Smart traffic management

-ZHAO Shengxun-
Université Côte d'Azur
L3-AI 2024-2025

Abstract

The smart traffic management system project is dedicated as a response to the demand given by university course SLEI606 – ECUE IA: Planification of L3 AI 2024-2025.

In this report, you will see a traffic simulation system presented by graph (supported by GraphStream[1]) who consists of BDI based intelligent agent (vehicle) and MDP based intersections (traffic lights).

Introduction

This project implements a dynamic urban traffic simulation system where autonomous vehicles navigate a city grid while adhering to traffic rules and responding to real-time environmental changes. The core of the vehicle class lies in its Belief-Desire-Intention (BDI) architecture, which enables vehicles (*agents*) to perceive their environment, set goals, and execute context-aware actions, and its Markov decision process (MDP) algorithm, which allows the traffic lights to adjust their signals based on the traffic situation. The simulation is built using GraphStream for visualization, with a focus on modularity and scalability.

Requirements

This project is coded with Java17.
To run this project, you just need to install maven dependencies then execute main.java.

System overview

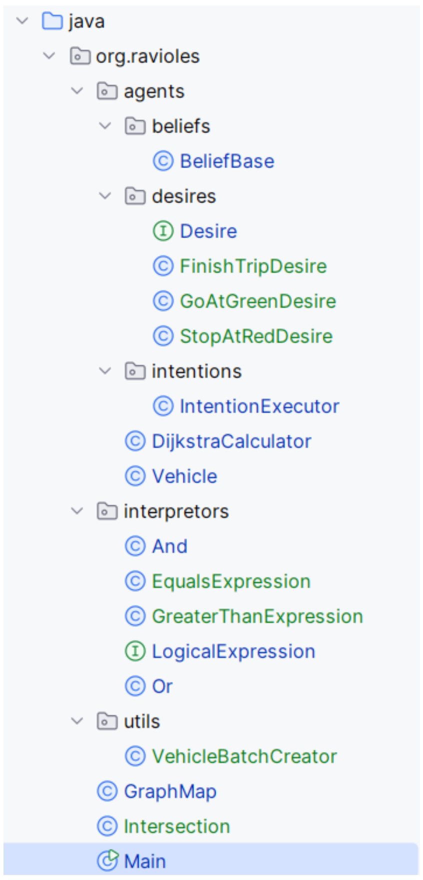
The system comprises three primary components:

- **Environment**: A grid-based road network with intersections managed by traffic lights.

- **Autonomous Vehicles**: Agents that navigate the network using path-finding algorithms and BDI-driven decision-making.

- **Intersection with traffic lights**: A MDP based finite state machine. Classic tri-color traffic lights manage a cross-shaped intersection.

Project structure



```
java
  org.ravioles
    agents
      beliefs
        BeliefBase
      desires
        Desire
        FinishTripDesire
        GoAtGreenDesire
        StopAtRedDesire
      intentions
        IntentionExecutor
      DijkstraCalculator
      Vehicle
    interpretors
      And
      EqualsExpression
      GreaterThanExpression
      LogicalExpression
      Or
    utils
      VehicleBatchCreator
    GraphMap
    Intersection
    Main
```

Technical details

Timer

Before we start to explain the big parts of the project, we shall know about the timer *tic* in the main class. This static volatile variable offers a possibility to synchronize all the system. The *tic* variable increments every 100ms while it is attached to a timer (java.swing).

The classes including GraphMap, Vehicle, Intersection have all implemented an interface called ActionListener and they can update their timer via an override function actionPerformed. Once the *tic* increases, they will *act*.

Environment

Relevant file : GraphMap.java
This file initializes a directed graph, a sprite manager “sman” who can manage the vehicles (*they are sprites*). The vehicles attached to a certain node or edge of the graph are saved in a list “vehicle”. The intersections are also initialized in this class as it will make a change on the graph. (*You will see it in the following demonstration*)
A node of the graph is named in a letter-number combination like “a2”, “b3”. We have currently 4 rows (a, b, c, d, e, f, g, h) and 8 columns (1, 2, 3, 4, 5, 6, 7, 8). An edge is named by its source node ID followed by its target node ID, for example: c2b2. By default, the *distance* attribute of an edge is 10 and its *congestion* level is 0.

The GraphMap class implements an interface called ActionListener who can detect the increment of the timer in the main class, to synchronize with other elements.

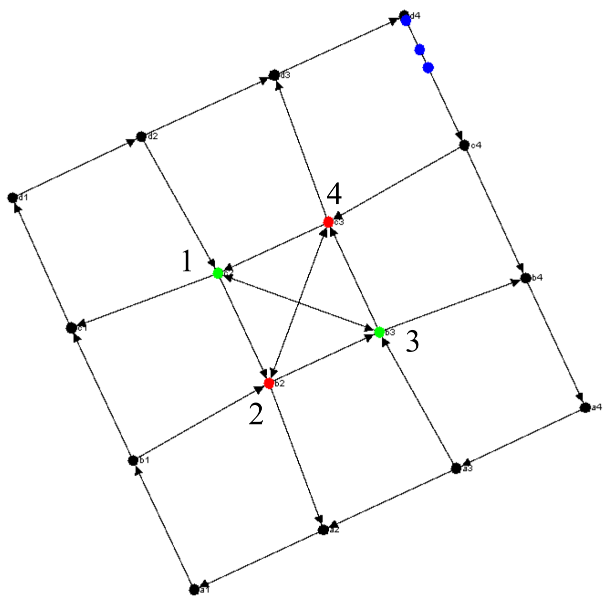
The synchronization in GraphMap helps to update the real-time congestion level of each edge. The congestion level of an edge increases by 1 once there is a sprite on it. However, the congestion level at the intersection is set always to be 0.

Then we can calculate the passibility of an edge by multiplying its distance by 1.5 raised to the power of the congestion level. This passibility value will be used to calculate the shortest path by Dijkstra method.

<u>Formla:</u>
$$Passiblity = distance * 1.5^{lv\_congestion}$$

Visual demonstration

This demonstrates a single intersection.



Autonomous Vehicles

Relevant files : Vehicle.java, BeliefBase.java, Desire.java, IntentionExecutor.java, DijkstraCalculator.java

In the vehicle class, we define a sprite to represent the vehicle. The sprite objects can be easily manipulated on the GraphStream graph thanks to a pre-built SpriteManager. Every sprite is registered to the SpriteManager.

The vehicle has its start node, end node and departure node, next node to realize basic functions on the graph. It will generate the shorted path at the beginning with the help of a class called DijkstraCalculator. This class will find the lowest weighted path by Dijkstra algorithm. It takes the passibility of the edges as the weight of it. The vehicle will get the next node to which it will move according to the path list generated.

The speed set to the vehicle will be divided by 100 and stored in *speed* attribute of the sprite. When operating on an edge, the vehicle calculates its position on edge by multiplying its speed by moving time. After it arrives (positionOnRoad = 1.0), the vehicle will update the departure node and fetch the next node. Then it will enter the edge whose source node is the vehicle's departure node and target node is the next node.

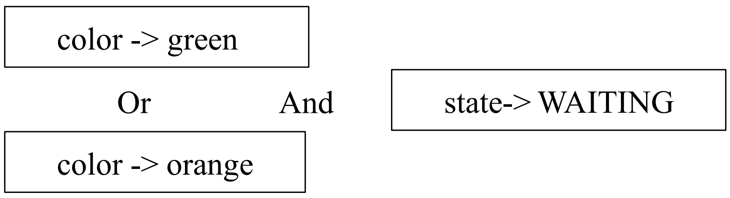
The vehicle has three states: DRIVING, WAITING and PARKED. Meanwhile, it has an adjustSpeed function who operates based on the current state of the vehicle.
DRIVING -> set the *speed* to the initial given speed.
WAITING -> set the *speed* to 0.
PARKED -> remove the vehicle from the graph and call the finishTrip function

This Vehicle class uses an architecture called BDI. A vehicle is initialized with BeliefBase, Desire and IntentionExcutor.

BeliefBase: the belief base of a vehicle allows it to perceive the environment (graph). It is initialized with basic beliefs such as "whether the vehicle has *arrived*", "the start point *startNode* of the vehicle" and "its destination *endNode*", and it will be updated every *tic* with the *state* of the vehicle and its *position* on the edge.
The vehicle can observe the traffic light with the belief base at the intersection associated with the current edge and updates the vehicle's desires accordingly. However, if the vehicle is at the intersection, this function will be disactivated because the traffic light manages only the traffic outside the intersection.

Desire: the Desire class is an interface who has three extensions: FinishTripDesire, StopAtRedDesire and GoAtGreenDesire. Every time the beliefs of the vehicle are updated, the desire list of the vehicle will be looped to determine which desire should be activated or is satisfied. Once a desire should be activated and it's not satisfied, it will generate a plan to trigger the IntentionExecutor.

Logical expressions are applied to desires. An interface called LogicalExpression with 4 extensions: And, Or, EqualsExpression, GreaterThanExpression, can be used to evaluate whether an expression is valid. For example, in the GoAtGreenDesire, we have an expression And aggregated with Or to determine the following logic:



If the expression above is true, the GoAtGreenDesire will generate a plan called "go"

Here is the plan examples generated by the desires:

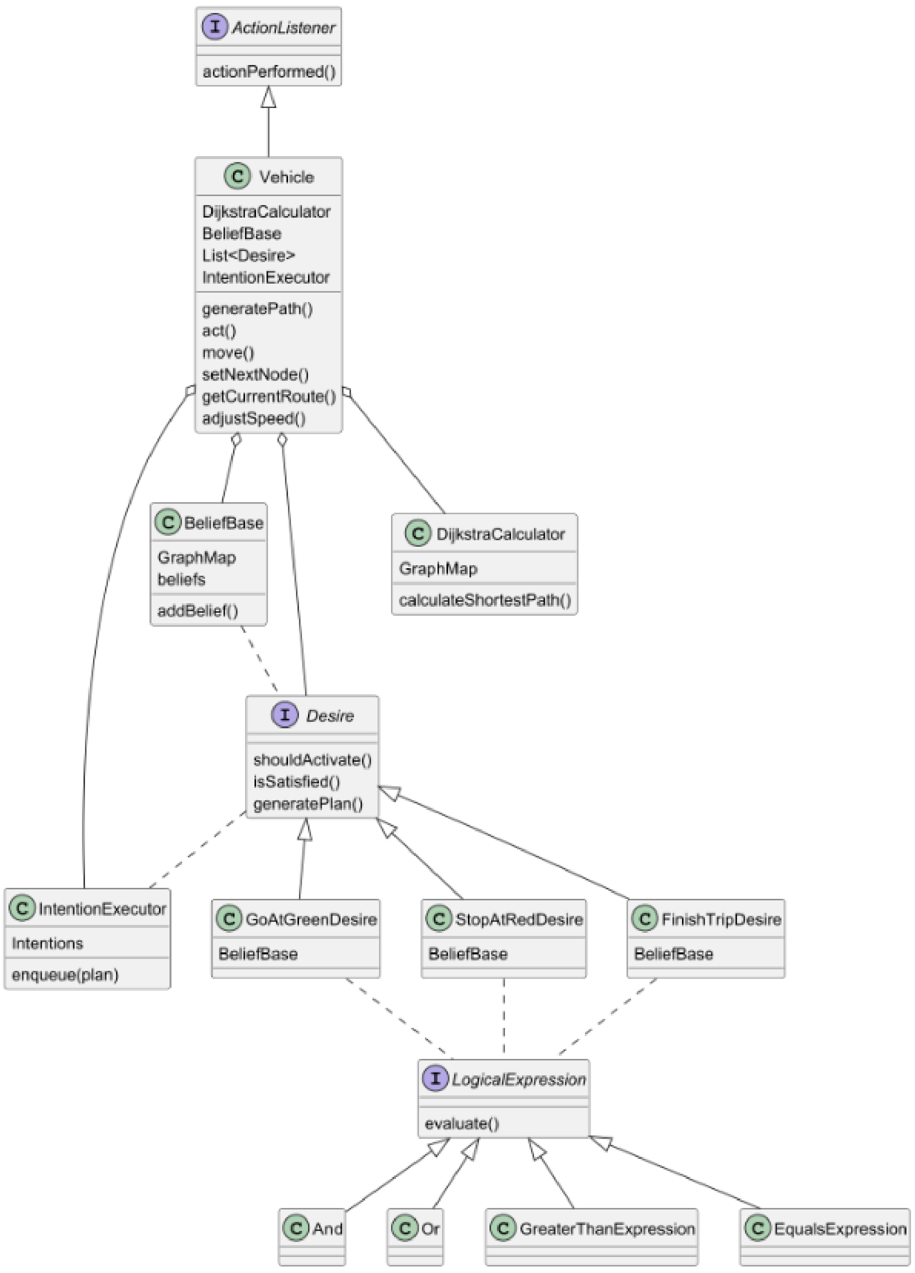
FinishiTripDesire -> “fin”
StopAtRedDesire -> “stop”
GoAtGreenDesire -> “go”

IntentionExecutor: this class explains the intentions of a vehicle by executing the plan generated by the desires.
“go” -> set current state of the vehicle to DRIVING
“stop” -> set current state of the vehicle to WAITING
“fin” -> set current state of the vehicle to PARKED
And it will trigger the adjustSpeed function of the vehicle accordingly.

The main action sequence for the vehicle is the *act()* function, which involves managing its state, updating its position and transitioning between nodes and edges in the graph.

The Vehicle class also provides its waiting time, which is the difference between the time on the road and the moving time. It is useful when calculating the punishment points for traffic lights.

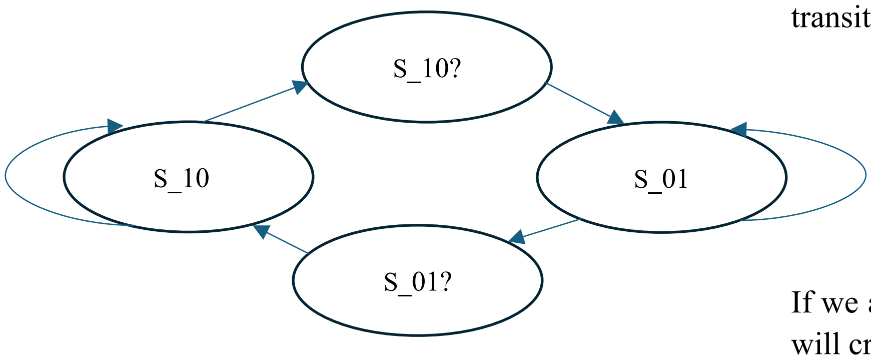
UML representation:



Intersection with traffic lights

Relevant file: Intersection.java

The intersection is a system consisting of 4 nodes as traffic lights. The traffic light is a finite state machine with the following transition:



It has 4 states:

s_10 : 1 -> green, 2 -> red, 3 -> green, 4 -> red
s_10? : 1 -> orange, 2->red, 3->orange, 4 -> red
s_01 : 1 -> red, 2 -> green, 3 -> red, 4 -> green
s_01?:1 -> red, 2-> orange, 3 -> red, 4 -> orange

(For the number of traffic lights please refer to the *visual demonstration* in the *environment* part)

4 actions:

a1 -> change to s_10
a2 -> change to s_10?
a3 -> change to s_01
a4 -> change to s_01?

An allowed action table is made to determine the transition.

If we attach an intersection object to the graph, it will create left-turns by connecting node 1 with 3, node 2 with 4. These new edges are bidirectional. It will also set the distances of all the edges in the intersection to 1.0. Please note that the congestion level at the intersection is always 0.

A Q table is initialized with the intersection. The Q table is a hashmap containing state as the key and an action map as the value. The action map contains actins as the key and its possibility to take that action as the value (the Q value).

The Q value is updated by the following formula:

*Q_new = (1-alpha)*Q_old + alpha * (reward + gamma*max(Q_next))*

(alpha = 0.5, gamma = 0.9)

The alpha value is the learning rate varied from (0,1]. When it's larger, it will emphasize the new feedback and converge faster.
The gamma value is the discount rate varied from [0,1). When it's closer to 1, it will emphasize the long-term profit.

The reward system of the intersection will give 10pts when a vehicle successfully passes the green traffic light and 1pt when it passes with the orange light. The reward system takes punishment points for keeping the vehicle waiting. The total reward will be reduced by the value of the waiting time at every *tic*.

The Intersection class implements the interface ActionListener just as the GraphMap and Vehicle class. It will also *act* to each *tic*. A countdown is created to set the interval between decisions. At every *tic*, the countdown decreases by 1 and when it comes to 0, the system will take actions according to the Q values in the Q table. Particularly, when the current state of the intersection is s_10? or s_01? ,which means some of the lights' color is orange, the countdown will be set to a smaller value to simulate the transition of orange lights.

Other utils

Relevant file: VehicleBatchCreator.java
The class VehicleBatchCreator is made to automatically create batches of vehicles on the graph every second. The vehicles created have a minimum speed of 10.0 and maximum speed 80.0. It will generate randomly the start node and the end node and avoid putting them at the intersections. By giving the number we want, it can create the same number of vehicles.

Conclusion

This project delivered a scalable, modular traffic simulation platform that integrates BDI-driven autonomous vehicles with MDP-based adaptive signal control, leveraging GraphStream for real-time visualization and fulfilling stakeholder requirements for a grid-based road network, context-aware agents, and intersections that dynamically adjust phases based on live congestion data
. Vehicles optimize route planning and manage state transitions (DRIVING–WAITING–PARKED) via a Dijkstra-enhanced BDI architecture and IntentionExecutor, while intersections employ Q-learning within an MDP framework to select signal actions that minimize wait-time penalties and maximize throughput
. To transition from prototype to operational deployment and drive strategic ROI, the platform should evolve through incorporation of ensemble reinforcement-learning strategies for global signal optimization, expansion to heterogeneous vehicle fleets across complex network topologies, and integration with live city traffic feeds—thereby establishing a practical foundation for future smart-city traffic management solutions.

Annexe

[1]GraphStream : https://graphstream-project.org/

Example of test: