Multi-Agent Traffic Simulation with BDI Agents and Logical Reasoning

Marius Casamian
CM22205773

April 28, 2025

Abstract

This project presents a dynamic urban traffic simulation where autonomous vehicles are modeled as BDI (Belief-Desire-Intention) agents equipped with logical reasoning capabilities. Agents perceive their environment — including traffic lights, obstacles, and road conditions — and make informed decisions through logical evaluation, relying on an *Interpreter Pattern* to process complex formulas. Traffic lights are independently managed as adaptive agents using Markov Decision Processes (MDP), capable of learning and optimizing their behavior over time through Q-Learning or Value Iteration. The system features a modular Java architecture and an intuitive JavaFX graphical interface, enabling real-time visualization, interaction, and detailed analysis of traffic dynamics and agent behavior. This document presents the overall design, the implementation choices, and the experimental observations that highlight the emergence of intelligent and coordinated behaviors in a dynamic environment.

Contents

1 Introduction **4**
1.1 Context and Challenges 4
1.2 Main Objectives 4
1.3 Report Structure 5

2 Conceptual MAS Model **5**
2.1 BDI Agent Architecture 5
2.1.1 Belief Management in Dynamic Environments 5

2.1.2 Desires 6
2.1.3 Intentions 6
2.2 BDI Cycle Management 7
2.3 Environment Structure 8
2.3.1 Network Topology 8
2.3.2 Traffic Regulation 8

3 Detailed Design **8**
3.1 Logical Reasoning System 8
3.1.1 Formula Types 9

4 Java Implementation **9**
4.1 Directory Tree 9
4.2 Key Packages 10

5 Experimental Results **10**
5.1 Tested Scenarios 10
5.2 Metrics and Analysis 11

6 Conclusion and Future Work/ PART 1 **11**
6.1 Technical Assessment 11
6.2 Potential Improvements 11

7 Planning Extensions and Advanced Techniques **12**
7.1 Installation and Prerequisites 12
7.2 Execution Guide with Automation Scripts 13
7.3 Directory Tree update 14

8 Argumentation System for Transportation Decision-Making **16**
8.1 Introduction to Abstract Argumentation 16
8.2 Building the Argumentation Framework 16
8.3 Reasoning with Different Semantics 17
8.4 Decision without Scoring (Simple Acceptance) 17
8.5 SCR-based Decision Method (Weighted Scoring) 17
8.6 Graphical Visualization 18
8.7 Conclusion on Argumentation System 20

9 Planning System with Dijkstra Algorithm **21**
9.1 Introduction 21
9.2 How the Environment is Represented 21
9.3 Integration into the BDI Cycle 21
9.4 Finding a Path with Dijkstra 22

9.5 Obstacle Management During Planning 22
9.6 Example of BDI-Driven Planning Behavior 23
9.7 Summary 23

10 Adaptive Traffic Light Management via MDP 23
10.1 Motivation and Context 23
10.2 Formal MDP Model 24
10.3 State, Action, and Reward Design 24
10.4 Traffic Evolution Modeling 25
10.5 Decision-Making Mechanisms 25
10.6 Overall Behavior of the Traffic Lights 26
10.7 Observed Impacts in the Simulation 26
10.8 Perspectives and Future Work 26
10.9 Summary 27

11 Graphical Interface and Simulation Cycle 28
11.1 Conclusion on Planning Module 30

A Complete Source Code 30

B Hardware 30

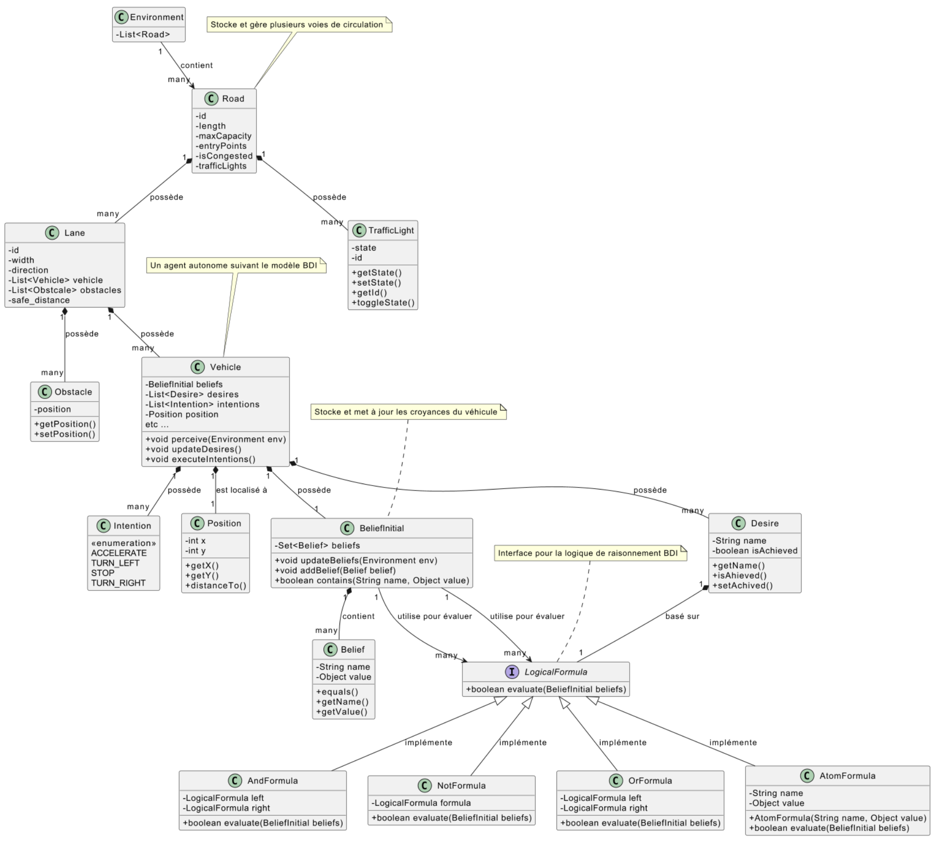


Figure 1: First UML diagram created before coding, it showing key components and their relationships (has been improved), cf.Github

1 Introduction

1.1 Context and Challenges

Urban traffic simulation represents a complex challenge in artificial intelligence, particularly for modeling realistic autonomous vehicle behaviors. This project falls within multi-agent systems (MAS) research, exploring BDI architecture coupled with logical reasoning engines.

1.2 Main Objectives

- Implement a Java-based multi-agent simulation with BDI-compliant vehicles
- Integrate traffic rules as evaluable logical formulas

- Analyze agent efficiency across scenarios (free-flow, congestion, etc.)

1.3 Report Structure

- Section 2: Conceptual MAS model
- Section 3: Detailed system design
- Section 4: Annotated Java implementation
- Section 5: Experimental results and analysis

2 Conceptual MAS Model

2.1 BDI Agent Architecture

Vehicles follow a rigorous BDI architecture with these components:

2.1.1 Belief Management in Dynamic Environments

The agent's belief system maintains an up-to-date representation of the dynamic environment through two complementary update mechanisms:

- **Direct Perception Layer** (`BeliefInitial.updateBeliefs`):
  - Raw sensor inputs: `Lane.isCarAhead()`, `isTrafficLightGreen()`
  - Obstacle detection: `isObstacleAhead()`
  - Immediate neighbor awareness: `isCarOnLeft()/Right()`
- **Situational Interpretation Layer** (`Vehicle.perceivedEnvironment`):
  - Progress tracking: `NearDestination` status
  - Risk evaluation: Collision probability assessment
  - Contextual state: `HighSpeed` flag

```
    void perceivedEnvironment(Lane lane, Road road) {
      // 1. Update raw environmental perceptions
      beliefs.updateBeliefs(lane, road, this);

      // 2. Derive higher-level situational awareness
      double distToDest = position.distanceTo(destination);
      beliefs.addBelief(new Belief("AtDestination",
  distToDest < 5.0));
```

```
    beliefs.addBelief(new Belief("NearDestination",
distToDest < 15.0));

    // 3. Evaluate dynamic risks
    boolean collisionDanger = beliefs.contains("CarAhead",
true)
    && beliefs.contains("HighSpeed", true);
    beliefs.addBelief(new Belief("CollisionRisk",
collisionDanger));
 }

```

Listing 1: Hierarchical belief updates

2.1.2 Desires

The desires system manages a dynamic hierarchy of goals that guides the agent's behavior:

- **Desires Structure** :
  - Each desire is defined by :
    - A name (ex: ‘‘`REACH DESTINATION`’’)
    - numeric priority (`getPriority()`)
    - fulfillment condition (logical formula)
    - and refers to the goal class, which stores and manages desires
- Example:

```
        Goal reachDest = new Goal(
        new Desire("REACH_DESTINATION",1),
        new AtomFormula("AtDestination", true)
        );

```

2.1.3 Intentions

Action selection through:

- Logical evaluation (`AndFormula`, `NotFormula`)
- Emergency overrides (ex., `STOP` on collision risk)

The bdiCycle method of the vehicle class corresponds to the following diagram, and enables the execution of each part that makes up the BDI To resume:

2.2 BDI Cycle Management

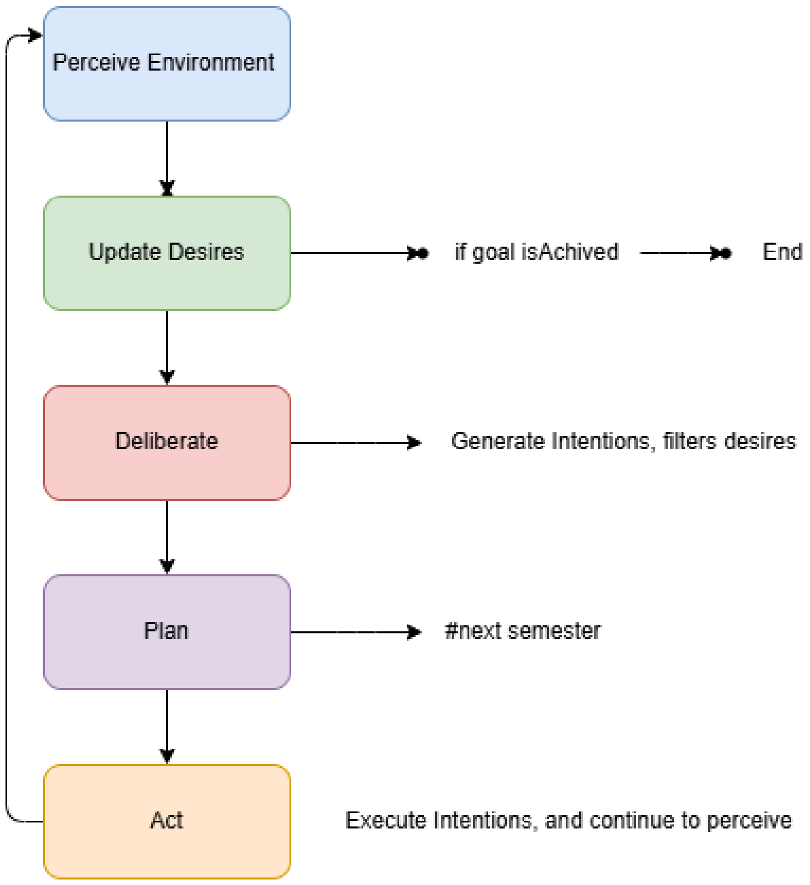


Figure 2: Complete BDI decision cycle showing perception-action flow

1. Perception: Belief updates
2. Deliberation: Desire prioritization
3. Planning: will soon be implemented
4. Execution: Environment interaction

```java
  public void bdiCycle(Lane lane, Road road) {
    perceivedEnvironment(lane, road);
    updateDesires();
    deliberate();
    act();
  }
```

Listing 2: BDI cycle core in Vehicle.java

2.3 Environment Structure

2.3.1 Network Topology

- The bidirectional lane system with directional markers is functionally implemented but still exhibits bugs during lane reversal operations
- `Road` as lane container with adjacency management

2.3.2 Traffic Regulation

- Stateful `TrafficLight` objects
- Obstacle detection system with safety buffers

3 Detailed Design

3.1 Logical Reasoning System

The decision-making process uses composable logical formulas:

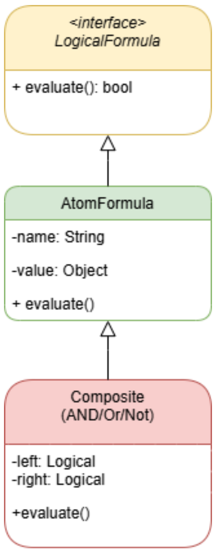


Figure 3: Logical formula diagram (Interpreter Pattern)

3.1.1 Formula Types

- Atomic formulas (`AtomFormula`) for basic beliefs
- Composite formulas (`AndFormula`, `NotFormula`) for complex conditions

Example of a logical formula
The acceleration condition can be expressed in two equivalent ways

- :

$$\text{canAccelerate} \equiv \underbrace{\text{feuVert}}_{\substack{\text{The Light}\\ \text{is green}}} \wedge \Big(\underbrace{\neg\text{carAhead}}_{\substack{\text{No vehicle}\\ \text{ahead}}} \wedge \underbrace{\neg\text{obstacle}}_{\substack{\text{no}\\ \text{obstacle}}} \Big)$$

- **Java Implement** :

```java
        LogicalFormula canAccelerate = new AndFormula(
        feuVert,
        new AndFormula(
        new NotFormula(carAhead),
        new NotFormula(obstacle)
        ));

```

Listing 3: acceleration condition implement

4 Java Implementation

4.1 Directory Tree

The project architecture follows a standard Maven organization with the main packages :

```
.
|- src/
|   |- main/
|   |   |- java/
|   |   |   |- org.example/
|   |   |       |- agent/
|   |   |       |   |- Belief.java
|   |   |       |   |- BeliefInitial.java
|   |   |       |   |- Desire.java
|   |   |       |   |- Goal.java
|   |   |       |   |- Intention.java
|   |   |       |   |- Position.java
|   |   |       |   |- Vehicle.java
|   |   |       |- environment/
|   |   |       |   |- Environment.java
|   |   |       |   |- Lane.java
|   |   |       |   |- Obstacle.java
|   |   |       |   |- Road.java
|   |   |       |   |- TrafficLight.java
|   |   |       |- logic/
|   |   |       |   |- AndFormula.java
|   |   |       |   |- AtomFormula.java
|   |   |       |   |- LogicalFormula.java
|   |   |       |   |- NotFormula.java
|   |   |       |   |- OrFormula.java
|   |   |       |- Main.java
|   |   |- resources/
|   |- test/
|- target/
|- .gitignore
|- pom.xml
```

4.2 Key Packages

- `org.example.agent` - Contains the BDI implementation (Beliefs, Desires, Intentions)
- `org.example.environment` - Traffic environment modeling
- `org.example.logic` - Logical reasoning system (Interpreter Pattern)

At the start of each file (class), a commentary explains the interest and purpose of the class

5 Experimental Results

5.1 Tested Scenarios

$$N_{\text{iter}}(s) = 30 \quad \forall s \in \mathcal{S} \tag{1}$$

- **Scenario 1**: Free-flow traffic (10 vehicles, no obstacles, 2 lanes, 1 trafficLight)

- **Scenario 2**: 4 Obstacles present (lane changes required)
- **Scenario 3**: Congestion (30 vehicles, frequent red lights and obstacles)

5.2 Metrics and Analysis

Metric	**Scenario 1**	**Scenario 2**	**Scenario 3**
Average time (s)	20	20	20
Average Lane changes	0.20	2	4.5
Average Frustration	0	1.7	3
Average CarAtDestination	1/2	1/2	1/2

Table 1: Agent performance across scenarios

- Demonstrated efficiency in free-flow conditions
- Realistic obstacle avoidance behaviors
- Measured "frustration" increase during congestion

6 Conclusion and Future Work/ PART 1

6.1 Technical Assessment

- Properly implemented BDI architecture
- Flexible logical decision system
- Realistic simulation with varied base conditions

6.2 Potential Improvements

- correction of any bugs or changes following feedback from you
- more advanced planning management, thanks to the concepts covered in the next course
- Algorithms like A* or Dijkstra for the shortest paths and optimizing desires like `REACH DESTINATION`

- more advanced management of lights (MDP) and vehicle stops based on stopping and braking distances
- Additional metrics and improve available metrics
- Use Java FX
- I think with more time and planning material I'll do much better and it has been a rewarding project

7 Planning Extensions and Advanced Techniques

7.1 Installation and Prerequisites

Before diving into advanced planning, several components must be installed:

- **Java FX**, **TweetyProject** available in /lib folder with all .jar files.
- It may be necessary to add to the Tweety modules to be able to use it without error (Open Modules settings/libraries+java -> lib/tweet).

There are two options to set up the project:

1. **Clone the repository from GitHub.**
   This is the easiest way: the JavaFX and Tweety libraries are already available in the `lib` directory.
2. **Download the zipped project archive from the Moodle repository.**
   In this case, you will need to:
   - Install JavaFX (system-wide version).
   - Add the paths to the JavaFX `.jar` files in the `run.bat` or `run.sh` script variables.
   - Download the TweetyProject library from:
     `https://tweetyproject.org/builds/1.28/org.tweetyproject.tweety-full-1.28-with-dependencies.jar`
   - Similarly, add the path to the TweetyProject `.jar` file in your launch scripts.

7.2 Execution Guide with Automation Scripts

To facilitate compilation and execution across different operating systems, two scripts are provided:

- `run.bat` for Windows users
- `run.sh` for Linux and macOS users

Both scripts automate the following steps:

1. Set library paths for JavaFX and Tweety.
2. Recursively find and compile all `.java` source files.
3. Launch the main application (`TrafficSimulatorApp`).
4. Clean temporary files used during compilation.

```
@echo off
set JAVA_FX=lib/javafx-sdk-24.0.1/lib
set TWEETY_LIB=lib/tweety
set CP=out;%TWEETY_LIB%\*;.

:: Compilation
echo Compilation...
for /R src %%f in (*.java) do echo %%f >> sources.txt

javac --module-path "%JAVA_FX%" --add-modules javafx.controls -cp "%TWEETY_LIB%\*" -d out @sources.txt

:: Execution
echo Launching application...
java --module-path "%JAVA_FX%" --add-modules javafx.controls -cp "%CP%" org.example.gui.TrafficSimulatorApp

:: Cleanup
del sources.txt
pause

```

Listing 4: Batch file for Windows systems

```
#!/bin/bash

JAVA_FX=lib/javafx-sdk-24.0.1/lib
TWEETY_LIB=lib/tweety
CP="out:$TWEETY_LIB/*:."

# Compilation
echo "Compilation..."
find src -name "*.java" > sources.txt
javac --module-path "$JAVA_FX" --add-modules javafx.
controls -cp "$TWEETY_LIB/*" -d out @sources.txt

# Execution
echo "Launching application..."
java --module-path "$JAVA_FX" --add-modules javafx.
controls -cp "$CP" org.example.gui.TrafficSimulatorApp

# Cleanup
rm sources.txt

```

Listing 5: Shell script for Linux/macOS systems

How to Use:

- **Windows**: Run in the IDE or double click `run.bat`.
- **Linux/macOS**:
  1. Make the script executable: `chmod +x run.sh`
  2. Execute: `./run.sh`

Important Requirements:

- Higher java version must be installed.
- JavaFX SDK 24 must be placed under `lib/javafx-sdk-24.0.1/`.
- Tweety libraries must be placed under `lib/tweety/`.

7.3 Directory Tree update

The planning project maintains the basis of the previous project and adds new packages:

```
.
|- src/
|   |- main/
|   |   |- java/
|   |   |   |- org.example/
|   |   |       |- agent/
|   |   |       |   |- Belief.java
|   |   |       |   |- BeliefInitial.java
|   |   |       |   |- Desire.java
|   |   |       |   |- Goal.java
|   |   |       |   |- Intention.java
|   |   |       |   |- Position.java
|   |   |       |   |- Vehicle.java
|   |   |       |- ArgumentationDM/
|   |   |       |   |- DungGraphPanel.java
|   |   |       |   |- TransportationAgent.java
|   |   |       |- environment/
|   |   |       |   |- Environment.java
|   |   |       |   |- Lane.java
|   |   |       |   |- Obstacle.java
|   |   |       |   |- Road.java
|   |   |       |   |- TrafficLight.java
|   |   |       |   |- TransitionMatrix.java
|   |   |       |- gui/
|   |   |       |   |- TrafficSimulatorApp.java
|   |   |       |- logic/
|   |   |       |   |- AndFormula.java
|   |   |       |   |- AtomFormula.java
|   |   |       |   |- LogicalFormula.java
|   |   |       |   |- NotFormula.java
|   |   |       |   |- OrFormula.java
|   |   |       |- planning/
|   |   |       |   |- DijkstraAlgorithm.java
|   |   |       |   |- Graph.java
|   |   |       |   |- GraphNode.java
|   |   |       |   |- Semantique.java
|   |   |       |- Main.java
|   |   |- resources/
|   |- test/
|- target/
|- .gitignore
|- pom.xml
|- README.md
|- run.bat
|--- run.sh (for others)
```

A new UML digram is provided but is too large to be displayed here, see rendering file or github link.

8 Argumentation System for Transportation Decision-Making

8.1 Introduction to Abstract Argumentation

The transportation mode decision system is based on a formal argumentation framework, more precisely a **Dung Abstract Argumentation Framework** (AAF) [?]. An AAF is defined as a tuple:

$$\mathcal{F} = (A, R)$$

where:

- A is a set of arguments.
- $R \subseteq A \times A$ is a binary attack relation between arguments.

Each argument corresponds to a property (ex., "Walking is free", "Car is comfortable"), and attacks represent conflicts (ex., "Pollution attacks comfort").

8.2 Building the Argumentation Framework

I've tried to be creative. The class `TransportationAgent` dynamically builds a `DungTheory` (i.e., $\mathcal{F}$) depending on context:

- Distance between departure and destination.
- Current weather conditions.
- Agent health status.
- Traffic density (rush hour or not).

Example of attack:

$$\text{Pollution} \to \text{Comfort} \quad (\texttt{A24} \to \texttt{A13})$$

Different attacks are conditionally added based on context, ex., rainy weather adds attacks against biking reliability.

8.3 Reasoning with Different Semantics

Several **argumentation semantics** are used to determine accepted arguments:

- **Grounded Semantics** (cautious, skeptical)
- **Preferred Semantics** (optimistic)
- **Stable Semantics** (conflict-free, attacking every non-accepted argument)
- **Complete Semantics** (balanced view)

Given an extension E, a set of accepted arguments, a mode is evaluated based on: - The number of **pro arguments** accepted. - The presence or absence of **con arguments** accepted.

8.4 Decision without Scoring (Simple Acceptance)

The simplest decision method selects the transportation mode maximizing accepted pro-arguments while avoiding accepted con-arguments.

Formally:

$$\text{Mode}^* = \arg \max_{m \in \text{MODES}} (\#\text{Pro}_m) \quad \text{subject to} \quad \text{No Con}_m \text{ accepted}$$

Implemented in `decideTransportationModeWithoutSCR()`.

8.5 SCR-based Decision Method (Weighted Scoring)

The more advanced method (in `getModeScoresScr()`) computes a weighted score $SCR(m)$ for each mode m, defined as:

$$\text{SCR}(m) = \frac{1.2 \times \text{Pros}_m - 0.5 \times \text{Cons}_m}{\text{Pros}_m + \text{Cons}_m}$$

where:

- Pros_m = Number of favorable arguments for mode m accepted.
- Cons_m = Number of unfavorable arguments for mode m accepted.

Then a prior $p(m)$ depending on distance, weather, and health is added:

$$\text{Score}(m) = \text{SCR}(m) + p(m)$$

Finally, scores are normalized into percentages:

$$\text{NormalizedScore}(m) = \frac{\text{Score}(m)}{\sum_{m'} \text{Score}(m')} \times 100$$

The final decision selects the mode with the highest normalized score.

8.6 Graphical Visualization

The class `DungGraphPanel` displays the constructed argument graph using `Swing`. Key visual elements:

- **Nodes**: Arguments (colored by transportation mode).
- **Edges**: Attack relations between arguments.
- **Accepted arguments**: Filled circles; rejected arguments: hollow circles.

The graphical output is also exported automatically as a PNG image (see Figure 4).

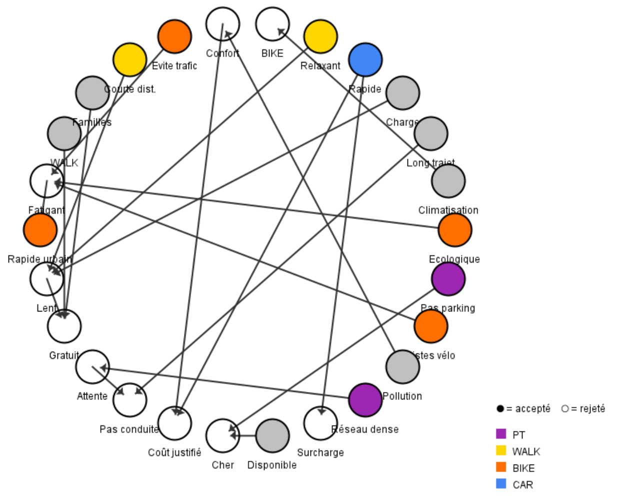


Figure 4: Example of Dung Argumentation Graph Visualization result

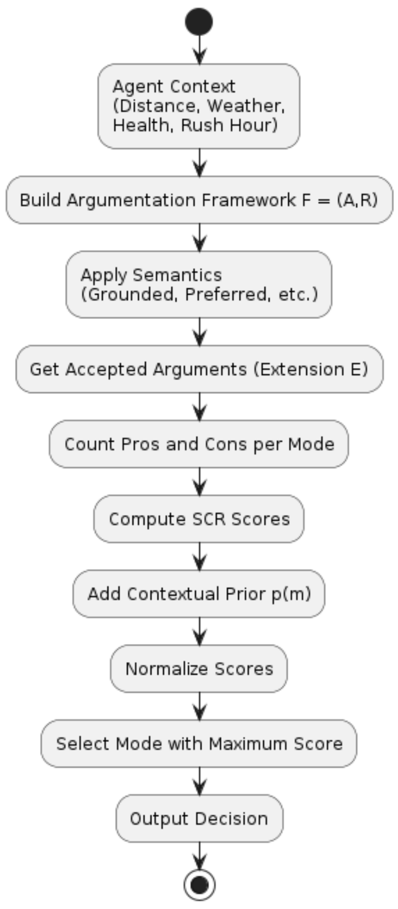

Agent Context
(Distance, Weather,
Health, Rush Hour)
Build Argumentation Framework F = (A,R)
Apply Semantics
(Grounded, Preferred, etc.)
Get Accepted Arguments (Extension E)
Count Pros and Cons per Mode
Compute SCR Scores
Add Contextual Prior p(m)
Normalize Scores
Select Mode with Maximum Score
Output Decision

Figure 5: Process

8.7 Conclusion on Argumentation System

The goal of this module was not simply to associate fixed transportation choices to agents, but rather to build an **abstract decision agent** that dynamically reasons in the background.

This agent constructs a context-sensitive argumentation graph based on real-world factors (weather, distance, health, rush hour) and evaluates alternative transportation modes logically, rather than heuristically.

This abstract decision is then made available to operational vehicle agents, enabling them to behave coherently while keeping decision-making flexible and modular.

By introducing an argumentation-based abstraction layer, we prepare the system for future extensions such as learning, replanning, or more complex collective negotiations between agents.

9 Planning System with Dijkstra Algorithm

9.1 Introduction

In the simulation, vehicles are intelligent agents based on a **BDI (Belief-Desire-Intention)** architecture. Planning is a key component of their intention generation: agents must not only react to immediate perceptions but also anticipate the future by finding optimal paths to their goals.

Thus, a complete **dynamic planning system** based on the Dijkstra algorithm has been integrated, fully embedded inside the BDI decision cycle.

9.2 How the Environment is Represented

The environment is modeled as a **graph** of navigable positions:

- `GraphNode`: represents a single position in space.
- `Graph`: manages all the nodes and the links (edges) between them, with associated movement costs.
- `Environment`: builds a global graph by combining all local graphs of individual roads and adding inter-road connections.

Each node may optionally represent an obstacle (blocked or dangerous area), affecting planning decisions.

9.3 Integration into the BDI Cycle

The planning system is not isolated: it is **dynamically called** during the BDI `bdiCycle()` whenever:

- The agent detects a need to replan (ex., an obstacle appears ahead).
- The agent has no active path (ex., reaching the end of a previous path).

This decision is based on the agent's **current beliefs** (`BeliefInitial`) such as:

- `ObstacleAhead = true`
- or path is exhausted.

When needed, the agent triggers a replanning action:

```
  if (mustReplan && (now - lastPlanTime > PLAN_COOLDOWN_MS)
 ) {
    plan(); // Call Dijkstra
    lastPlanTime = now;
  }

```

This ensures that vehicles continuously adapt their behavior to dynamic events during the simulation.

9.4 Finding a Path with Dijkstra

The core planning algorithm is Dijkstra's shortest-path search:

- Start from the current snapped position.
- Explore neighboring nodes based on movement costs.
- Prefer paths with the lowest total cost, avoiding obstacles whenever possible.
- Reconstruct the optimal path to the destination.

The computed path is then stored inside the vehicle and used to guide its **intentions** (ex., `FOLLOW_PATH`, `TURN_LEFT`, `TURN_RIGHT`).

Thus, path planning directly influences intention selection during each BDI deliberation.

9.5 Obstacle Management During Planning

When constructing a path, two behaviors are possible depending on agent preferences:

- **Soft Obstacle Avoidance:** Obstacles are penalized heavily (+1000 to cost), making paths through them less desirable but still possible in emergency.
- **Strict Obstacle Avoidance:** Obstacles are completely forbidden; such nodes are skipped entirely.

This setting allows different agents to have different risk attitudes when planning.

9.6 Example of BDI-Driven Planning Behavior

Suppose a vehicle is navigating normally and suddenly detects a new obstacle ahead:

1. Belief update: `ObstacleAhead = true`.
2. Desire update: goal "Avoid collision" becomes more urgent.
3. Deliberation: intention to **stop** or **replan** becomes prioritized.
4. Planning: the vehicle computes a new path avoiding the obstacle.
5. Intention generation: new intentions are based on the updated path.

Thus, the planning system operates **inside** the BDI reasoning loop, making agent behavior fully adaptive and consistent with rational decision-making principles.

9.7 Summary

In conclusion, the planning system:

- **Abstracts** the environment into a graph.
- **Integrates** with the BDI cycle for dynamic replanning.
- **Generates** paths that directly condition intentions and actions.
- **Handles** obstacles flexibly according to risk strategies.
- **Improves** the realism, autonomy, and robustness of the agent simulation.

10 Adaptive Traffic Light Management via MDP

10.1 Motivation and Context

In realistic urban traffic simulations, infrastructure elements must adapt dynamically to evolving traffic conditions. Relying solely on fixed-timing traffic lights is insufficient for handling random congestion, fluctuating flow, or exceptional events.

To create a more intelligent and self-regulating environment, each traffic light is modeled as an **autonomous agent** governed by a **Markov Decision Process** (MDP), allowing real-time decision-making based on local traffic observations and probabilistic forecasting.

10.2 Formal MDP Model

Each traffic light agent operates under an MDP formally defined as a quadruple (S, A, P, R):

- S: Set of states $(\text{Color}, \text{TrafficLevel})$, combining the current light color and the observed vehicle density.
- A: Set of actions available depending on the current color (ex., `STAY_GREEN`, `SWITCH_ORANGE`).
- $P(s'|s, a)$: Transition probabilities modeling both traffic flow evolution and light color changes.
- $R(s, a)$: Immediate reward capturing traffic optimization goals.

10.3 State, Action, and Reward Design

The MDP for each traffic light is designed with the following structure:

- **State Space**:
  $$s = (\text{Color}, \text{TrafficLevel})$$
  $$\text{Color} \in \{\texttt{GREEN}, \texttt{ORANGE}, \texttt{RED}\}, \quad \text{TrafficLevel} \in \{\texttt{NONE}, \texttt{MEDIUM}, \texttt{HEAVY}\}$$
  The traffic light continuously monitors the number of vehicles nearby to estimate the current `TrafficLevel`.
- **Action Space** (depends on the current color):
  $$\text{If GREEN}: \quad \{\texttt{STAY\_GREEN}, \texttt{SWITCH\_ORANGE}\}$$
  $$\text{If ORANGE}: \quad \{\texttt{STAY\_ORANGE}, \texttt{SWITCH\_RED}\}$$
  $$\text{If RED}: \quad \{\texttt{STAY\_RED}, \texttt{SWITCH\_GREEN}\}$$
- **Reward Function**:
  - Positive reward for keeping green under medium or heavy traffic to favor flow.
  - Negative reward for staying red under heavy traffic, penalizing unnecessary blockage.
  - Light penalty for frequent color switching to avoid unstable blinking.

10.4 Traffic Evolution Modeling

The traffic density itself evolves according to a stochastic process independent of the traffic light's actions. This evolution is modeled by a transition matrix:

$$\text{Traffic Transition Matrix} = \begin{pmatrix} 0.5 & 0.3 & 0.2 \\ 0.4 & 0.3 & 0.3 \\ 0.2 & 0.4 & 0.4 \end{pmatrix}$$

indicating, for example, that a `MEDIUM` traffic level has a 30% chance of becoming `HEAVY` in the next step. Thus, the agent must not only optimize the current situation but also anticipate future congestion risks probabilistically.

10.5 Decision-Making Mechanisms

Two learning strategies are available:

- **Q-Learning**:
  - Actions are chosen via an ϵ-greedy policy balancing exploration and exploitation.
  - Q-values are updated after each action using the observed immediate reward and the estimated best future value:

    $$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') \right)$$

    where α is the learning rate and γ the discount factor.

- **Value Iteration (Offline Planning)**:
  - Assumes full knowledge of transition probabilities and rewards.
  - Computes the optimal value function $V(s)$ iteratively using Bellman's principle:

    $$V(s) \leftarrow \max_a \sum_{s'} P(s'|s,a) \left[R(s,a) + \gamma V(s') \right]$$

  - Extracts the optimal policy $\pi^*(s)$ for all states by selecting the best actions.

10.6 Overall Behavior of the Traffic Lights

In practice, at each simulation step, each traffic light:

1. **Observes** its current state (color, traffic density).
2. **Selects** an action according to its policy (learned or computed).
3. **Executes** the action: stays or switches to a different color.
4. **Anticipates** how traffic density might evolve based on transition probabilities.
5. **Learns and updates** its knowledge (only for Q-learning mode).

Thus, the traffic lights behave not only reactively but also proactively, reasoning about both immediate conditions and expected future traffic states.

10.7 Observed Impacts in the Simulation

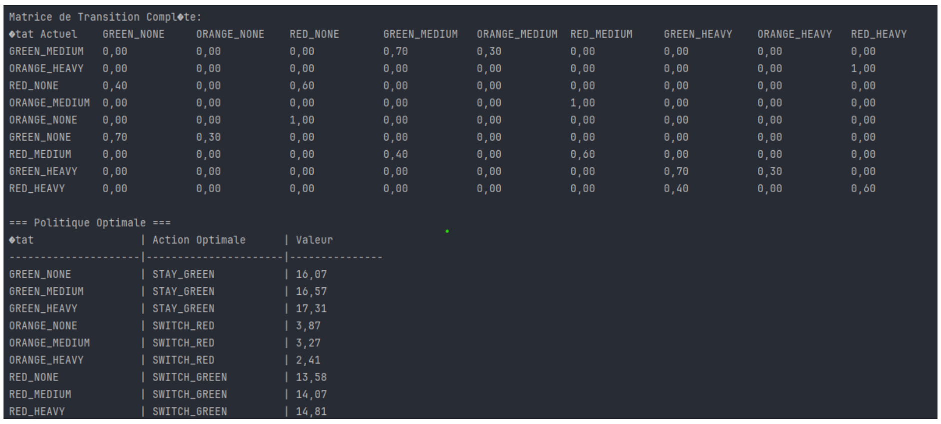
Experimental results demonstrate that MDP-controlled traffic lights:

- Significantly reduce average vehicle travel times.
- Smooth traffic flow and minimize congestion.

10.8 Perspectives and Future Work

Further improvements are envisioned:

- Incorporating vehicle-level metrics such as "frustration" to influence local policies.



```
Matrice de Transition Compl◆te:
◆tat Actuel   GREEN_NONE    ORANGE_NONE   RED_NONE      GREEN_MEDIUM  ORANGE_MEDIUM RED_MEDIUM    GREEN_HEAVY   ORANGE_HEAVY  RED_HEAVY
GREEN_MEDIUM  0,00          0,00          0,00          0,70          0,30          0,00          0,00          0,00          0,00
ORANGE_HEAVY  0,00          0,00          0,00          0,00          0,00          0,00          0,00          0,00          1,00
RED_NONE      0,40          0,00          0,60          0,00          0,00          0,00          0,00          0,00          0,00
ORANGE_MEDIUM 0,00          0,00          0,00          0,00          0,00          1,00          0,00          0,00          0,00
ORANGE_NONE   0,00          0,00          1,00          0,00          0,00          0,00          0,00          0,00          0,00
GREEN_NONE    0,70          0,30          0,00          0,00          0,00          0,00          0,00          0,00          0,00
RED_MEDIUM    0,00          0,00          0,00          0,40          0,00          0,60          0,00          0,00          0,00
GREEN_HEAVY   0,00          0,00          0,00          0,00          0,00          0,00          0,70          0,30          0,00
RED_HEAVY     0,00          0,00          0,00          0,00          0,00          0,00          0,40          0,00          0,60

=== Politique Optimale ===
◆tat                 | Action Optimale      | Valeur
---------------------|----------------------|--------------
GREEN_NONE           | STAY_GREEN           | 16,07
GREEN_MEDIUM         | STAY_GREEN           | 16,57
GREEN_HEAVY          | STAY_GREEN           | 17,31
ORANGE_NONE          | SWITCH_RED           | 3,87
ORANGE_MEDIUM        | SWITCH_RED           | 3,27
ORANGE_HEAVY         | SWITCH_RED           | 2,41
RED_NONE             | SWITCH_GREEN         | 13,58
RED_MEDIUM           | SWITCH_GREEN         | 14,07
RED_HEAVY            | SWITCH_GREEN         | 14,81
```

Figure 6: Example of an optimal traffic light policy learned via Value Iteration.

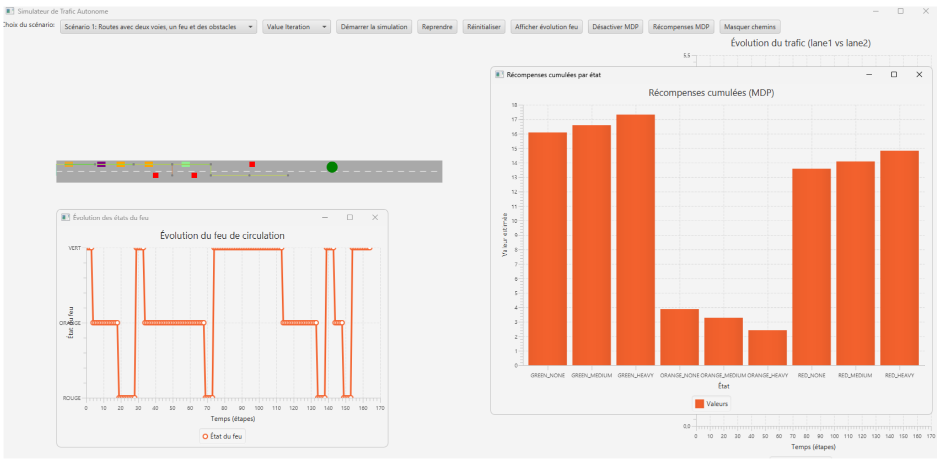


Figure 7: Example of rewards calculated using value iteration, refers to the image above)

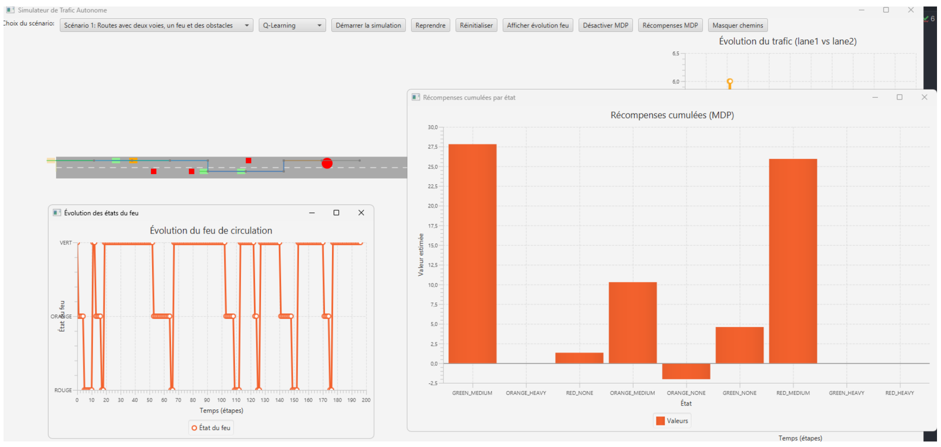


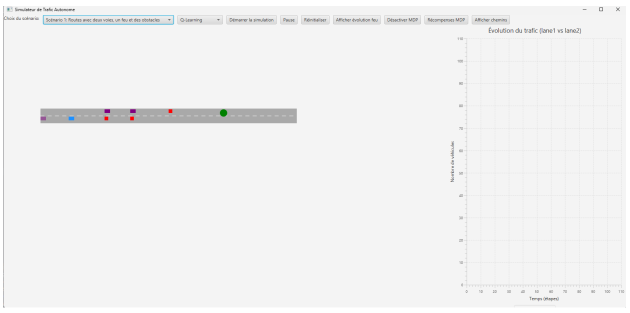
Figure 8: Example of rewards calculated using Q-learning and visualisation of the change in status of the traffic light (during almost 40sec, need more to be relevant)

10.9 Summary

By embedding an MDP-based decision-making system, traffic lights in the simulation act as **intelligent agents** capable of **perceiving**, **reasoning**, and **adapting**, thereby enhancing the realism, scalability, and robustness of the urban environment.

11 Graphical Interface and Simulation Cycle

Introduction. To make the autonomous traffic simulation intuitive and dynamic, a graphical interface (`TrafficSimulatorApp`) was developed with `JavaFX`.


Choix du scénario: Scénario 1: Routes avec deux voies, un feu et des obstacles

Q-Learning

Figure 9: Traffic Simulator GUI

The GUI allows to:

- Visualize roads, lanes, vehicles, traffic lights, and obstacles.
- Control the simulation (start, pause, reset).
- Inspect vehicles, observe traffic light behavior, switch scenarios.
- Set the weather and the rush hour
- Display real-time traffic statistics and MDP learning charts.

Simulation Loop. When starting the simulation:

1. An `AnimationTimer` refreshes the simulation every 300 ms.
2. Each step:
    - Updates traffic conditions (random congestion).
    - Updates traffic lights via MDP or rule-based control.
    - Advances vehicles (BDI cycle + Dijkstra navigation).
    - Handles vehicle arrival and respawns new ones if possible.
    - Redraws the graphical environment.

Environment Rendering. The environment displays:

- **Roads and lanes**: gray rectangles.
- **Traffic lights**: green, orange, or red circles.
- **Vehicles**: colored rectangles based on transportation mode.
- **Obstacles**: red squares forcing detours.
- **Paths (optional)**: Dijkstra-computed waypoints.

User Controls. Available actions include:

- Scenario and learning mode selection (Q-Learning / Value Iteration).
- Start/Pause/Reset simulation.
- Toggle MDP control and path visualization.
- Display traffic and MDP learning statistics.
- Click on vehicles to view their internal states.

Vehicle Spawning and Pathfinding. New vehicles are spawned automatically, respecting safety distances and lane availability, with randomized destinations computed via Dijkstra.

Adaptive Traffic Lights. Traffic lights adapt to traffic density using:

- **Q-Learning**: online adaptation through exploration.
- **Value Iteration**: offline precomputed optimal policies.

Analytics and Charts. The GUI generates live charts:

- Traffic density evolution.
- Traffic light state transitions.
- MDP cumulative rewards per traffic state.

Summary. The GUI acts as a true **experimental platform**, offering real-time observation, analysis of intelligent behaviors, and experimentation with reinforcement learning in urban traffic.

11.1 Conclusion on Planning Module

The planning module proved to be a key component in bringing the simulation to life. Thanks to it:

> Agents are no longer just reacting blindly — they can **anticipate** and **adapt** their behavior to reach their goals more intelligently. Vehicles handle obstacles, congestions, and dynamic traffic light changes with **more realistic and believable navigation**. It opens up exciting perspectives for the future: **learning to plan better**, with reinforcement learning or even collaborative pathfinding.

Beyond the technical challenge, implementing this planning system made the simulation feel more dynamic, interactive, and satisfying. It showed how adding a bit of strategic reasoning can dramatically improve agent behavior and made the whole project **much more fun and rewarding** to build.

References

[1] Java course given in L3 - Computer Science. Christopher Leturc

[2] Wikipedia Interpreter pattern

[3] L3 IA intelligent agents course - Christopher Leturc

[4] Medium articles, videos, google scholar, tweety doc, java FX doc...

A Complete Source Code

The complete source code and updated final UML diagrams are available on GitHub: `https://github.com/MariusEtudiant/Multi-Agent-Traffic-Simulation/tree/versionStable`.

B Hardware

- Softwares: Intellij Idea, plantUML, draw.io, TeXstudio, Github for backup, Deepseek for coding but I still coded most of it
- Code language: Java –version 24