Multi-agent Systems Project Report: Traffic Simulation based on Real-Time Information

Konstantin Kloster[1] and Oliver Berg[1]

[1]Technical University Kaiserslautern

November 24, 2018

Abstract

This paper presents a novel multi-agent system to simulate traffic agents to then visualize and evaluate performance through unified information formats and losely connected components. We show that car-agent traversing a network of nodes and links perform considerably better when utilizing global network information as compared to local static graph information. This resembles that usage of dynamic real-time information benefits traffic agents in planning. The impact of global network-wide events has impact on both groups of agents, where global agents are - whilst being more flexible and resistant in general - still vulnerable to unfortunate circumstances.

Contents

1 Introduction **3**

2 Related Work **4**

3 MAS Traffic Simulator Backend **6**

3.1 Backend Architecture 6

3.2 Implemented Agents 7

3.3 Utilized Graph 9

4 Web-based Simulation Frontend **11**

4.1 Persisted Logging Information 11

4.2 Frontend Architecture 13

4.3 User Interface Experience 13

4.4 Web-based Animations with D3 15

5 Agent Performance Analysis **17**

5.1 Evaluation Procedure 17

5.2 Performance Results 18

6 Conclusion **19**

1 Introduction

Traditionally, real-time information (RTI) was used by transit providers for operation and control. In today's age of always-on telecommunication devices and sensor input of an armada of connected input devices through the "Internet of Things" (IoT) movement, this information is also increasingly utilized by travellers themselves for route planning. This includes car routes as well as coordinating public transport timetables or available ride-sharing resources.

In this work, we will simulate, display and analyze the dynamic movement and interaction of travelling & planning agents within a traffic coordination setting.

To this end, a multi-agent system (MAS) is implemented which takes formal network graphs and specific user input regarding the simulation scale to then simulate a continuously spawning group of car-agents traversing the network with the goal to arrive at an assigned location as fast as possible. An internal coordination unit in the form of a planner-agent distributes route information according to the type of agent it is being queried from. The architecture setup is based on propositions from provided sources ([Mastio et al., 2015], [Brakewood and Watkins, 2018]). Random events altering the state of the network take place to simulate traffic incidents and to provoke agents to deviate from initial plans.

Having run the simulation and persisted the simulation run in respective log-files, a web-based frontend implementation reads network- and log-files to visualize the network graph and temporal agent-behavior. The visualization allows for selection of graph and run-data and displays the read and interpreted data. This should allow the observer to understand the car-agent's behavior and make numeric analysis results more visually interpretable.

This report is structured as follows: Following this introductory description of context and task, section 2 will inspect related work surrounding the field of RTI traffic simulation and derived research and system implementation propositions. These are then being put into application context in section 3 where the implemented MAS and respective architectural constraints are outlined. Next, section 4 continues to describe the frontend visualization and transitions into the traffic performance analysis in form of agent arrival-time metrics being is read from persisted system logs and put into perspective within this report's section 5. Finally, section 6 presents the findings and concludes this project's report.

2 Related Work

Simulating development of traffic is a well-traversed research topic regarding scheduling and network traversal simulation problems. Throughout application, dedicated simulators have been applied frequently and early-on like the popular "Multi-agent Transport Simulator" (MATSim) [Sezen, 2003] or "Repast Simphony" [Zargayouna et al., 2013] alongside dedicated extensions like the Symphony-based "SM4T Simnulator" [Ksontini et al., 2016]. Such applications provide advantages like automated timetables for public transport, advanced types of travelling agent and unified logging formats for simulation runs. Resorting to holistic solutions like the above mentioned is especially useful for non-technological research regarding behavioral analysis or city planning [Brakewood and Watkins, 2018]. If the multi-agent system aspects are predominant though (as is the case in this work), adapting similar implementation structures whilst doing the actual agent implementation work (done here) proves beneficial.

In contrast to fully incorporated applications, some approaches in literature already shift focus towards surrounding aspects surrounding of traffic simulation, like focusing on the distributing simulation [Mastio et al., 2015].

As of [Mastio et al., 2015], which depicts the simplest yet most fundamental approach to general network traversal, it utilizes a basic graph structure with vertices (also called "nodes") as intersection points and edges (also called "links") connecting these intersections. Agents are assigned a path over a fixed set of vertices as the shortest path over weighted edges. Path updates may occur only upon arrival at a given vertex when the agent then queries for a possible updated path. The travel time t on an edge depends on the number of agents currently on the edge in question. Calculating this t can be done based on different kinds of functions modelling e.g. a certain free-flow capacity where for a given n number of vehicles the weight (t) of an edge will not be impacted and only after reaching a certain threshold number n_{thresh} the weight will increase to depict a slower movement (to a degree of α) of traffic along said edge; exemplary formulas can, for example, be found in [Mastio et al., 2015] or [Ksontini et al., 2016].

To this agent-centric travel choice procedure, a literature review as is being depicted in the transport review of [Brakewood and Watkins, 2018] adds a theoretical framework of traveler agent's perspective concerning travel- and mode choice as well as choices regarding route, boarding and departure. Incorporating this thinking, one ends up with a tight path-choosing procedure across multiple channels which theoretically boils down to the simple procedure of the previously mentioned formulas adapted to modes, routes and fixed schedules. As this framework is a theoretical methodology at heart, the practical implementation aspect of this formal procedure raises multiple performance and architectural issues which without utilizing afore-mentioned well-established holistic framework solutions is expected to introduce bottlenecks.

As proposed by [Zargayouna et al., 2013], agents adhere to a specific simulation workflow where they continuously query for updates regarding path choice whenever reaching nodes of the underlying network. Opposing the precise setup depicted here and utilizing in addition the thinking of [Mastio et al., 2015], no precise time-step-based approach of procedurally checking all agents for their position but rather an event-based truly (distributed) multi-agent approach is implemented where agents query after complete time lapses instead of single short global time intervals.

The visualization procedure of a given simulation then needs to adhere to the time lapse nature of the event logs, which not necessarily adheres well to standards defined by holistic framework solutions (like used in [Zargayouna et al., 2013][Ksontini et al., 2016]). As visualization is a side-aspect of most MAS-focused implementations (see [Mastio et al., 2015]), this is generally speaking seen as an added bonus for both debugging and reporting purposes.

Regarding performance metrics of a traffic simulation, metrics like use-of-transit, satisfaction and travelling-time are frequently proposed ([Brakewood and Watkins, 2018]). With the use of dedicated car- and planner-agents (alongside [Zargayouna et al., 2013]) and the omittance of public transport time-table-based scheduling / availability-based carsharing, this leaves travelling-time as well as deviation from planning to execution performance as major indicators for performance. In addition to this, subjective monitoring of network behavior may enhance the result ([Brakewood and Watkins, 2018], [Zargayouna et al., 2013]).

3 MAS Traffic Simulator Backend

This section describes the backend part of the implemented MAS traffic simulator. First, we give an overview of the MAS architecture. In subsection 3.2 we characterize the different types of implemented agents. Finally, this section ends with an outline of the simulation.

3.1 Backend Architecture

Several multi-agent traffic/transport simulation frameworks were mentioned in section 2, but for this project, we decided to implement our own traffic simulation. Building a multi-agent system from the ground up can be a laborious task. Therefore, it seemed to be a good idea to utilize a multi-agent framework. The most popular one is probably the JAVA Agent DEvelopment Framework (JADE). Nevertheless, we wanted to implement the simulation in the Python programming language. We tried osBrain and SPADE and chose the latter one because of it's subjectively better usability.

SPADE allows building concurrent multi-agent systems by utilizing the Python asyncio library. For agent communication, it uses the Extensible Messaging and Presence Protocol (XMPP). Thus, it is necessary to run an XMPP server in order to use SPADE. We chose Prosody as recommended by SPADE, but it is possible to use any other XMPP server. Using XMPP might be useful when agents not only need to communicate with other agents but also with humans.

Furthermore, SPADE supports the FIPA Agent Communication Language (ACL). While we incorporate FIPA ACL parameters like sender, receiver, performative and content in our agents, we only use a couple of performatives like receive and inform. But those are not critical to the functionality of our agents.

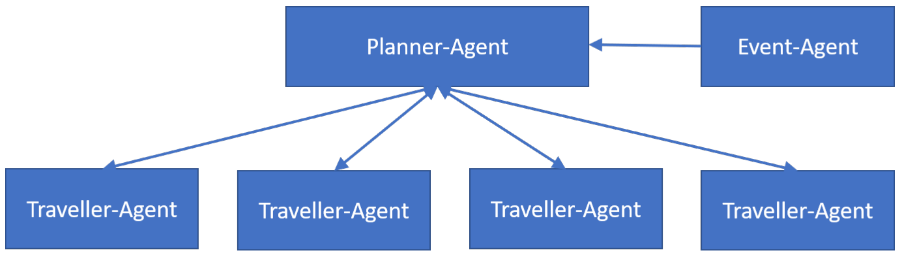


Figure 1: Backend architecture overview

To achieve the goals for this project, we decided to use an architecture that focuses on a planner agent. All traveller agents connect to the planner agent in order to receive real-time information while moving on a graph. An event agent is able to trigger events in the simulation. Figure 1 gives a rough overview of the architecture. The following section will describe the different agent types in detail.

3.2 Implemented Agents

3.2.1 Planner Agent

The planner agent is responsible for keeping track of the traveller agents and their positions on the graph. When started, the planner agent loads a graph. It is assumed that the distances between nodes and the maximum capacities k_c of the edges are defined. Next, the free flow speed α and the congestion speed β have to be specified. While traveller agents can also be spawned with a specific maximum free flow speed, for simplicity reasons, we decided to use a global approach. This approach can be compared to speed limits on highways.

The planner agent is able to provide travellers with shortest paths and current possible travel times. In order to fulfill this function, the planner updates the travel times each time an agent enters or exits an edge in the graph. The travel time for an edge can be computed with $t = \frac{distance}{s}$ where the speed s is calculated by Equation 1 in dependence of the current edge density k like proposed by [Mastio et al., 2015]. Additionally, we set the minimum speed to 10.

$$s = \begin{cases} \alpha, & \text{if } k \leq k_c \\ -\beta(k - k_c) + \alpha * k_c, & \text{otherwise} \end{cases} \tag{1}$$

Figure 2 shows an example with $\alpha = 100$, $\beta = 20$, and $k_c = 20$.

Besides, the planner agent also logs all interactions with the travellers and the event agent. For example, when a traveller receives a route or a travel time for an edge, the planner saves this information. If the same agent receives an updated route or travel time, the planner registers those changes. Such information can be used to analyze the traffic simulation and also to visualize it as described in section 4.

Technically, the planner agent is utilizing SPADE's cyclic behavior. The cyclic behavior allows an agent to perform an action repeatedly. In our case, he listens on several endpoints for messages from other agents and processes them.

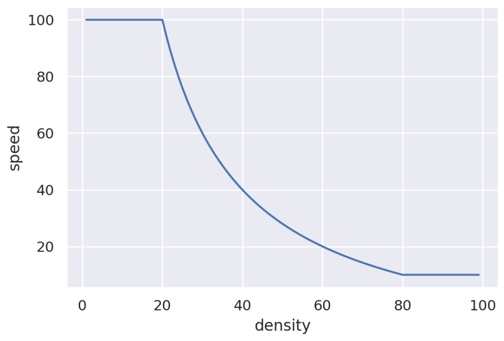


Figure 2: Speed in relation to edge density

3.2.2 Traveller Agents

We implemented two different types of traveller agents. Local traveller agents compute their optimal route only at the beginning of their journey. They can only access information from the initial graph like edge distances for example. Global agents update their route at each node, incorporating the current traffic situation. Traveller agents can be initialized with predefined start and destination nodes. Alternatively, start and destination node can also be selected randomly from the set of nodes in the graph.

Both agent types are built on top of SPADE's finite state machine behavior. When an agent is created, he starts in the *spawn* state. In this state, he will send a message with the current timestamp to the planner agent. The planner agent will use this information for logging reasons. If a global agent is in the *get_route* state, he will send a message to the planner agent with his current node and his destination node. After that, he will wait for an answer from the planner agent containing the next node and the maximum speed. Local agents do not get route updates. They will only receive the maximum speed.

Next, the traveller agent will send another message to the planner to inform him that he is going to start his journey. The planner will increment the density of the edge and recompute the maximum speed. When in the *drive* state, traveller agents will compute the travel time necessary to traverse the current edge and sleep for this duration. In the *end_edge* state, travellers send another message to the planner. The planner will update the maximum speed of the edge accordingly. If the reached node is equal to the destination node, the agent will transition into the *arrived* state. Otherwise, he will go back into the *get_route* state. In the *arrived* state, the traveller agent will send a last message to the planner for logging reasons. Figure 3 gives a graphical overview of the implemented finite state machine.

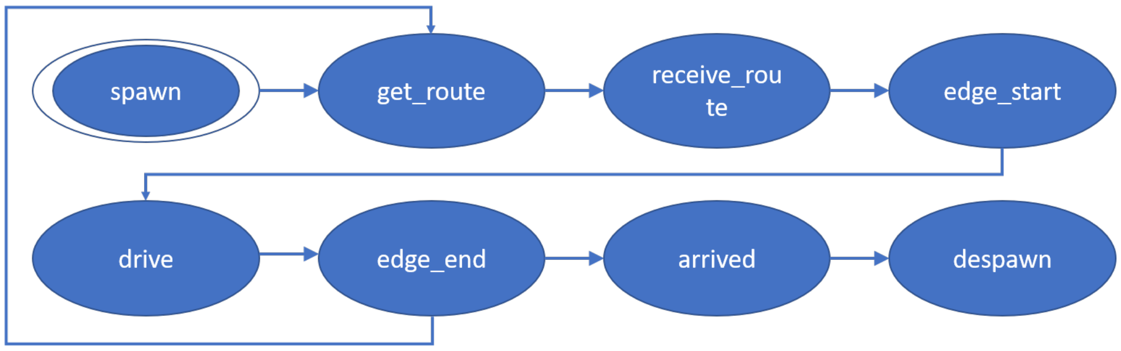


Figure 3: Finite state machine for the traveller agent

3.2.3 Event Agent

The event agent is able to trigger different events which can have an impact on the simulation in several ways. The *road_works_start* event selects an edge in the graph and a factor in the range from 0.2 to 0.9 randomly. This information will be sent to the planner agent. The planner agent will multiply the maximum speed of the edge with the selected factor thus the travel times on this edge will be raised. Such roadworks last until the *road_works_end event* is performed by the event agent. This event resets the factor of an affected edge to 1.

The *spawn_random_agents* event spawns a random amount of agents with randomized start and destination nodes. An arbitrary amount of agents with the same start and destination node will be spawned by the *spawn_agents event*.

The period, at which events occur and also the maximum number of events per simulation can be adjusted in the *file event_agent.py*.

3.3 Utilized Graph

Traffic networks are not randomly generated. In most cases, they are elaborately planned networks that have to meet the requirements of high traffic volumes. That is why we avoid random graphs and instead use so-called scale-free networks. Specifically, we use the Barabási-Albert model from the NetworkX package. In this model, a node is more likely to be connected with a node that already has many edges. Such a node can be compared to a large motorway junction.

The file *graphcreator.py* generates Barabási-Albert graphs [Barabási, 1999] which can be used with our simulation. It will assign a distance to the edges ranging from 1 to 1000. The capacity of the edges will be chosen according to a given distribution. The distribution can be adjusted by changing the global variable *CAPACITIES*.

Finally, to use a created graph in the simulation, the path to the graph has to be set in *simulation.py*. The simulation will create several log files. The following section will describe how they are used by the frontend.

4 Web-based Simulation Frontend

Following the in section 3 described MAS architecture, this section now presents the web-based frontend application visualizing a simulation run based on the persisted simulation run logging information and other provided miscellaneous resources.

4.1 Persisted Logging Information

The frontend application reads graphs from json-files as well as a corpus of simulation-run logging files.

4.1.1 Graphs

The network graph json-files are generated via the Python framework GraphX and then stored as json-files in the project folder. These are then being read from the MAS backend as well as the frontend-application. The file format can be seen in the example of figure below.

```json
{
    "directed": false,
    "graph": {
        "name": "A Testing Graph"
    },
    "links": [
        {"source": 0, "target": 1, "value":1},
        {"source": 1, "target": 0, "value":2}
    ],
    "multigraph": false,
    "nodes": [
        {"id": 0, "color": "purple", "size": 16},
        {"id": 1, "color": "green", "size": 9}
    ]
}
```

Figure 4: Example of GraphX output format (JSON)

These graphs are being read form disc and parsed into program structures (backend) or exposed to an API (frontend, see subsection 4.2).

They do however only depict the formal structure of the graph, which is all the backend needs to simulate agent paths. For a proper visualization of the graph on the 2D image plane, the web-frontend additionally utilizes some formal graph-drawing algorithms (see subsection 4.4).

4.1.2 Simulation-Logs

The simulation logs are the actual output of the traffic simulator and come on a collection of multiple associated logs:

- *[id]-[graphId]-carAgents.log*
- *[id]-[graphId]-plannerAgent.log*
- *[id]-[graphId]-events.log*

Each log is named by a unique id which is unique only across multiple sets of different logs but coherent across all associated logs. This allows to map each carAgents-log to its respective event-log and plannerAgent-log. The graph-id is specified for each set of associated logs in order to map the utilized graph to the logs. Logs of a respective simulation run can only be visualized for the specified graph it was run on.

Each log-file has lines for any event occurring at a specified point in time (indicated with the line's initial timestamp-value) and multiple values separated with semicolons.

The **carAgents-log** holds lines structured with values of: *timestamp* (ts), *action* ("spawn" / "enter" / "reach" / "despawn"), *agentID* and depending on the action some more attributes. It as such depicts the events thrown by car-agents and logged to reason about their timely execution. Action-type "spawn" gets additional attributes *spawnNode* and *agentType* to specify where and which kind of (car-)agent was spawned. Action-type "enter" gets additional attributes *linkEdge-From*, *linkEdgeTo* and *linkValue* to specify which edge the agent entered and at which speed he will traverse the link. Action-type "reach" gets the additional attribute *node* to declare which (final) node the agent has just reached. Action-type "despawn" finally does not get additional attributes as this action only indicates to remove the agent from all later considerations.

The **plannerAgent-log** holds lines structured with values of: *timestamp* (ts), *action* ("init" / "update" / "reroute"), *agentId*, *routeNodes* and *routeLinkValues*, where the both "routeNodes" and "routeLinkValues" are lists of ids / numbers respectively indicating graph edges and edge weights of paths. As such this log captures the path planning done by the planner-agent for car-agents on initialization, updating of path link weights when reaching vertices or rerouting with new vertices and edges all together.

The **events-log** holds lines structured with values of: *timestamp* (ts), *listOfEdges* and *factor*, where the "listOfEdges" presents a list of nodes that identify links between them to be impacted in their weight / speed by the provided "factor". As such, the events viable for the application include all types of events that impact a given set of linked concerning travelling speed. This type of event was identified as being the most general one whilst having a major impact in route planning and provides intuitive real-world counterparts (e.g. break-down of a node slows down all traffic to that node, break down in a rode is a single link being impacted, new road means a single link becomes faster to use).

4.2 Frontend Architecture

The overall frontend application's architecture is based on a traditional Model-View-Controller setup implemented in NodeJS. The webserver is setup to listen to port 3000 and receive http requests to then respond with the corresponding page content.

Internally, an API plays the role of the controller-component connecting the view-layer to the model. This interface is being utilized primarily to expose local file data to the end-user via a unified interface, preemptively resolving possible difficulties connecting the frontend application to the backend result or later adding further functionality based on said data. It is used to return json-formated data on graphs, logs and other miscellaneous internal status information.

The API primarily talks to the *masSimulatorConnection* which reads, formats and persists logging and graph information as it is being setup in the web-UI. Please note that as such the web-page is not quite capable to handle multi-user when being deployed, but rather a single user entity.

Through the dedicated intermediate API layer, access control for resources as requested by the end-user application would be both possible and straight -forward to implement later on if needed.

4.3 User Interface Experience

The exposed User Interface depicts a web-page written in plain HTML5, CSS3 and JavaScript with the help of common libraries such as *jQuery* and *Bootstrap* for simple best-practice setup

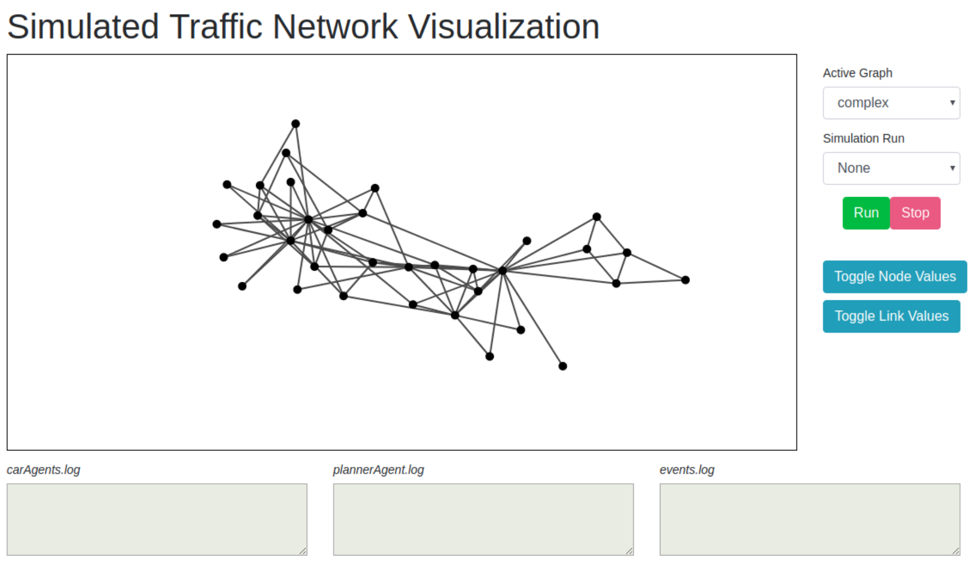


Figure 5: Web-UI upon initial loading of the page

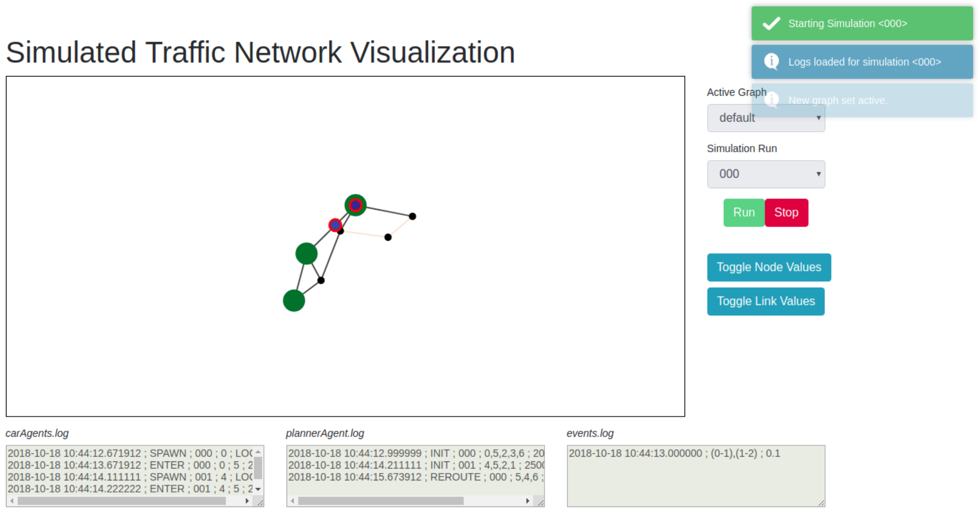
in addition to *toastr* for notifications (see Figure 5).

For canvas-based visualization and especially animation, the visualization library *D3.js* was utilized (more on that in subsection 4.4).

The navigation flow adheres to basic top-to-bottom / left-to-right principle: The network graph is displayed in the canvas to the upper left, from which the selections take place to its right via selection panels for graph and simulation run. Controls regarding run visualization are positioned directly under the run selection with appropriate colors for starting and stopping. General non-intrusive display options (turning on Node-ID / Link-ID displays) are presented in more neutral blue color and positioned below the other primary controls.

Below the basic visualization panel, textareas are used to display retrieved current-run logs (conforming to the conventions established in subsubsection 4.1.1). These serve the purpose to more intuitively display the information available from the simulation for visualization, as well as give instant feedback to selections made regarding the simulation run. Selections made regarding graph network immediately prompt a redrawing of the new network. All selections and button presses yield immediate visual feedback (via onchange-events).

To ease the usage of the frontend navigation elements, they are en-/disabled based on the state



```
2018-10-18 10:44:12.671912 ; SPAWN ; 000 ; 0 ; LO
2018-10-18 10:44:13.671912 ; ENTER ; 000 ; 0 ; 5 ; 2
2018-10-18 10:44:14.111111 ; SPAWN ; 001 ; 4 ; LO
2018-10-18 10:44:14.222222 ; ENTER ; 001 ; 4 ; 5 ; 2
```

```
2018-10-18 10:44:12.999999 ; INIT ; 000 ; 0,5,2,3,6 ; 20
2018-10-18 10:44:14.211111 ; INIT ; 001 ; 4,5,2,1 ; 250
2018-10-18 10:44:15.673912 ; REROUTE ; 000 ; 5,4,6 ;
```

```
2018-10-18 10:44:13.000000 ; (0-1),(1-2) ; 0.1
```

Figure 6: Web-UI with running visualization

of the application. As such, only simulation runs belonging to the currently active (drawn) graph are enabled for selection in the second select, the run-button only gets enabled with a drawn graph and selected run and the stop-button is disabled as long as no simulation is running.

4.4 Web-based Animations with D3

The underlying animation and visual display of graph elements are being done via the visualization library *D3.js* [Bostock et al., 2012]. Generally speaking, it is based on the assumption that instead of separate independent graphical operations, D3 takes arrays of objects and manipulates (svg-) objects based on this set of provided data-points. Through procedure, the visualization of an array of vertices and edges becomes much more intuitive as well as computationally feasible.

In Figure 6, you see a running simulation visualization as it is being projected on the drawn network graph.

The graph-drawing utilizes D3's dynamic visualization procedure to create a force-directed layout of the theoretical network description. Throughout D3's iteration, fixed positional and directional values are being applied, making the drawing process - though being dynamic - a deterministic process, thus drawing the same graph visualization of a given graph every time it is being called upon to be drawn on the canvases 2D plane. Animations are created by transform-operations on svg-elements on the canvases.

Regarding concrete visualization of simulation-run events: Agents are drawn as circles of similar shape compared to nodes (as they are initially spawned on top of nodes of the network, thus maintaining similarity is intuitive) with borders indicating the type of (car-)agent displayed ("local" / "global") and body colors ranging in the randomly interpolated blue spectrum. Upon path-scheduling by the planner-agent, the nodes of the identified path are briefly highlighted (increased in shape) in the same color as the agent the path was identified for. Updating paths only based on new link-values without rerouting does not prompt a visual feedback, while rerouting actions prompt similar highlighting as done uponm initialization, but in the respectively interpolated green color spectrum for all nodes included in the new path.

The overall setup of animations is timestamp-based, as the globally initial (smallest) timestep in the car-agents-log is identified and set as *time-0*, to then set timeouts for all other log entries based on the difference in timestamp values to this zero-time.

5 Agent Performance Analysis

Having presented MAS backend traffic simulation and frontend visualization procedures in section 3 and section 4 respectively, we now add evaluation procedure and its implementation as well as resulting performance metrics.

5.1 Evaluation Procedure

There are different types of agents (of *local* or *global* nature) with randomly engaged origin and destination nodes as well as random events influencing the state of the network. Agents move along edges and thus reduce the speed of other agents proceeding to traverse said edge.

With ouputs of the underlying simulation on carAgents-, planner- and events-log together with the graph-file, all performance metrics can be retrieved from these information:

- The planner-log allows to compute **expected travelling time** from the initiated path for an agent with the weights of the edges as depicted in the graph-file
- From 'spawn'- and 'reach'-events in the carAgents-log, the **actual travelling time** can be computed. Note that the actual travelling time may never be larger than the expected travelling time, as events in the network never speed-up edge traversal and traversal time gets impacted only negatively by other car-agents.
- Subtracting expected from actual travelling time yields the agent's **travelling time discrapency**, which depicts one of the major performance number
- Also note that the carAgent-log's 'spawn'-event holds information regarding the **type of car-agent** spawned. Travelling time information is persisted and compared with regards to the respective car-agent type as well.
- The number of reroutes for car-agents of type 'global' shall also be considered.

The above depicted measures are calculated through a JavaScript-object being initialized upon start of the simulation. Though this process is not necessarily simulation-dependent, having all information ready and extracted for the purpose of simulating it makes access and reorganization of said information easier. The performance summary is not persisted to storage but rather displayed in the browser console window for simplicity reasons, but a proper storing of the summary to a json-file would certainly be a future enhancement.

5.2 Performance Results

Beginning the project implementation, our hypothesis was that global car-agents - having access to information regarding all exact current edge-weights - would perform superior to local car-agents as they rely on solely local intrinsic information that does not capture the global network state. We did however expect that even local car-agents perform comparably well (discrepancies of not-too-high double-digit percentages), depending on the number of spawned travelling agents and intensity of network events.

The primary test-simulation revolves around strictly local and strictly global car-agents. A total of 50 car-agents are spawned (18 of type global, 32 of type local) and a number of network events occur regularly every few (approx. 2) seconds.
The overall average travelling time discrepancy between expected and actual travelling time was 0.62748 seconds with an average travelling time of 5.43866 seconds. In comparison, the average travelling time discrepancy for local car-agents was 0.86022 seconds, while for global car-agents average discrepancy was only 0.21372 seconds. Local car-agents had discrepancies of 0.025 - 6.612 seconds, with 8 out of 32 local car-agents arriving more than a full seconds later than expected and only 2 more arriving more than 0.1 seconds late. Global car-agents had a discrepancy of 0.017 - 2.649 seconds, with 1 out of 18 global car-agents arriving more than a full seconds later than expected and 1 more arriving more than 0.1 seconds late. No rerouting occurred for the rerouting-capable global car-agents.

With 1/9 global car-agents arriving more than 100 ms late, global car-agent performance was said to be excellent. The single outlier global car-agent with high discrepancy in travelling time was to heavy changes in the network and unfortunate positioning of the agent; this is however a very possible situation in real-world traffic and as such still a nice finding in the simulation.
With 1/4 local car-agents arriving more than 100 ms late, performance of local car-agents is certainly worse than in case of global car-agents. They still (subjectively) traverse the network "well-enough", as they frequently arrive close or precisely on-time as well.
No possible rerouting of global car-agents is an indication of mostly-consistent network state, which was an intentional decision as more complex influences to the network would not allow a quite feasible visualization. For performance analysis, this statistic is not necessarily mandatory and the initial hypothesis is still inspected and validated nicely.

6 Conclusion

The setup of multi-agent system traffic simulation is feasible from a coordination and implementation standpoint, where the multi-agent characteristic well approximates the simulation behavior. The decision of logging agent-like information rather than a timestep itterative approach made the subsequent visualization and performance calculation more straight-forward to implement and utilizing results in the car-agent's context was less of an abstract sequence-like implementation.

The interaction of different technology - Python backend and JavaScript frontend with intermediate json and custom log-file format - was mostly seamless and notably well separated, such that functionality had not to be collaboratively shared but rather lose specifications regarding information- (not implementation-) format needed to be adhered to.

We showed that car-agent traversing a network of nodes and links perform considerably better when utilizing global network information as compared to local static graph information. The impact of global network-wide events has impact on both groups of agents, where global agents are - whilst being more flexible and resistant in general - still vulnerable to unfortunate circumstances.

The decision to exclude public transport and sharing approaches like car-sharing, whilst being unfortunate regarding an evaluation standpoint, was directly motivated by complexity constraints regarding a unified architecture. This well resembles specifications of available framework solutions and with the here depicted setup focusing on a novel multi-agent system implementation, running car-agent simulations ensured enough of complexity to allow for evaluation whilst also making the architecture more intuitive to connect and utilize.

Future work may include dedicated performance output formating, a more refined in-depth visualization procedure that allows focusing on specific vehicles at a given point in time and inclusion of timestep-based simulation to incorporate schedules and timetables for sharing and public transport agents.

References

[Barabási, 1999] Barabási, Albert-László, R. A. (1999). Emergence of scaling in random networks. *science*, 286.5439:509–512.

[Bostock et al., 2012] Bostock, M. et al. (2012). D3. js. *Data Driven Documents*, 492:701.

[Brakewood and Watkins, 2018] Brakewood, C. and Watkins, K. (2018). A literature review of the passenger benefits of real-time transit information. *Transport Reviews*, pages 1–30.

[Ksontini et al., 2016] Ksontini, F., Zargayouna, M., Scemama, G., and Leroy, B. (2016). Building a realistic data environment for multiagent mobility simulation. In *Agent and Multi-Agent Systems: Technology and Applications*, pages 57–67. Springer.

[Mastio et al., 2015] Mastio, M., Zargayouna, M., and Rana, O. (2015). Towards a distributed multiagent travel simulation. In *Agent and Multi-Agent Systems: Technologies and Applications*, pages 15–25. Springer.

[Sezen, 2003] Sezen, B. (2003). Modeling automated guided vehicle systems in material handling.

[Zargayouna et al., 2013] Zargayouna, M., Zeddini, B., Scemama, G., and Othman, A. (2013). Agent-based simulator for travelers multimodal mobility. In *KES-AMSTA*, pages 81–90. Citeseer.