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

Konstantin Kloster[1] and Oliver Berg[1]

[1]Technical University Kaiserslautern

October 31, 2018

Abstract

TODO write abstract

Contents

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; like shown in Equation 1 - being a slightly modified version of the formulas used in [Mastio et al., 2015] and [Ksontini et al., 2016].

$$t = \begin{cases} s, & \text{if } n < n_{\text{thresh}} \\ s * \alpha(n - n_{\text{thresh}}), & \text{otherwise} \end{cases} \tag{1}$$

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 Equation 1 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

TODO write backend-section

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 1: 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 *linkEdgeFrom*, *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 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

Plain HTML, CSS, JavaScript whilst utilizing D3 for visus (more on that in subsection 4.4).

- Navigation flow, Selection, Disables, Displayed Information

4.4 Web-based Animations with D3

- visualization library *D3.js* [Bostock et al., 2012] - visu graph - visu array of objects - visu animations

5 Agent Performance Analysis

TODO write performanceAnalysis-section

6 Conclusion

TODO write conclusion-section

References

[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.