mapPixelToCoords(sf::Mouse::getPosition(window));
          manager.toggleCarDebug(mousePos);
        }
      }

      if (event.type == sf::Event::KeyPressed) {
        if (event.key.code == sf::Keyboard::Escape) {
          window.close();
          return;
        }
        if (event.key.code == sf::Keyboard::Up) {
          view.move(0, -height * MOVE_SPEED);
        }
        if (event.key.code == sf::Keyboard::Down) {
          view.move(0, height * MOVE_SPEED);
        }
        if (event.key.code == sf::Keyboard::Left) {
          view.move(-width * MOVE_SPEED, 0);
        }
        if (event.key.code == sf::Keyboard::Right) {
          view.move(width * MOVE_SPEED, 0);
        }
        if (event.key.code == sf::Keyboard::Equal) {
          view.zoom(1.0f - ZOOM_SPEED);
        }
        if (event.key.code == sf::Keyboard::Dash) {
          view.zoom(1.0f + ZOOM_SPEED);
        }
        if (event.key.code == sf::Keyboard::R) {
          resetView();
          spdlog::debug("View reset");
        }
        if (event.key.code == sf::Keyboard::D) {
          debug = !debug;
          spdlog::debug("Debug mode: {}", debug);
        }
        if (event.key.code == sf::Keyboard::S) {
          speedUp = !speedUp;
        }
        if (event.key.code == sf::Keyboard::P) {
          pause = !pause;
        }
      }

      // If resizing the window, reset the view
      if (event.type == sf::Event::Resized) {
        resetView();
      }
    }

    window.setView(view);
    window.clear(sf::Color(247, 246, 242));
    renderCityMap(cityMap);
    renderManager(manager);
    if (!pause) {
      if (clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME ||
          (speedUp && clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME / 5)) {
        time += SIM_STEP_TIME;
        manager.moveCars();
        clockCars.restart();
      }
    }
    if (debug) {
      renderCityGraph(cityGraph, view);
    }
    // Remove outside the border (draw blank)
    sf::RectangleShape rectangle(sf::Vector2f(width, height));
    rectangle.setFillColor(sf::Color(247, 246, 242));

    float w = width;
    float h = height;

    std::vector<sf::Vector2f> border = {{-w, -h}, {0, -h}, {w, -h}, {w, 0}, {w, h}, {0, h}, {-w, h}, {-w, 0}};
    for (auto b : border) {
      rectangle.setPosition(b);
      window.draw(rectangle);
    }

```

```
    renderTime();
    window.display();
  }
}

void Renderer::renderCityMap(const CityMap &cityMap) {
  // Draw buildings
  std::vector<sf::Color> randomBuildingColors = {
      sf::Color(233, 234, 232), sf::Color(238, 231, 210), sf::Color(230, 229, 226), sf::Color(236,
234, 230),
      sf::Color(230, 223, 216), sf::Color(230, 234, 236), sf::Color(210, 215, 222)};

  std::vector<sf::Color> greenAreaColor = {sf::Color(184, 230, 144), sf::Color(213, 240, 193)};

  sf::Color waterColor(139, 214, 245);

  auto greenAreas = cityMap.getGreenAreas();
  for (int i = 0; i < (int)greenAreas.size(); i++) {
    const auto &greenArea = greenAreas[i];
    auto points = greenArea.points;
    sf::ConvexShape convex;
    convex.setPointCount(points.size());
    for (size_t i = 0; i < points.size(); i++) {
      convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
    }
    convex.setFillColor(greenAreaColor[greenArea.type]);

    window.draw(convex);
  }

  auto waterAreas = cityMap.getWaterAreas();
  for (int i = 0; i < (int)waterAreas.size(); i++) {
    const auto &waterArea = waterAreas[i];
    auto points = waterArea.points;
    sf::ConvexShape convex;
    convex.setPointCount(points.size());
    for (size_t i = 0; i < points.size(); i++) {
      convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
    }
    convex.setFillColor(waterColor);

    window.draw(convex);
  }

  auto buildings = cityMap.getBuildings();
  for (int i = 0; i < (int)buildings.size(); i++) {
    const auto &building = buildings[i];
    auto points = building.points;
    sf::ConvexShape convex;
    convex.setPointCount(points.size());
    for (size_t i = 0; i < points.size(); i++) {
      convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
    }
    convex.setFillColor(randomBuildingColors[i % randomBuildingColors.size()]);

    window.draw(convex);
  }

  // Draw roads
  sf::Color roadColor(194, 201, 202);
  for (const auto &road : cityMap.getRoads()) {
    for (const auto &segment : road.segments) {
      sf::Vector2f basedP1(segment.p1.x, segment.p1.y);
      sf::Vector2f basedP2(segment.p2.x, segment.p2.y);

      double angle = segment.angle;

      sf::Vector2f widthVec(sin(angle), -cos(angle));
      widthVec *= (float)road.width / 2;

      sf::Vector2f p1 = basedP1 + widthVec;
      sf::Vector2f p2 = basedP1 - widthVec;
      sf::Vector2f p3 = basedP2 - widthVec;
      sf::Vector2f p4 = basedP2 + widthVec;

      sf::ConvexShape convex;
      convex.setPointCount(4);
      convex.setPoint(0, p1);
      convex.setPoint(1, p2);
      convex.setPoint(2, p3);
      convex.setPoint(3, p4);

      convex.setFillColor(roadColor);

      window.draw(convex);

      // Draw a circle at the start end end of the road (for filling the gap)
```

```
      double radius = road.width / 2;
      sf::CircleShape circle(radius);
      circle.setFillColor(roadColor);
      circle.setPosition(basedP1.x - radius, basedP1.y - radius);
      window.draw(circle);
      circle.setPosition(basedP2.x - radius, basedP2.y - radius);
      window.draw(circle);
    }
  }

  // Draw intersections
  if (debug) {
    for (const auto &intersection : cityMap.getIntersections()) {
      double radius = intersection.radius;
      sf::CircleShape circle(radius);
      circle.setFillColor(sf::Color(0, 255, 0, 50));
      circle.setPosition(intersection.center.x - radius, intersection.center.y - radius);
      window.draw(circle);
    }
  }
}

void Renderer::renderCityGraph(const CityGraph &cityGraph, const sf::View &view) {
  std::unordered_set<CityGraph::point> graphPoints = cityGraph.getGraphPoints();
  std::unordered_map<CityGraph::point, std::vector<CityGraph::neighbor» neighbors = cityGraph.getNeighbors();

  // Draw a line between each point and its neighbors
  for (const auto &point : graphPoints) {
    for (const auto &neighbor : neighbors[point]) {
      if (!neighbor.isRightWay)
        continue;

      double radius = turningRadius(neighbor.maxSpeed);
      auto space = ob::DubinsStateSpace(radius);
      ob::RealVectorBounds bounds(2);
      space.setBounds(bounds);

      // Draw only if one of the points is inside the view
      sf::Vector2f viewCenter = view.getCenter();
      sf::Vector2f viewSize = view.getSize();
      sf::Vector2f viewMin = viewCenter - viewSize / 2.0f;
      sf::Vector2f viewMax = viewCenter + viewSize / 2.0f;

      if (point.position.x < viewMin.x && neighbor.point.position.x < viewMin.x) {
        continue;
      }
      if (point.position.x > viewMax.x && neighbor.point.position.x > viewMax.x) {
        continue;
      }

      ob::State *start = space.allocState();
      ob::State *end = space.allocState();

      start->as<ob::DubinsStateSpace::StateType>()->setXY(point.position.x, point.position.y);
      start->as<ob::DubinsStateSpace::StateType>()->setYaw(point.angle);

      end->as<ob::DubinsStateSpace::StateType>()->setXY(neighbor.point.position.x, neighbor.point.position.y);
      end->as<ob::DubinsStateSpace::StateType>()->setYaw(neighbor.point.angle);

      // Draw the Dubins curve
      double step = CELL_SIZE / 2.0f;
      double distance = space.distance(start, end);
      int numSteps = distance / step;
      sf::Vector2f lastPosition;
      sf::Color randomColor = sf::Color(rand() % 255, rand() % 255, rand() % 255, 60);

      for (int k = 0; k < numSteps; k++) {
        if (k == 0) {
          lastPosition = {point.position.x, point.position.y};
          continue;
        }

        ob::State *state = space.allocState();
        space.interpolate(start, end, (double)k / (double)numSteps, state);

        double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
        double y = state->as<ob::DubinsStateSpace::StateType>()->getY();

        double distance = std::sqrt(std::pow(x - lastPosition.x, 2) + std::pow(y - lastPosition.y, 2));
        double angle = atan2(y - lastPosition.y, x - lastPosition.x) * 180 / M_PI;

        // Draw an arrow between the points
        drawArrow(window, lastPosition, angle, distance * 0.9, distance * 0.9 / 2, randomColor, false);
```

```
          lastPosition = {(float)x, (float)y};
        }

        continue;
        // Write the speed of the point
        sf::Text text;
        sf::Font font;
        font.loadFromFile("assets/fonts/arial.ttf");
        text.setFont(font);
        text.setString(std::to_string((int)(neighbor.maxSpeed * 3.6f)) + " km/h");
        text.setCharacterSize(24);
        text.setFillColor(sf::Color::Black);
        text.setOutlineColor(sf::Color::White);
        text.setOutlineThickness(1.0f);
        text.setPosition(point.position * 0.2f + neighbor.point.position * 0.8f);
        text.setScale(0.02f, 0.02f);
        text.setOrigin(text.getLocalBounds().width / 2.0f, text.getLocalBounds().height / 2.0f);
        window.draw(text);
      }

      // Draw a dot at each points
      double size = 0.3;
      sf::CircleShape circle(size);
      circle.setFillColor(sf::Color(255, 0, 0, 70));
      circle.setPosition(point.position.x - size, point.position.y - size);
      window.draw(circle);
    }
}

void Renderer::renderManager(Manager &manager) { manager.renderCars(window); }

void Renderer::renderTime() {
  // At the top right corner of the view (keep the same size even if the view is resized)
  sf::Text text;
  sf::Font font = loadFont();
  sf::Vector2f viewSize = window.getView().getSize();
  text.setFont(font);
  text.setCharacterSize(24);
  text.setFillColor(sf::Color::White);
  text.setPosition(window.getView().getCenter() + sf::Vector2f(viewSize.x / 2, -viewSize.y / 2) +
                   sf::Vector2f(-viewSize.x * 0.01f, viewSize.y * 0.01f));
  text.setString(std::to_string((int)time) + " s");
  text.setOutlineColor(sf::Color::Black);
  text.setOutlineThickness(1.0f);
  text.scale(viewSize.x * 0.001f, viewSize.x * 0.001f);
  text.setOrigin(text.getLocalBounds().width, 0);
  window.draw(text);
}
```

## 4.51 test.cpp File Reference

A file for testing the project.

```
#include "test.h"
#include "spdlog/spdlog.h"
#include "tinyxml2.h"
#include <SFML/Graphics.hpp>
```

### 4.51.1 Detailed Description

A file for testing the project.
Definition in file test.cpp.

## 4.52 test.cpp

Go to the documentation of this file.

```

#include "test.h"
#include "spdlog/spdlog.h"
#include "tinyxml2.h"
#include <SFML/Graphics.hpp>

void Test::runTests() {
  testSpdlog();
  testTinyXML2();
  testSFML();
}
```

```

void Test::testSpdlog() {
  try {
    spdlog::debug("Testing spdlog...");
    spdlog::debug("spdlog is working as expected.");
  } catch (const std::exception &e) {
    throw std::runtime_error("spdlog is not working as expected.");
  }
}

void Test::testTinyXML2() {
  try {
    spdlog::debug("Testing TinyXML2...");
    tinyxml2::XMLDocument xmlDoc;
    xmlDoc.Parse("<root></root>");
    if (xmlDoc.Error()) {
      spdlog::error("TinyXML2 is not working as expected.");
      throw std::runtime_error("TinyXML2 is not working as expected.");
    }
    spdlog::debug("TinyXML2 is working as expected.");
  } catch (const std::exception &e) {
    spdlog::error("TinyXML2 is not working as expected.");
    throw std::runtime_error("TinyXML2 is not working as expected.");
  }
}

void Test::testSFML() {
  try {
    spdlog::debug("Testing SFML...");
    sf::RenderWindow window(sf::VideoMode(100, 100), "Test");
    if (!window.isOpen()) {
      spdlog::error("SFML is not working as expected.");
      throw std::runtime_error("SFML is not working as expected.");
    }
    window.close();
    spdlog::debug("SFML is working as expected.");
  } catch (const std::exception &e) {
    spdlog::error("SFML is not working as expected.");
    throw std::runtime_error("SFML is not working as expected.");
  }
}
```

## 4.53 timedAStar.cpp File Reference

Timed A∗ algorithm implementation.

```
#include "aStar.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"
#include <spdlog/spdlog.h>
#include <unordered_set>
```

### 4.53.1 Detailed Description

Timed A∗ algorithm implementation.

This file contains the implementation of the Timed A∗ algorithm. It is used to find the shortest path between two points in a graph with time constraints.

Definition in file timedAStar.cpp.

## 4.54 timedAStar.cpp

Go to the documentation of this file.

```

#include "aStar.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"

#include <spdlog/spdlog.h>
#include <unordered_set>

TimedAStar::TimedAStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph,
                       ConstraintController *conflicts, int carIndex) {
  this->start.point = start;
  this->start.speed = 0;
```

```cpp
  this->end.point = end;
  this->end.speed = 0;
  this->graph = cityGraph;
  this->conflicts = conflicts;
  this->carIndex = carIndex;
}

void TimedAStar::process() {
  path.clear();

  std::unordered_map<AStar::node, AStar::node> cameFrom;
  std::unordered_map<AStar::node, double> gScore;
  std::unordered_map<AStar::node, double> fScore;

  auto heuristic = [&](const AStar::node &a) {
    sf::Vector2f diff = end.point.position - a.point.position;
    double distance = std::sqrt(diff.x * diff.x + diff.y * diff.y);
    return distance / CAR_MAX_SPEED_MS;

    CityGraph::neighbor end_;
    end_.point = end.point;
    end_.maxSpeed = CAR_MAX_SPEED_MS;
    end_.turningRadius = CAR_MIN_TURNING_RADIUS;
    Dubins dubins(a.point, end_, CAR_MAX_SPEED_MS, CAR_MAX_SPEED_MS);
    return dubins.time();
  };
  auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };

  std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSet(compare);
  std::unordered_set<AStar::node> isInOpenSet;

  openSet.push(start);
  gScore[start] = 0;
  fScore[start] = heuristic(start);

  auto neighbors = graph.getNeighbors();

  int nbIterations = 0;
  while (!openSet.empty() && nbIterations++ < 1e5) {
    AStar::node current = openSet.top();
    openSet.pop();
    isInOpenSet.erase(current);

    if (current.point == end.point) {
      AStar::node currentCopy = current;

      while (!(currentCopy == start)) {
        path.push_back(currentCopy);
        currentCopy = cameFrom[currentCopy];
      }
      path.push_back(currentCopy);
      std::reverse(path.begin(), path.end());
      processed = true;
      break;
    }

    for (const auto &neighborGraphPoint : neighbors[current.point]) {
      if (current.speed > neighborGraphPoint.maxSpeed)
        continue;

      if (!neighborGraphPoint.isRightWay && ROAD_ENABLE_RIGHT_HAND_TRAFFIC)
        continue;

      std::vector<double> newSpeeds;
      newSpeeds.push_back(current.speed);

      double distance = neighborGraphPoint.distance;
      double nSpeedAcc = std::sqrt(std::pow(current.speed, 2) + 2 * CAR_ACCELERATION * distance);
      double nSpeedDec = std::sqrt(std::pow(current.speed, 2) - 2 * CAR_DECELERATION * distance);

      auto push = [&](double nSpeed) {
        int numSpeedDiv = 5;
        for (int i = 1; i < numSpeedDiv + 1; i++) {
          double s = (current.speed + (nSpeed - current.speed) * i / numSpeedDiv);
          if (s < SPEED_RESOLUTION)
            continue;
          newSpeeds.push_back(s);
        }
      };

      if (nSpeedAcc > neighborGraphPoint.maxSpeed && current.speed < neighborGraphPoint.maxSpeed) {
        push(neighborGraphPoint.maxSpeed);
        // newSpeeds.push_back(neighborGraphPoint.maxSpeed);
        // newSpeeds.push_back((current.speed + neighborGraphPoint.maxSpeed) / 2);
      } else if (nSpeedAcc < neighborGraphPoint.maxSpeed) {
        push(nSpeedAcc);
        // newSpeeds.push_back(nSpeedAcc);
```

```
        // newSpeeds.push_back((current.speed + nSpeedAcc) / 2);
      }

      if (nSpeedDec == nSpeedDec && std::isfinite(nSpeedDec)) {
        if (nSpeedDec < 0 && current.speed > 0) {
          push(0);
          // newSpeeds.push_back(0);
          // newSpeeds.push_back((current.speed + 0) / 2);
        } else if (nSpeedDec >= 0) {
          push(nSpeedDec);
          // newSpeeds.push_back(nSpeedDec);
          // newSpeeds.push_back((current.speed + nSpeedDec) / 2);
        }
      }

      AStar::node neighbor;
      neighbor.point = neighborGraphPoint.point;
      neighbor.arcFrom = {current.point, neighborGraphPoint};
      if (distance == 0) {
        neighbor.speed = current.speed;
        if (gScore.find(neighbor) == gScore.end() || gScore[current] < gScore[neighbor]) {
          cameFrom[neighbor] = current;
          gScore[neighbor] = gScore[current];
          fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);

          if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
            openSet.push(neighbor);

            isInOpenSet.insert(neighbor);
          }
        }
        continue;
      }

      for (const auto &newSpeed : newSpeeds) {
        if (newSpeed > CAR_MAX_SPEED_MS || newSpeed > neighborGraphPoint.maxSpeed || newSpeed < 0)
          continue;

        if (newSpeed == current.speed && newSpeed == 0)
          continue;

        neighbor.speed = newSpeed;

        double duration = 2 * distance / (current.speed + newSpeed);
        double tentativeGScore = gScore[current] + duration;

        double t = gScore[current];

        if (conflicts != nullptr &&
            conflicts->checkConstraints(carIndex, current.speed, newSpeed, t, current.point,
neighborGraphPoint))
          continue;

        if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
          cameFrom[neighbor] = current;
          gScore[neighbor] = tentativeGScore;
          fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);

          if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
            openSet.push(neighbor);
            isInOpenSet.insert(neighbor);
          }
        }
      }
    }
  }
}
```

## 4.55 utils.cpp File Reference

Utility functions implementation.

```
#include <spdlog/spdlog.h>
#include "car.h"
#include "utils.h"
```

### Functions

- sf::Font loadFont ()

  *Load a font.*

- bool carsCollided (Car car1, Car car2, int time)
- bool carConflict (sf::Vector2f carPos, double carAngle, sf::Vector2f confPos, double confAngle)

    *Check if two cars have a conflict.*

### 4.55.1 Detailed Description

Utility functions implementation.
Definition in file utils.cpp.

### 4.55.2 Function Documentation

#### carConflict()

```
bool carConflict (
            sf::Vector2f carPos,
            double carAngle,
            sf::Vector2f confPos,
            double confAngle)
```

Check if two cars have a conflict.

**Parameters**

| | |
|---|---|
| *carPos* | The position of the car |
| *carAngle* | The angle of the car |
| *confPos* | The position of the conflicting car |
| *confAngle* | The angle of the conflicting car |

**Returns**

If the cars have a conflict

Definition at line 49 of file utils.cpp.

#### carsCollided()

```
bool carsCollided (
            Car car1,
            Car car2,
            int time)
```

@bref Check if two cars collided

**Parameters**

| | |
|---|---|
| *car1* | The first car |
| *car2* | The second car |

Definition at line 23 of file utils.cpp.

#### loadFont()

```
sf::Font loadFont ()
```

Load a font.

**Returns**

The font

Definition at line 13 of file utils.cpp.

## 4.56 utils.cpp

Go to the documentation of this file.

```cpp

#include <spdlog/spdlog.h>

#include "car.h"
#include "utils.h"

bool fontLoaded = false;
sf::Font font;

sf::Font loadFont() {
  if (!fontLoaded) {
    if (!font.loadFromFile("assets/fonts/arial.ttf")) {
      spdlog::error("Failed to load font");
    }
    fontLoaded = true;
  }
  return font;
}

bool carsCollided(Car car1, Car car2, int time) {
  std::vector<sf::Vector2f> path1 = car1.getPath();
  std::vector<sf::Vector2f> path2 = car2.getPath();
  sf::Vector2f diff = path1[time] - path2[time];
  return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;

  sf::Vector2f pos1 = path1[time];
  sf::Vector2f pos2 = path2[time];

  double angle1 = atan2(path1[time + 1].y - path1[time].y, path1[time + 1].x - path1[time].x);
  double angle2 = atan2(path2[time + 1].y - path2[time].y, path2[time + 1].x - path2[time].x);

  sf::Vector2f p11 = pos1 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f * sin(angle1));
  sf::Vector2f p12 = pos1 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f * sin(angle1));
  sf::Vector2f p21 = pos2 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f * sin(angle2));
  sf::Vector2f p22 = pos2 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f * sin(angle2));

  bool colides = false;
  colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
  colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
  colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
  colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;

  return colides;
}

bool carConflict(sf::Vector2f carPos, double carAngle, sf::Vector2f confPos, double confAngle) {
  sf::Vector2f diff = carPos - confPos;
  return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;

  sf::Vector2f p11 = carPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle), CAR_LENGTH / 2.0f * sin(carAngle));
  sf::Vector2f p12 = carPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle), CAR_LENGTH / 2.0f * sin(carAngle));
  sf::Vector2f p21 = confPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle), CAR_LENGTH / 2.0f * sin(confAngle));
  sf::Vector2f p22 = confPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle), CAR_LENGTH / 2.0f * sin(confAngle));

  bool colides = false;
  colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
  colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
  colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
  colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;

  return colides;
}
```

# Index