# Multi-Agents Systems

# Traffic Lights - Project 2

Bejan-Topse Lucian-Cosmin,
Toderita Mihai-Catalin
- DSWT II 1A -

November 2023

# Contents

# Chapter 1

# Paper Scope

## 1.1 General Description

Implement a multi-agent system which manages traffic on a street layout with multiple intersections. The streets and intersections are configured as in the following image 1.1:

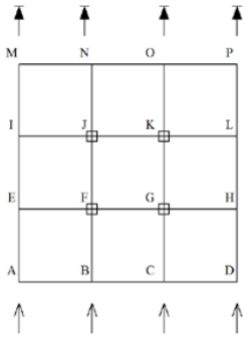



Figure 1.1: Map example

Each car is represented by an agent. Cars are incoming from entry points A, B, C, or D and each wants to get to one of the exits M, N, O or P. E … L are intersections; in each intersection there is a traffic light. Traffic lights should be represented by agents as well. Each traffic light will have its own switching time interval. The switching times should be initialized with equal values at first.

At each intersection, the car agents decide which way to go in a greedy manner, i.e they will choose a direction which:

- is either left, right or forward
- is selected such that the total distance from entry point to exit is equal to the Manhattan distance between the entry point and the exit (for example, if a car wants to get from B to O, it should stay within the rectangle B-C-O-N)
- has no obstacles / the least amount of obstacles

Possible obstacles are:

- **binary**: a red street light for that direction
- **continuous**: the amount of traffic in the desired direction (the number of cars in the corresponding road segment)

The system should be controllable by multiple user-defined parameters, which should be read from a simple config file. The control parameters are:

- The intelligence of traffic lights:
  - non-intelligent traffic light: the switching time is constant, no matter the traffic

  - intelligent: the switching time depends on the traffic
    * in this case, another parameter would be the level of intelligence:
      · level 1: the street light has knowledge of the amount of traffic in its adjacent street segments
      · level 2: the street light has knowledge of the amount of traffic as far as two street segments away from its position
      · level 3: the street light has full knowledge of the amount of traffic, on all street segments

- The rate with which cars are generated at each entry point A, B, C, D. Rate = cars/second, cars/minute etc.

- Whether cars will prioritize street lights or the amount of traffic, meaning:
  - whether a car at an intersection will choose a street segment with a green light or
  - whether a car will wait at a red light if there is lower traffic on the following street segment

A visual representation of the street segments, street lights and cars is required in order to better analyze the behavior of the system.

Note: you do not have to implement the intelligent behavior of the street lights, but you have to provide the functionality for the street light to receive the required amount of knowledge.

## 1.2 Project Usage

This project aims to demonstrate the efficiency of a range of traffic optimization techniques while providing a visual representation of the differences between those methods. Various parameters can be tweaked in the program to see certain scenarios. Traffic engineers and city planners can make great use of this type of project, testing how changes to the infrastructure affect volume and flow, and help making decisions which maximize efficiency.

# Chapter 2

# Implementation details

## 2.1 Individual components

### 2.1.1 Map

The map is composed of a 19x19 size grid (see figure 2.1), each square representing one unit of measurement for distance. There are 4 starting points (light blue) from which cars (blue) are spawned at a defined rate. Each car will have to cross through roads (gray) to reach one of the 4 final points (brown; which are on the same x axis). To implement the grid we created two lists, one for interest points on X axis (0, 6, 12, 18), and one for interest points on Y axis (5, 6, 12, 13), which are also used for creating the map drawing, and also by intersecting these two lists we get the intersections coordinates.

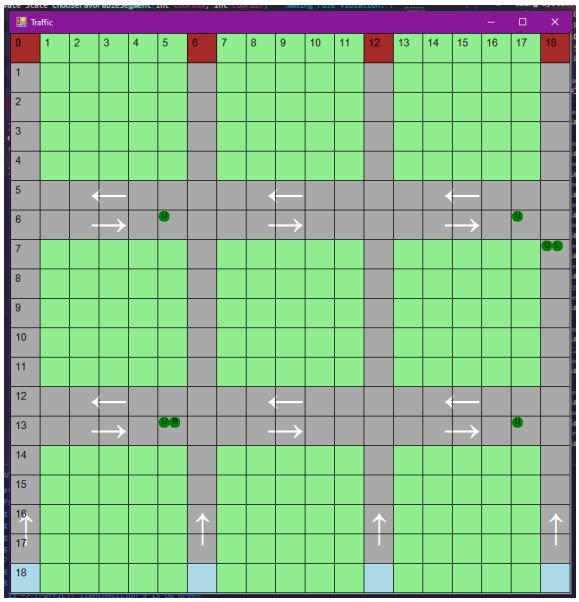



Figure 2.1: Map example

To implement the GUI we used a similar approach as in the Reactive Agents laboratory, using the Monitor Agent (Traffic Agent in our case) as the Form Owner and running the GUI in a separate thread, and continuously updating the Map every time the Act() method of the agent is called.

### 2.1.2 Road

Roads are defined as a sequence of gray squares, on which cars can travel in a certain way depending on the road position: up, left and right. The down direction is not taken into consideration since every

car has to travel from bottom to top. A consequence of this is that on the vertical axis there is only one lane, while horizontal roads have two lanes each.

### Road segments

Any sequence of road cells in which a car can only travel in one direction is considered a road segment (figure 2.2; segment color has no relevance). Depending on the placement, the only allowed direction for travelling is one of the three declared before. Road segments are isolated from each other, being connected only through intersections. Its size is determined by how many cars it can fit on the containing surface. These segments can be of various sizes, but in our representation, all road segments are of length 5.

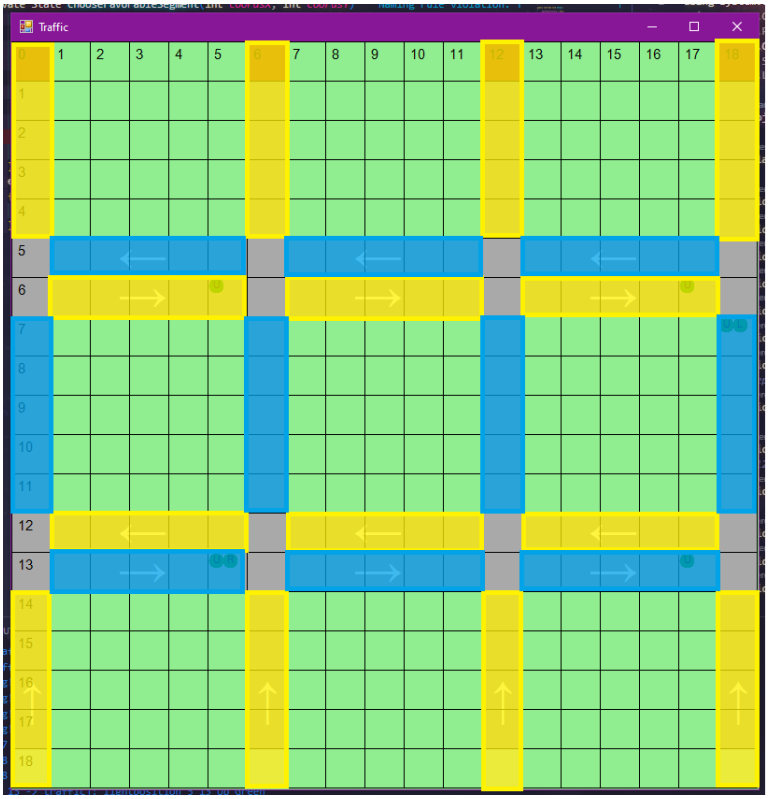



Figure 2.2: Road segments

### Intersections

Intersections (figure 2.3) are a series of road squares in which a car can adjust its direction in order to go on different road segments. In our representation, each intersection can be connected to 4 or 6 road segments, depending if it is on the edge of the map or not.

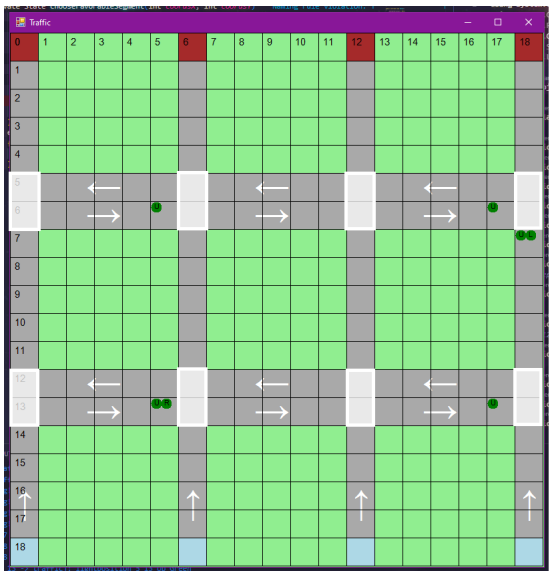



Figure 2.3: Intersections

### 2.1.3 Car

Cars (fig 2.4) are entities which travel on roads from a start position to a desired end position. For simplification, every car is considered to be identical in terms of size and speed. Acceleration is not taken into account, each car either not moving at all, or at the rate of one grid cell per turn.

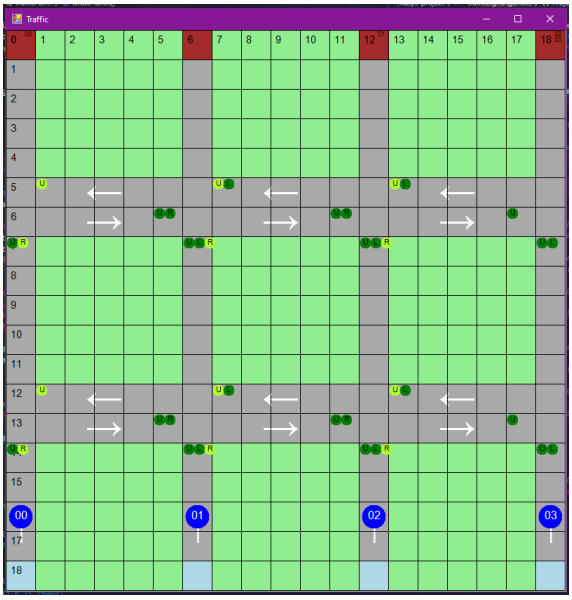



Figure 2.4: Cars

Two cars cannot overlap in traffic, if one car would want to move to a cell that is already occupied, then it has to wait until that cell is free. A car can be identified in the visual representation by its id, which starts from 0 and increases as more cars are generated. In our multi-agent system, first the car agent will start at its defined start position and tell the traffic so it can register its existence inside a global mapping variable.

```
public override void Setup()
{
    _optimalDirection = null;
    _direction = State.Up;
    Console.WriteLine($"[{Name}]: Starting! with target pos=[{this.targetPos.x} {this.targetPos
        .y}]");
    Send("traffic", Utils.Str("position", currentPos.ToString(), targetPos.ToString()));
}
```

Then after each tick, the car will change its position and notify the traffic agent about its intended movement:

```
public void HandleMove() {
    ...
    switch (_direction)
    {
        case State.Up:
            intendedPosition.y -= 1;
            break;
        case State.Left:
            intendedPosition.x -= 1;
            break;
        case State.Right:
            intendedPosition.x += 1;
            break;
        default:
            break;
    }
    //try sending car to intended position
    Send("traffic", Utils.Str("change", intendedPosition.ToString()));
}
```

### 2.1.4 Street light

Street lights (figure 2.5) are agents which can control if cars are allowed to enter an intersection or not. These are placed at the end of each road segment, and have a number of directional switches, their count varying from one to three. The up switch is mandatory for each street light, while left and right ones are optional.

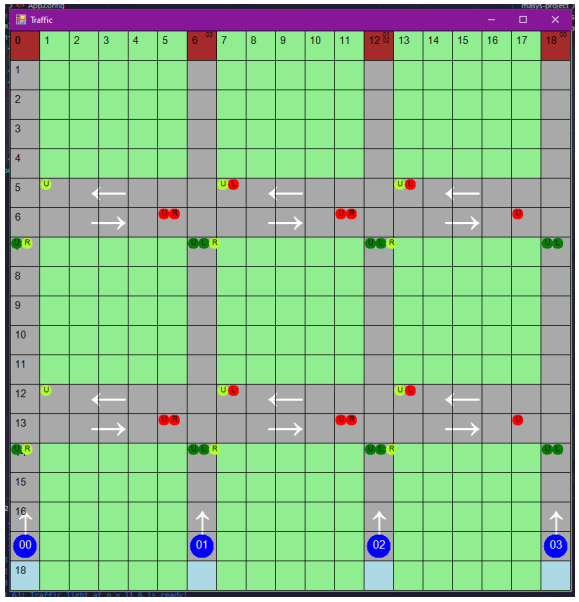



Figure 2.5: Street lights

The state of a street light switch can either be red (no cars passing), green (allow car passing) or intermitent green (allow cars going to the right direction). Depending on the street light intelligence setting, they have different behaviors which will be described more accurately in the optimization techniques section.

Since a Traffic Light Agents can have up to 3 traffic lights (Figure 2.6), we decided to create a separate class **TrafficLight** that would contain the traffic light details and also the light change logic. For clearer implementation, we chose the add a dictionary in our Traffic Light Agent that contains as keys the possible directions a car can go on, and as value a TrafficLight object.

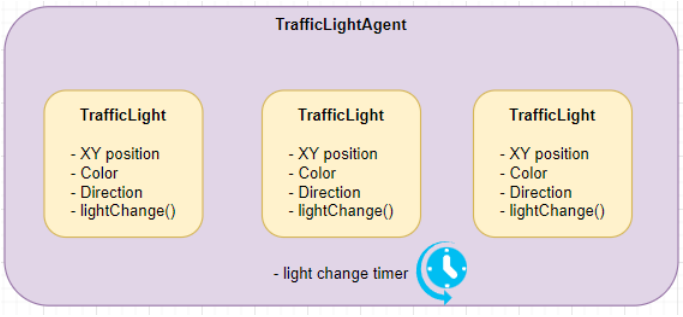



Figure 2.6: Street lights

### 2.1.5 Monitor agent (traffic)

The traffic agent is a central piece of the application, handling interactions between components. It is responsible for:

- Handling the map view, by translating the in-memory cars, traffic lights, roads lists into the GUI representation
- Spawning of cars and street lights
- Validating car moves
- Signaling cars when to wait
- Updating traffic lights switch statuses

## 2.2 Car Basics

### 2.2.1 Generation

Car generation is controlled by the traffic agent, which on an configurable interval will send itself a ”spawn” action. When this action is handled, on every 4 starting points it will be checked if another car exists already on the spawn row or one row above it; if this is the case, remove that starting point from the spawn possibilities in that turn.

```
1  private void HandleSpawn()
2  {
3      int x, y;
4      string[] t;
5      int[] possibleX = Utils.interestPointsX;
6
7      foreach (string k in Utils.CarPositions.Values)
8      {
9          t = k.Split();
10         x = Convert.ToInt32(t[0]);
11         y = Convert.ToInt32(t[1]);
12
13         if (((y == Utils.gridLength - 1) || (y == Utils.gridLength - 2))
14             && possibleX.Contains(x))
15         {
16             possibleX = possibleX.Where(val => val != x).ToArray();
17         }
18     }
19
20     for (int i = 0; i < Utils.carsToGenerate; i++)
21     {
22         if (possibleX.Count() == 0)
```

```
23          {
24              Console.WriteLine($"[{Name}] Can not spawn any more vehicles now. Will spawn again
                    at next cycle.");
25              break;
26          }
27          else
28          {
29              x = possibleX[Utils.RandNoGen.Next(possibleX.Count())];
30              possibleX = possibleX.Where(val => val != x).ToArray();
31
32              var carAgent = new CarAgent(new Position(x, Utils.gridLength - 1), new Position(
                    Utils.interestPointsX[Utils.RandNoGen.Next(4)], 0));
33
34              Utils.noAgents += 1;
35              this.Environment.Add(carAgent, string.Format("car{0:D2}", Utils.noAgents));
36          }
37      }
38  }
```

### 2.2.2 Greedy traversal

Cars initially travel where the roads allow them, and in intersections they are able to choose any road. This is bad, as we want the cars to reach their destination consistently in terms of time spent.

In order to achieve this, at the first opportunity in which the car can change its horizontal position we can set the car to choose the directon to go which will arrive at the vertical road which has the destination.

We do this by comparing the car and target x positions: if the car x is greater than the target x, it means that the car should lower its x value until it matches the target, equivalent with a change of direction to left; if the values are equal, the car is already on the right track, so it will continue to go upwards; otherwise, increase the x position until it reaches the desired value, or in this case: turning right.

```
1  //check car intention; basically if it is already in targetX/targetX-1/targetX+1 it will go UP,
       and if not it will consider also LEFT/RIGHT
2  int xAxisDifference = currentPos.x - targetPos.x;
3  _intendedDirection = (xAxisDifference < -1) ? State.Right : (xAxisDifference > 1) ? State.Left
       : State.Up;
```

The big flaw in this implementation is that all the decisions are made only on the first intersections, rendering the upper ones useless. Moreover, since every car makes left/right turns from the start, the lower horizontal sections become crowded very fast, while upper ones will always be empty. This behavior is aggravated even more when street lights are taken into account.

In order to avoid mentioned or similar situations, we can implement a series of enhancements which can be combined to yield better traffic flow.

### 2.2.3 Car moving

This is done based on the **intendedDirection** presented above. Basically, after the care chose a direction in which to travel, the intendedPosition X and Y values will change accordingly, and a message will be sent to the Traffic Agent to record its new position. If the received message is **wait** then the Car Agent will not update its position, and the intendedPosition will be reset:

```
1  public void HandleWait()
2  {
3      //reset intendedPosition if the car has to wait, so that the actual position is not updated
           in HandleMove
4      this.intendedPosition.x = -1;
5      this.intendedPosition.y = -1;
6      Send(this.Name, "move");
7  }
```

If the Traffic Agent considers that the Car can move on its intended position, it will respond with a **move** message, which will update the currentPosition with the previously computed intendedPosition, and will go on and decide the further step.

```
public void HandleMove()
{
    ...
    //update current position with intended position
    if (this.intendedPosition.x != -1)
    {
        this.currentPos.x = this.intendedPosition.x;
        this.currentPos.y = this.intendedPosition.y;
    }
    ...
}
```

## 2.3 Car Prioritization

This is one of the main project requirements, having a user defined parameter that decides the main car prioritization focus, which can be either Traffic Lights, or Traffic generated by Cars.

### 2.3.1 Adapting cars to streetlight situations

In this case the car will dont consider the traffic ahead, on any direction, but only the Traffic Light colors from its possible direction choices. Initially, between the start and final destination, a car could take multiple paths, but in this point it is not possible because the car will decide its path on the first intersection. To fix this, we let the car take alternate routes where the streetlight allows earlier, as long as it doesn't deviate further away from the finish. In our implementation, this means that if a car has the intended direction left or right we also allow it to go up if the streetlight shows green in that direction, but never in the opposite direction of the intended one.

```
private State chooseFavorableSegment() {
    var trafficLightState = Utils.TrafficLightPositions[this.currentPos.ToString()];

    //if the car has the oportunity to chose between two directions. Basically this happens
        only when the intendedDirection is "Left" or "Right"
    //as in that moment the car can go either "Up" or "Left"/"Right", and it will chose the
        first Green trafficlight
    if ($"{_intendedDirection}" != "Up")
    {
        string desiredDirection = trafficLightState.Where(x => (x.Key == "Up" || x.Key == $"{
            _intendedDirection}") && x.Value.Contains("Green"))
            .FirstOrDefault(x => x.Value.Contains("Green")).Key;
        switch (desiredDirection)
        {
            case "Up":
                return State.Up;
            case "Left":
                return State.Left;
            case "Right":
                return State.Right;
            default:
                return _intendedDirection;
        }
    }
    return _intendedDirection;
}
```

This will happen only on the first level of intersection, as in the last level the car just goes to the destination. On the first level, usually there are 2 choices a car can make, either Up or Left/Right (based on the final destination position), and by prioritizing the traffic lights, the car will choose the direction in which the color is Green, or will turn to Green first.

### 2.3.2 Avoiding traffic as a car

In the case the car will prioritize the road with less traffic the traffic light colors are not really important. A car can know before making its turn decision if the road segment ahead of its desired turn will be crowded or not, by analyzing the map and gathering information about the traffic in segments of interest. In our case, the car will decide first if its destination is on the left/right; if this is the case, it will compare the costs of going to the chosen in that direction and the up direction, eventually choosing the segment which has the lower cost.

```
1  private State chooseFavorableSegment() {
2      List<(State, double)> segmentCosts = new List<(State, double)>
3      {
4          getUpperSegmentCost(coordsX, coordsY)
5      };
6
7      if (_intendedDirection == State.Left) { segmentCosts.Add(getLeftSegmentCost(coordsX,
           coordsY)); }
8      if (_intendedDirection == State.Right) { segmentCosts.Add(getRightSegmentCost(coordsX,
           coordsY)); }
9
10     segmentCosts.Sort((a, b) => a.Item2.CompareTo(b.Item2));
11     return segmentCosts[0].Item1;
12 }
13 private (State, double) getUpperSegmentCost(int coordsX, int coordsY)
14 {
15     int noCars = 0;
16     int yDifference = 0, xDifference = 0;
17     if (Utils.interestPointsX.Contains(coordsX)) yDifference = 3;
18     else if (Utils.interestPointsX.Contains(coordsX + 1)) { yDifference = 2; xDifference = -1;
           }
19     else if (Utils.interestPointsX.Contains(coordsX - 1)) { yDifference = 1; xDifference = 1; }
20
21
22     for (int y = coordsY - yDifference; y > Utils.interestPointsY[1]; y -= 1)
23     {
24         if (Utils.CarPositions.Values.Contains(Utils.Str(coordsX - xDifference, y)))
25         {
26             noCars++;
27         }
28     }
29     return (State.Up, 25 / (5 - (double)noCars));
30 }
31 private (State, double) getLeftSegmentCost(int coordsX, int coordsY){ ... }
32 private (State, double) getRightSegmentCost(int coordsX, int coordsY){ ... }
```

If the streetlight color in that direction is Red, the car will receive a **wait** message from the Traffic Agent, which will make the car recompute the road segment costs, so it is possible that the care will change its direction if it will stay for long enough on a traffic light square.

The cost of a segment represents the result of the following formula:

$$c_k(t) = \frac{l_k}{v_k(t)} \qquad v_k(t) = \overline{v_k} \cdot \left[1 - \frac{q_k(t)}{y_k(t)}\right]$$

Where $l_k$ is the length of the road segment k, $\overline{v_k}$ is the maximum speed attainable on the segment k, $q_k$(t) is the number of vehicles on segment k at time t, $y_k$(t) is the maximum number of vehicles that can occupy the segment k. In our case, $l_k = 5$, $\overline{v_k} = 1$, and $y_k$(t)=5 on any road segment, simplifying the previous formula to:

$$c_k(t) = \frac{25}{5 - q_k(t)}.$$

## 2.4 General architecture. Component interactions.

In this chapter, we will focus on the inner workings of our multiagent system architecture, mainly how our components (Cars, Traffic Lights and Monitor Agents) communicate and work together. We tried to create a high level diagram (figure 2.7) that contains all the project components and how they communicate, but it is pretty hard to express all the edge cases, conditions, and actions. In our project, the core is represented by the Traffic Agent (which is our Monitor), and it has two types of communications, with either Cars Agents or Traffic Light Agents (Cars and Traffic Light agents don't communicate with each other). Based on this, in order to get a better understanding on how each component works, we will present every possible flow/interaction in a different subchapter and diagram, and explain in detail every message displayed in the high level diagram.

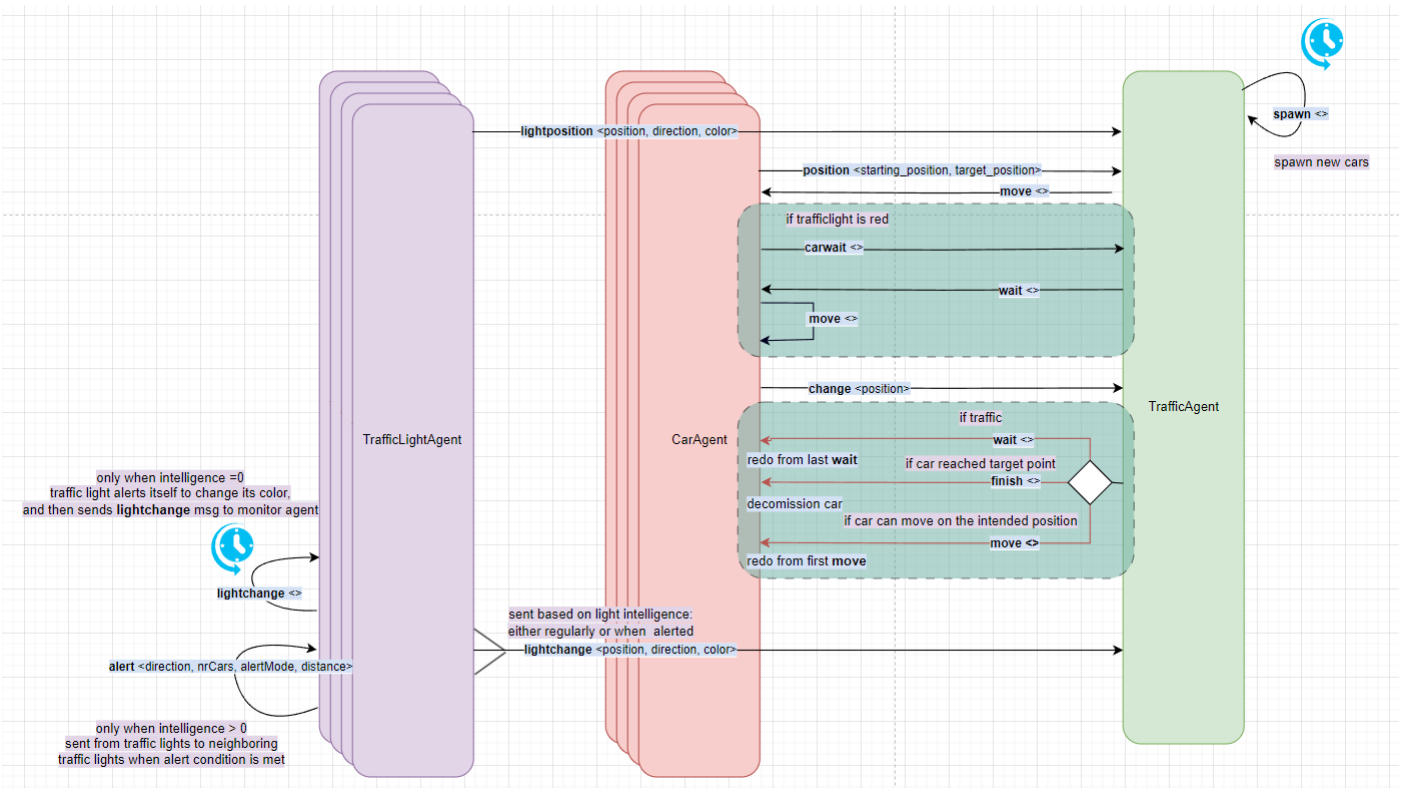



Figure 2.7: General Architecture

### 2.4.1 Car communication flow

The Car Agent communication implementation was the first task we tackled in this project. Initially we started by considering the traffic lights as green by default, and just focus on the movement and interaction between the cars and the road segments, but gradually we improved this logic as the project got more complex. In the end, we concluded on the architecture presented in Figure 2.7.

At the program startup, by default 1 to 4 cars will be spawned in different positions, and from this point the Monitor Agent is responsible with regularly spawning more Cars, based on the **CarGenerationRate** user defined parameter, as requested in the project requirement.

As a Car is spawned (either at the program startup, or later by the Traffic Agent), it will first choose a starting position and a target position and send this data in a **position** message to the Traffic Agent, which will register the Car existence in the Cars Positions Dictionary and respond by making the car **move**.

```
1 private void HandlePosition(string sender, string positions){
2     string[] t = positions.Split();
```

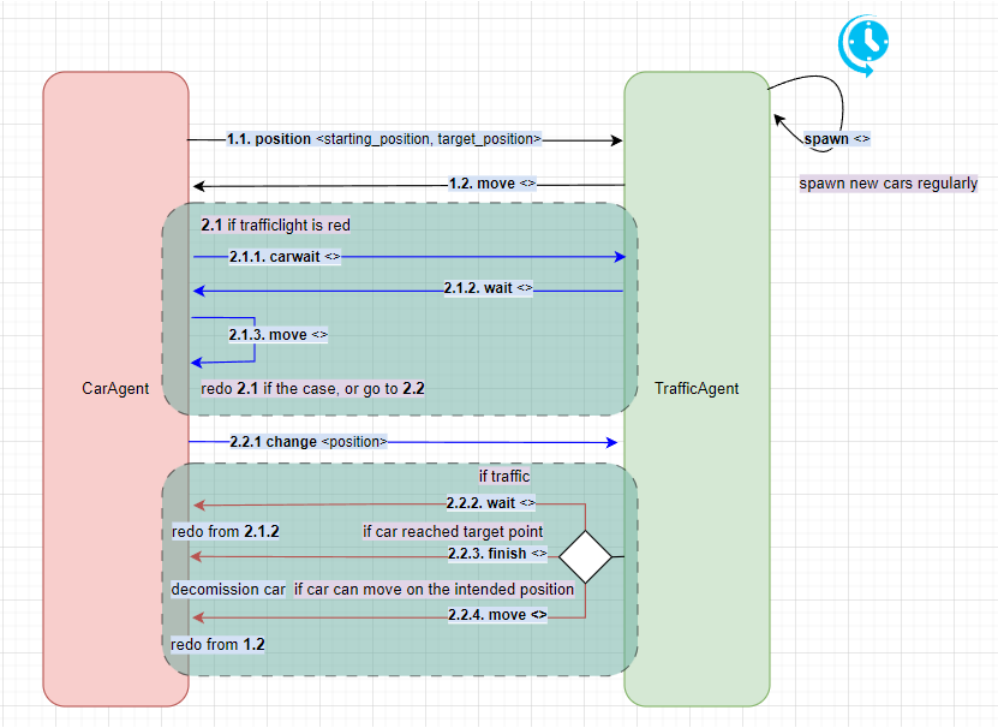



Figure 2.8: Car Agent Communication

```
3       Utils.CarPositions.Add(sender, $"{t[0]} {t[1]}");
4       Utils.CarDestinations.Add(sender, $"{t[2]} {t[3]}");
5       Send(sender, "move");
6   }
```

From this point, the car will try to move forward, by sending **change** messages to the Traffic Agent and update its position, if no obstacle is encountered. Inevitably, it will face a traffic light, that manages the traffic flow on up to three directions. Based on the car's intended direction (which is computed based on the car-prioritization user defined parameter) the traffic light color is either Green or IntermitentGreen, and if so the car can continue its journey, but if the color is Red (Figure 2.8 Step 2.1), it will be stuck for a short period of time. In the second scenario, when encountering a red colored traffic light, the car agent will send a **carwait** message to the Traffic Agent, which will respond with a **wait** message. This strange exchange is necessary, because we tried doing a Thread.Sleep() operation, but it was blocking the whole execution, and we also tried sending directly the **wait** message to the car itself, but this resulted in an infinite loop. The **wait** message will reset the intendedPosition value, and sent a **move** message to the car itself so that it can retry moving and recompute a new intendedPosition value. When the agent will try to move again, it will have to restart the whole process, check again which direction is optimal and resend a **change** message to the Traffic Agent.

On the Traffic Agent side, when it receives a **change** message from a Car agent, there are 3 possible outcomes:

- The car intended position is already occupied in the Car Positions dictionary, which means the car has to **wait** before trying to move again (redo from step 2.1.2).
- The car reached the destination it which case it is deleted from the Car Positions dictionary, and deleted from the Multi Agent environment.
- The car can actually update its position, in which case the Monitor agent will send a new **move** message to it (redo from step 1.2).

To resume, the architecture is quite repetitive, as the cars keep moving or just wait for a few turns, until they reach the destination and are decommissioned. The Traffic Agent's main jobs are to spawn new Car Agents and keep updating the Car Positions dictionary, so that all the cars don't risk in running into each other, or outside the bounds. In this equation, the Traffic Light Agents are not involved at

all, as there is no need for any extra message exchanges. The traffic light status are also kept in a global dictionary, which the cars can check before making any decision of moving/ waiting.

### 2.4.2 Traffic light flow - intelligence 0

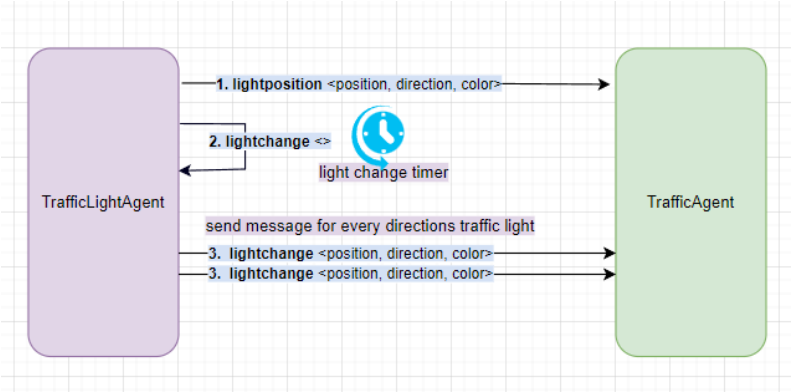



Figure 2.9: Traffic Light Agent Communication - intelligence 0

Right after finishing the basics of Car Agent movement and decision-making, we continued with making the traffic lights more complex, as just having them all on green by default was not an option. The first requirement, in regard to the traffic lights, was to have non-intelligent traffic lights, where the switching time is constant, no matter the traffic. We will call these **Intelligence Level 0 Traffic Lights**.

At the program startup, all the Traffic Light Agents are created, each with the correct position, number of TrafficLight objects that have the correct initial colors and directions. For the traffic light order and times we decided the following: internal intersections will have 3 traffic lights intervals, and the external/edge intersections will have only 2 traffic light intervals.

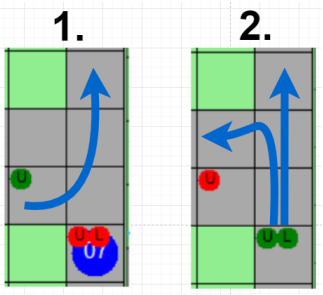



Figure 2.10: 2 Traffic light intervals

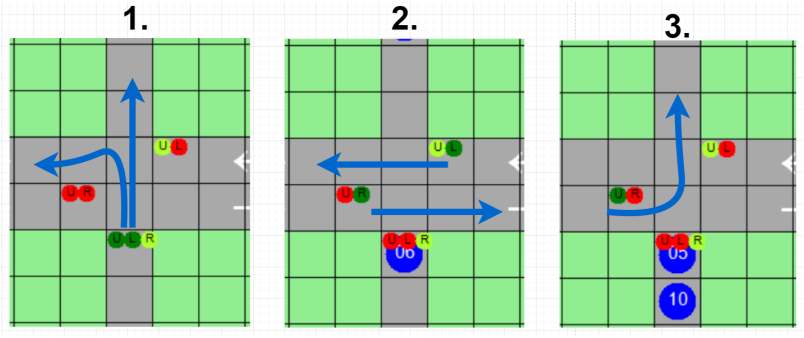



Figure 2.11: 3 Traffic light intervals

These intervals will be rotated once every 8 seconds, using a single timer per Traffic Light Agent, and using a configurationState variable to remember the last presented state. When the timer goes off, a **lightchange** message will be sent from the agent to itself, and it will check which interval it needs to adjust to, and will change the colors of the respective TrafficLight object, and also send a **lightchange** message to the Traffic Agent with the recent changes, so that it can update the global TrafficLightState dictionary.

### 2.4.3 Traffic light flow - intelligence > 0

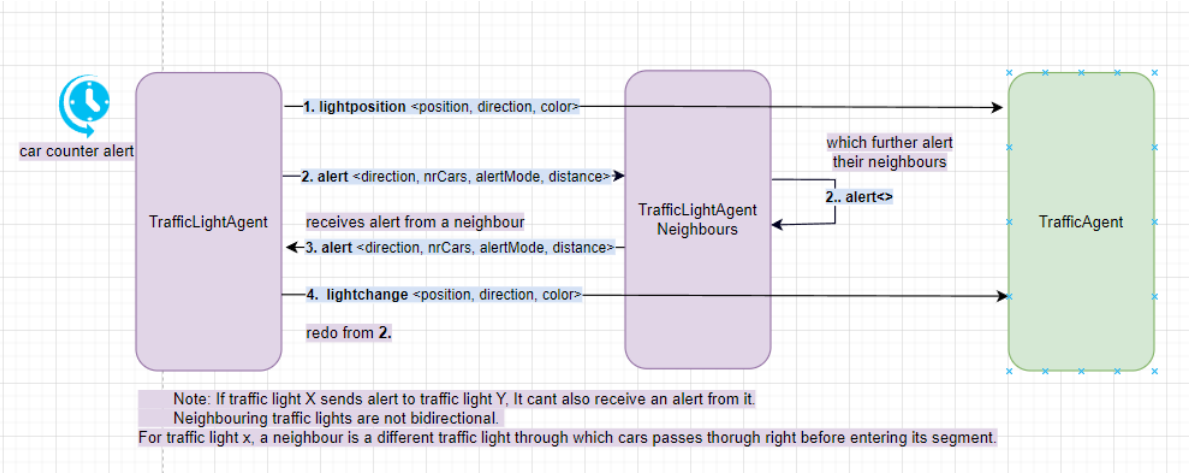



Figure 2.12: Traffic Light Agent Communication - intelligence 1/2/3

For the next level of traffic light intelligence, we needed to come up with a plan. The next 3 levels of intelligence are quite similar: for level 1 the traffic light knows the traffic on his adjacent streets, for level 2 it knows the traffic for up to 2 street segments, and for level 3 it knows the traffic on all street segments. The initial thought was that each traffic light could constantly check the whole map, but this would be pretty resource demanding. Another idea we had was that every traffic light could do a broadcast to all the other traffic lights to let them know of their local traffic flow, but this would mean that a very big number of messages would be sent. The thing that made us think outside the box was the last note in the project requirement, which stated the following: ”you do not have to implement the intelligent behavior of the street lights, but you have to provide the functionality for the street light to receive the required amount of knowledge”.

In the end, we came up with a different idea, based on the Fundamentals of Distributed Processing course we completed last year. We took inspiration from the Ring Graph Theory, mostly used by us in Leader Election Algorithms. The main concept we encountered last year was the idea of sending a pebble (a message containing different values) from one graph node to another, to pass any type of information around. We implemented a similar concept in this project as well.

We came up with the idea of Neighboring Traffic Lights, which are the Traffic Lights through which the cars go right before entering a Traffic Light road segment (which are the squares before the traffic light). Basically every traffic light would have 1 to 3 neighboring traffic lights, and if you think about it, this connection forms some kind of graph. The following is a pretty stuffed diagram (Figure 2.13), but it represents basically all the possible connections between the traffic lights. It almost represents a directed graph, and because of this we decided that it is a good and efficient idea to use some sort of pebble messages to make the traffic lights inform each other of any traffic issues.

Firstly, we created a Pebble class that contains the following attributes: Direction (the direction in which the pebble is sent), NrOfCars (the number of cars on the traffic lights road segment), AlertMode (which can be true or false), Distance (how many nodes the pebble passed through), Source (initial traffic light that sent the pebble). The idea would be the following: if on a traffic light, the number of

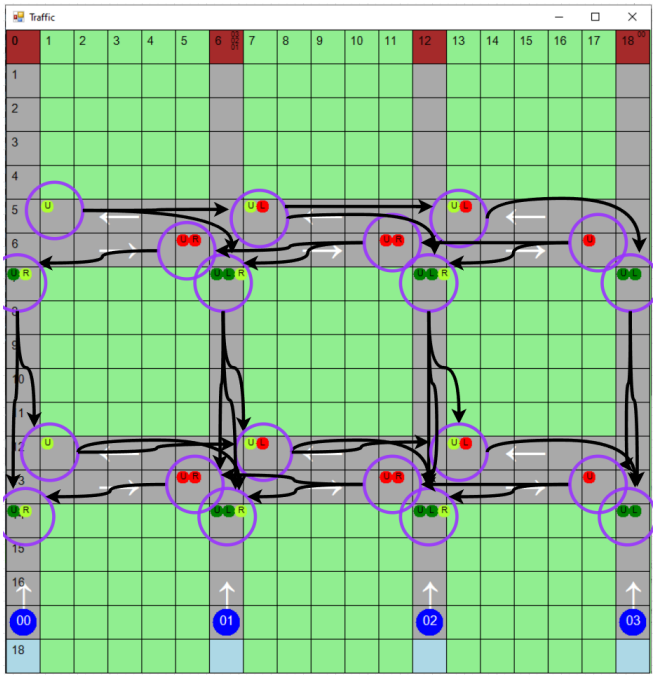



Figure 2.13: Direction Graph Idea

cars is bigger than a certain threshold, the traffic light would send a pebble message with AlertMode set to true, to all of its neighbours, which would let the neighboring traffic lights know that they should maybe turn Red. When the traffic ends, the initial traffic light would send a new pebble, but this time with AlertMode set to False, which would be basically a revoke of the last sent alert, and would travel across the same nodes.

For the different level of traffic light intelligence, on the architecture we developed, the only thing that is changed, is how far we let the Pebble travel, which is constantly checked through the Distance attribute.

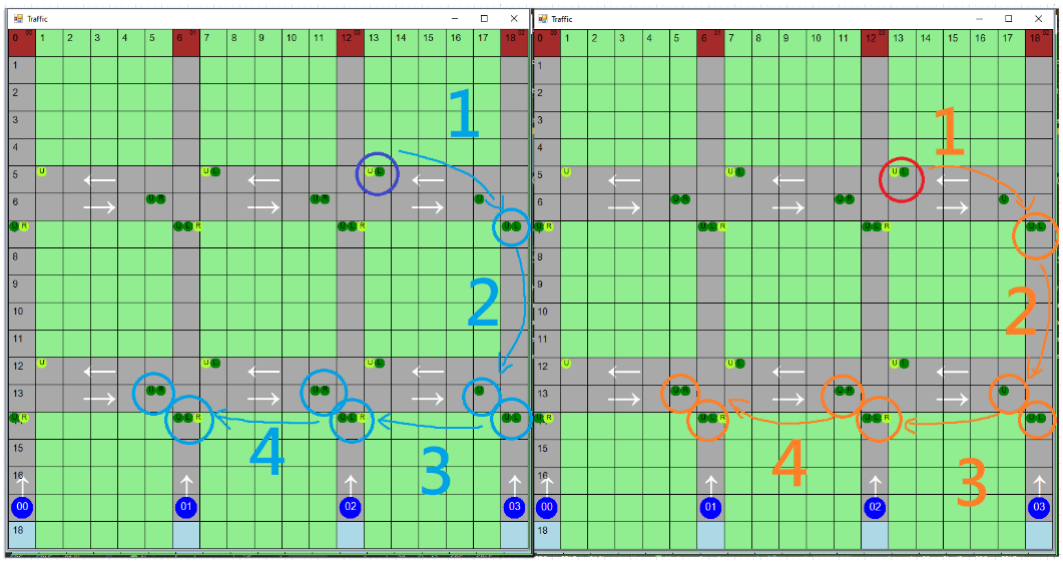



Figure 2.14: Pebble Alert

Let's dive through the implementation details. On this case, we don't need the traffic light interval switch timer, but we need a different timer that would constantly check the number of cars on the traffic light road segment, and when the threshold is exceeded it will send an **alert** message that contains a pebble with the necessary information.

```
private void alert_Elapsed(object sender, ElapsedEventArgs e)
{
    HandleCarCountOnSegment();
    if (carsOnMe >= Utils.AlertThreshold && localAlertMode == false) HandleAlertHelper(true);
    else if (carsOnMe < Utils.AlertThreshold && localAlertMode == true) HandleAlertHelper(false
        );
    return;
}

public void HandleAlertHelper(bool newAlertMode)
{
    localAlertMode = newAlertMode;
    Pebble alert = new Pebble(whereAmI, carsOnMe, localAlertMode, 1, this.Name);
    var neighboursToAlertAux = new List<string>(neighboursToAlert.Keys);

    foreach (var neigh in neighboursToAlertAux)
    {
        if (Utils.TrafficLightPositions.ContainsKey($"{neigh}") && neighboursToAlert[neigh] ==
            !localAlertMode)
        {
            Send($"light {neigh}", Utils.Str("alert", alert.ToString()));
            neighboursToAlert[neigh] = localAlertMode;
        }
    }

    //update ui so that it is visible which traffic light is in alert mode
    if (Utils.TrafficLightAlertMode.ContainsKey($"{pos.x} {pos.y}") && localAlertMode != Utils.
        TrafficLightAlertMode[$"{pos.x} {pos.y}"] && !firstRowTrafficLight)
    {
        Utils.TrafficLightAlertMode[$"{pos.x} {pos.y}"] = localAlertMode;
    }
}
```

When a traffic light agent receives an **alert** message it has several tasks to perform. Firstly, every Traffic Light Agent contains a dictionary which keeps up to date the alert status of every Traffic Light Neighbour. So the idea is that, if a neighbour is marked as in alert mode, there is no point in sending that way another pebble with AlertMode set to true. The only time we send alerts to neighbouring traffic lights is when the alert mode of that traffic light is different than the Pebble Alert Mode. For intelligence bigger then 1, before a pebble if passed further away it is also checked if the average number of cars from the initial traffic lights that sent the pebble, to the current threshold is bigger than a set up threshold. This was implemented in order to avoid sending out pebbles if the roads are empty.

```
public void HandleAlert(string sender, string alertPebble)
{
    Pebble receivedAlert = Utils.returnPebbleFromAlertString(alertPebble);
    Direction dir = Utils.ConvertToString(receivedAlert.Direction);

    //do stuff only if received alert is different than the current alert on the received
        direction
    if (directionAlertModes[receivedAlert.Direction] != receivedAlert.AlertMode){
        directionAlertModes[receivedAlert.Direction] = receivedAlert.AlertMode;
        Send("traffic", Utils.Str("lightchange", Utils.Str(pos.x, pos.y, trafficLights[dir].
            direction, trafficLights[dir].lightChange())));
    }

    HandleCarCountOnSegment();
    double localThreshold = (receivedAlert.NrOfCars + carsOnMe)/(receivedAlert.Distance + 1);
    //alert the neighbouring traffic lights only if the distance permits (the distance is
        unlimited only in intelligence 3 scenario)
    if (receivedAlert.Distance < Utils.TrafficLightAlertDistance){
        receivedAlert.Distance += 1;
        receivedAlert.Direction = whereAmI;
        var neighboursToAlertAux = new List<string>(neighboursToAlert.Keys);

        //send alert to neighbouring traffic lights with an updated pebble
        if (receivedAlert.AlertMode && localThreshold >= Utils.DistantAlertThreshold){
            receivedAlert.NrOfCars += carsOnMe;
            foreach (var neigh in neighboursToAlertAux){
                if (Utils.TrafficLightPositions.ContainsKey($"{neigh}") && neighboursToAlert[
                    neigh] == false){
                    Send($"light {neigh}", Utils.Str("alert", receivedAlert.ToString()));
                    neighboursToAlert[neigh] = true;
                }
```

```
28                  }
29              }
30              else if (!receivedAlert.AlertMode){
31                  foreach (var neigh in neighboursToAlertAux){
32                      if (Utils.TrafficLightPositions.ContainsKey($"{neigh}") && neighboursToAlert[
                            neigh] == true) {
33                          Send($"light {neigh}", Utils.Str("alert", receivedAlert.ToString()));
34                          neighboursToAlert[neigh] = false;
35                      }
36                  }
37              }
38          }
39      }
```

In our implementation, a traffic light does not have all the grid traffic knowledge, BUT, it has all the necessary data it needs, as it is only alerting the traffic lights that would send cars its way. So we consider this to be a pretty optimized solution for this use case.

# Chapter 3

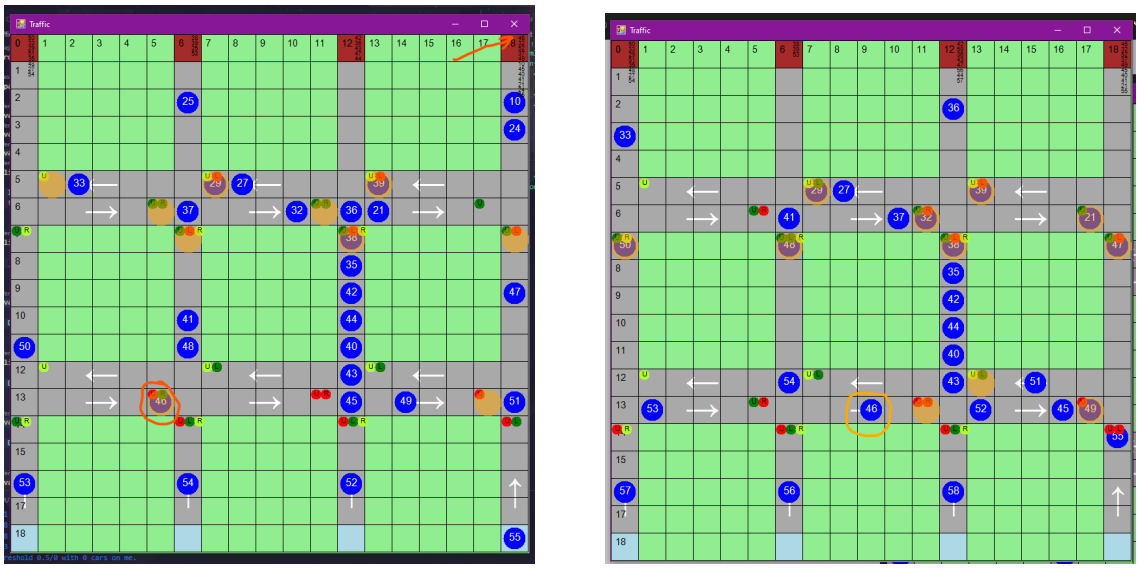
# Application in Execution

## 3.1 Car prioritization - Traffic lights

Config setup:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
    </startup>
  <appSettings>
    <add key="TrafficLightIntelligence" value="2" />
    <add key="CarGenerationRate" value="4" />
    <add key="CarPrioritization" value="trafficlights" />
    <add key="TurnDelay" value="1000" />
    <add key="AlertThreshold" value="1" />
    <add key="DistantAlertThreshold" value="0" />
  </appSettings>
</configuration>
```

In this scenario, car 46 has as target the right-most finish line. Upon reaching the street light, it observes that the up segment is blocked (the one which would be chosen by default if there is a tie). Seeing that the right road is free, the car takes that path instead of the default behavior one.



(a) Initial (b) Final

Figure 3.1: Traffic light prioritization cases

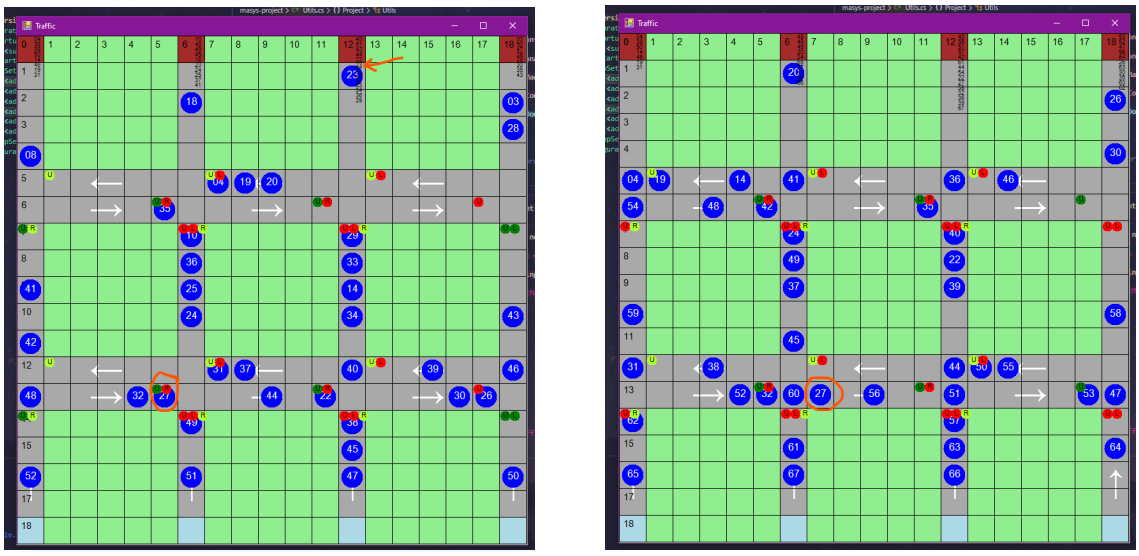
## 3.2 Car prioritization - Other cars

Config setup:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
    </startup>
  <appSettings>
    <add key="TrafficLightIntelligence" value="0" />
    <add key="CarGenerationRate" value="3" />
    <add key="CarPrioritization" value="cars" /> //important
    <add key="TurnDelay" value="1000" />
    <add key="AlertThreshold" value="2" />
    <add key="DistantAlertThreshold" value="0" />
  </appSettings>
</configuration>
```

The car 27 has to go from far left to the 3rd exit (left to right). When reaching the street light, it computes the costs for the segments of interest (up and right). According to the formula (2.3.2) the values in the first check being 25 for up and 8.(3). While both roads are blocked, the car recalculates these costs on every turn, such that when one of the segments is freed it has already decided where to go.

In this interval of time, even though the upper road has green light, the car waits longer because it has a higher cost than the right one. Once the switch is flipped green for the right section, the car will go on.



(a) Initial (b) Final

Figure 3.2: Other cars prioritization cases

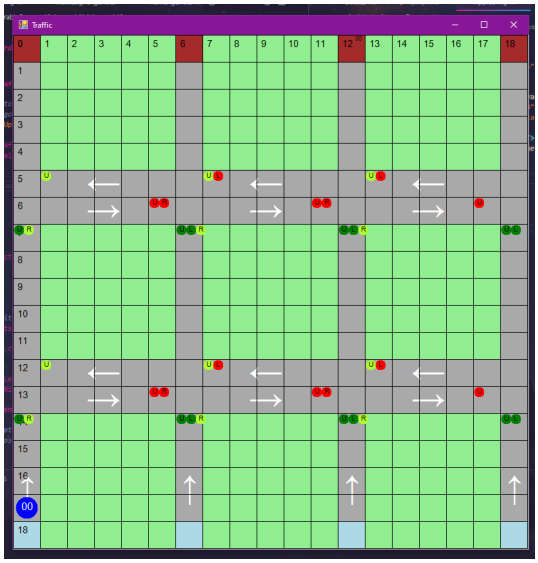
## 3.3 Traffic light intelligence 0

Config setup:

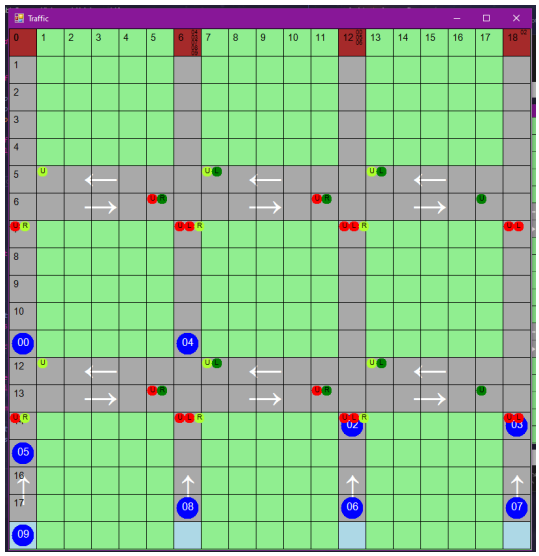
```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
    </startup>
  <appSettings>
    <add key="TrafficLightIntelligence" value="0" /> //important
    <add key="CarGenerationRate" value="3" />
    <add key="CarPrioritization" value="cars" />
```

```
    <add key="TurnDelay" value="1000" />
    <add key="AlertThreshold" value="2" />
    <add key="DistantAlertThreshold" value="0" />
  </appSettings>
</configuration>
```

For this case, we only need to look at the street light statuses and see that they cycle every 3 iterations (2 for lights in edge of map intersections)



(a) Iteration 1



(b) Iteration 2

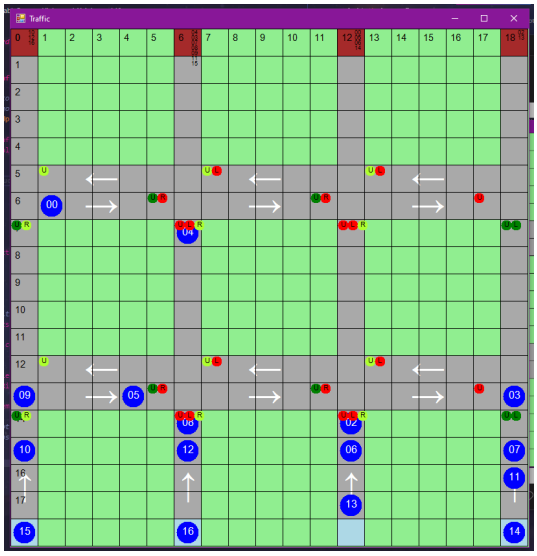


Figure 3.3: Street light cycle iteration 3

## 3.4 Traffic light intelligence - 1

Note - for intelligences 1 and 2 the configuration is the same as the one presented in the intelligence 0, but with the appropiate parameter set to the current case.

We can see that the segment containing cars 11 and 6 has triggered the alert: 2 or more cars are on itself. The figure 3.4 depicts the direct light neighbours that are affected by this alert.

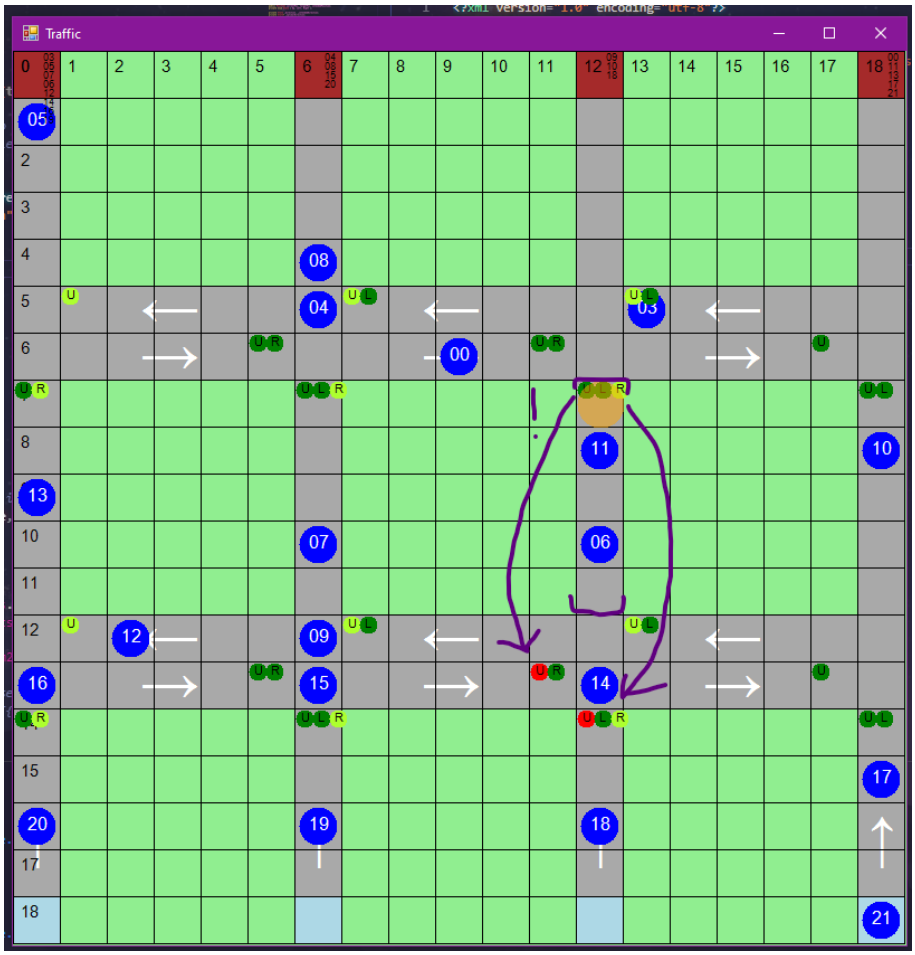



Figure 3.4: Street light intelligence 1

Note - the reason why the traffic light at coordinates (12, 13) is not showing the red light in the upper direction even when alerted is because intermitent green has always priority.

## 3.5 Traffic light intelligence - 2

There are multiple alerts triggered in the figure 3.5 but for our scenario we only need to focus on the one containing cars 2 and 12 in the left side of the map. With purple are depicted nearest alerted neighbours while the red arrows show the propagation of the alert message to the 2-segment distanced ones. In the blue box it is emphasized that the alert doesn't reach beyond the distance of 2 segments, leaving the street lights unaffected.

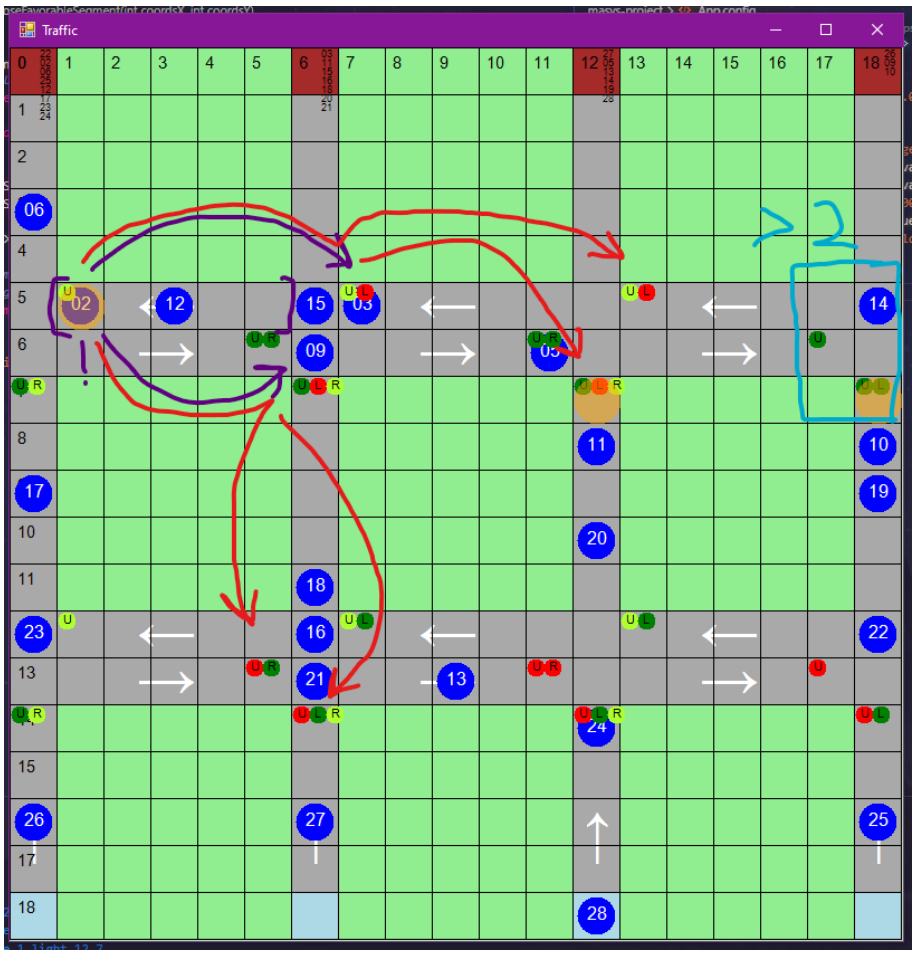



Figure 3.5: Street light intelligence 2

## 3.6 Traffic light intelligence - 3

Config setup:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
    </startup>
  <appSettings>
    <add key="TrafficLightIntelligence" value="3" /> //important
    <add key="CarGenerationRate" value="1000" />
    <add key="CarPrioritization" value="cars" />
    <add key="TurnDelay" value="1000" />
    <add key="AlertThreshold" value="1" />
    <add key="DistantAlertThreshold" value="0" />
  </appSettings>
</configuration>
```

Note - this setup was also made with only 1 spawned car to help in visualization.

On triggering the upper segments alerts, almost all the street lights are turned into red as a result of the alerting system.

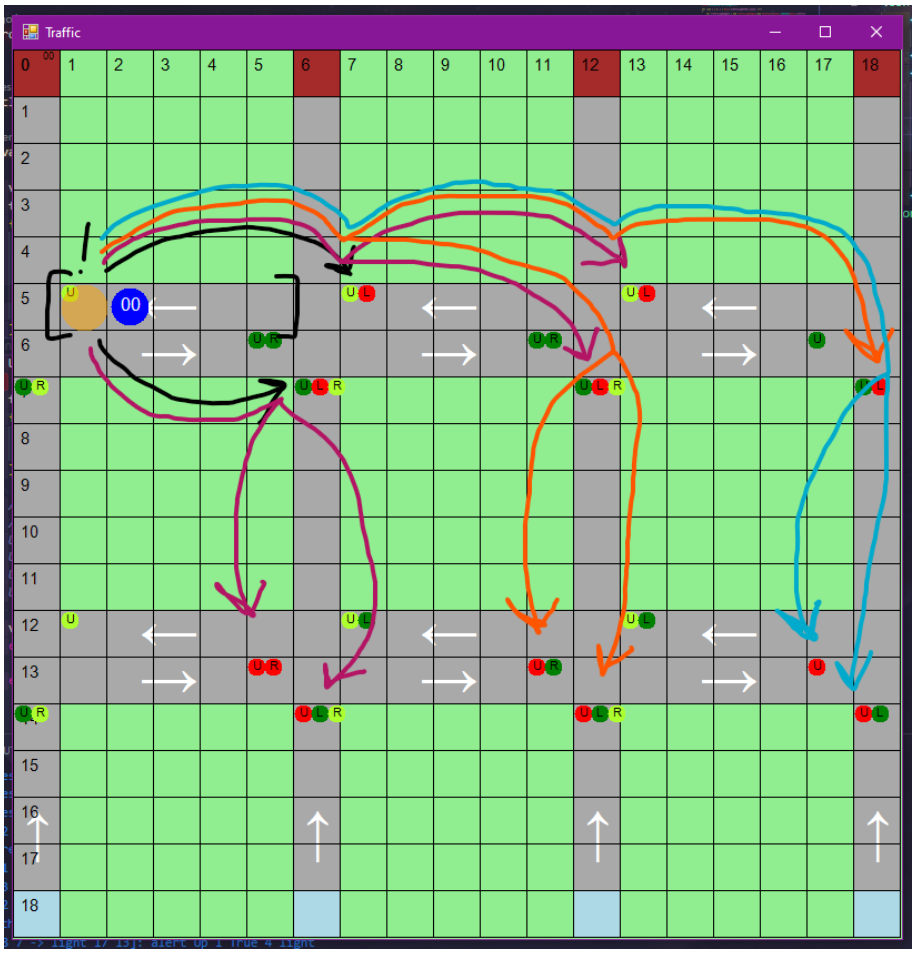



Figure 3.6: Street light intelligence 3

The figure 3.6 showcases the wide propagation of messages, grouped into distance traveled:

- Distance 1 - black
- Distance 2 - purple
- Distance 3 - orange
- Distance 4 - blue

# Chapter 4

# Team Members Responsabilities

During the project implementation, we worked mostly all the time together. We used GitHub as our version control system, where we have a similar number of commits. We constantly aligned with each other in person or online, we brainstormed different ideas together, and we actually did most of the complex tasks by peer programming. Although as a general outline we could try to differentiate the tasks as follows:

- GUI design and implementation: peer programmed led by Mihai
- Monitor Agent: Mihai and Lucian
- Car agent moving, decision making: Mihai and Lucian
- Car agent prioritization: Mihai
- Traffic Light - Intelligence 0 color intervals: Mihai
- Traffic Light - Intelligence > 0: Lucian
- Documentation: Mihai and Lucian

# Chapter 5

# Further improvements

As the complexity of this project is pretty high, we tried to make it more simpler, and easy to understand, and because of this there is a lot of room for further improvements:

- The grid map can be changed, so that every square could permit more cars, and maybe also have multiple lanes for every road segment. This way cars could have different speed values, and can try overtaking one another.
- the GUI can be improved so that the map is not redrawn on every frame. Some advanced techniques could be used here, but this was not the most important scope of the project.
- we could have different car sizes, or car types; like buses, bicycles, smaller cars, bigger cars and so on. We could also have buss stops, pedestrians, and many other obstacles, in order to make the map and the multi-agent system resemble the reality much more.
- the communication between the agents could also be improved. We could maybe change the architecture and have some communication rules between the cars and the streetlights as well, but this idea would need a lot of brainstorming.