Load Generators for Automatic Simulation of Urban Fleets

Pasqual Martí[1], Jaume Jordán[1][0000−0003−0400−9136], Javier Palanca[1][0000−0002−6209−9603], and Vicente Julian[1][0000−0002−2743−6037]

Valencian Research Institute for Artificial Intelligence (VRAIN), Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain.
pasmargi@inf.upv.es,{jjordan,jpalanca,vinglada}@dsic.upv.es
http://vrain.upv.es/

Abstract. To ensure cities sustainability, we must deal with, among other challenges, traffic congestion, and its associated carbon emissions. We can approach such a problem from two perspectives: the transition to electric vehicles, which implies the need for charging station infrastructure, and the optimization of traffic flow. However, cities are complex systems, so it is helpful to test changes on them in controlled environments like the ones provided by simulators. In our work, we use SimFleet, an agent-based fleet simulator. Nevertheless, SimFleet does not provide tools for easily setting up big experiments, neither to simulate the realistic movement of its agents inside a city. Aiming to solve that, we enhanced SimFleet introducing two fully configurable generators that automatize the creation of experiments. First, the charging stations generator, which allocates a given amount of charging stations following a certain distribution, enabling to simulate how transports would charge and compare distributions. Second, the load generator, which populates the experiment with a given number of agents of a given type, introducing them dynamically in the simulation, and assigns them a movement that can be either random or based on real city data. The generators proved to be useful for comparing different distributions of charging stations as well as different agent behaviors over the same complex setup.

Keywords: multi-agent system · simulation · transportation · electric vehicle · smart city · urban fleets

1 Introduction

With more than half of the world's population living in cities, the list of challenges for keeping them sustainable has grown. "A smart sustainable city is an innovative city that uses ICTs (Information and Communication Technologies) to improve quality of life, the efficiency of urban operations and services and competitiveness while ensuring that it meets the needs of present and future generations concerning economic, social, environmental and cultural aspects"[1]. In

[1] This definition was provided by the International Telecommunication Union (ITU) and United Nations Economic Commission for Europe (UNECE) in 2015

this paper, we focus on one of these challenges, traffic congestion (and its associated carbon emissions), from two different approaches. The first is the transition to electric vehicles (EV), which would greatly reduce the carbon emissions generated by cities. The main problem that users encounter to switch to EV is the insecurity they feel towards access to EV charging stations [6,2]. The allocation and installation of EV charging stations inside a city is not a trivial problem and it entails a great disbursement of money for municipalities. As for the second approach, it is based on traffic flow optimization. Improving this requires accurate data. One way of researching how to deal with such challenges is through the use of simulators [4]. With them, we can see the effect some of the changes would have over the city after defining them. However, cities are very complex systems, so it is necessary to have a complete simulator that allows experimentation with big, complex configurations for both charging station allocation and traffic inside the city. The more realistic the simulator, the more accurate and useful experiments would be for real-world applications.

In this work, we use SimFleet simulator [5], which is able to place different varieties of agents with custom behaviors over real-world cities to develop and test any type of strategies. Nevertheless, there is an important flaw in SimFleet to prepare simulations with a significant number of agents, and also to have an appropriate representation of the traffic of a city. Therefore, in this paper, we propose a significant improvement for SimFleet: the inclusion of two generators at different levels. On the one hand, a charging stations generator to create several distributions of these infrastructures, and to be able to make comparisons and simulations with well-informed charging stations emplacing systems such as the one in [3], which uses several data sources to feed a genetic algorithm that obtains solutions. On the other hand, a load generator of agents on the move in a city such as urban fleets of taxis, private vehicles, delivery transports, buses, etc.; and customers of taxis or packages. Furthermore, this load generator can consider real data of the city, which implies a more informed approach to generate the real traffic of a city to be used in dynamic simulations.

These generators will allow SimFleet users to create realistic scenarios easily without having to write long configuration files, and more importantly, to generate load representing real traffic of the specific city by using available data such as population, traffic, and tweets, from open data portals[2], or gathered with other tools such as [7].

The rest of this paper is structured as follows. Section 2 presents the fleet simulator SimFleet, in which this work is built upon, and its main flaw. Then, Section 3 explains the two main generators proposed in this work, that is, the charging stations simulator and the load generator of urban fleets. Section 4 presents some experimental results using the generators in SimFleet. Finally, the conclusions of this work are presented in Section 5.

[2] http://gobiernoabierto.valencia.es/en/data/

2 SimFleet Description

SimFleet [5] is an agent-based fleet simulator to test strategies. Each simulation counts with a series of customers, transports and a fleet manager. Customer agents represent people (or packages) that need to be transported from their origin location to their destination in the city. For doing so, each Customer agent requests a single transport service, provided by a Transport agent. Finally, the FleetManager agent is responsible for putting in contact the customers in need of a transport service, and the transports that may be available to offer these services. In short, the FleetManager agent acts like a transport call center or directory facilitator [1], accepting the incoming requests from customers and forwarding these requests to the appropriate transports.

For the movement of Transport agents around the city, SimFleet makes use of OSRM[3], a routing engine for finding shortest paths in road networks. Querying an OSRM server specifying the service *route* and passing origin and destination points returns the shortest route between those two points.

The behavior of each agent and the way they get to agreements is determined by its strategy. In our work, however, we set aside the aspect of implementation and testing of strategies to focus on the creation of simulations, a feature in which SimFleet has weaknesses.

2.1 The SimFleet flaw

To understand the weaknesses SimFleet presents in relation to simulation creation we must first explain how the experiments are described. SimFleet uses a configuration file in JSON format to load the agents of a simulation. The agents and their features are the relevant part of the configuration for our line of work. There are four types of agents: fleets, transports, customers and stations. Each agent, depending on its type, has a certain amount of attributes that must be indicated for their creation.

Currently, the only way of filling the configuration file is to manually create each agent, giving values to their attributes. This is inconvenient to create scenarios with a great number of vehicles, customers or packages. Besides that, it is likely that users employ SimFleet for reproducing the mobility around a city in a simulated environment. This task involves the dynamic input of agents in the simulation as well as their informed movement around the environment based on real data from the city. Additionally, the introduction of randomly generated agents that interfere with the main simulated task could be useful for building a more realistic trial as well as testing different distributions of charging stations or simply evaluating new strategies. Our work aims to fulfill those needs by the introduction of Generators that automatize the creation of simulation configurations.

[3] `http://project-osrm.org/`

3 Generators

We will now introduce our work, which consists of generators to facilitate the setup of bigger and more realistic simulations with SimFleet. It includes a charging stations generator, which populates the simulation area with a given number of charging stations following a determined pattern; and a load generator, which populates the simulation space with different types of agents with an associated movement that can be pseudo-random or informed (provided we have access to real-world data). Besides that, fully random versions of both generators were also implemented so as to compare informed versions against them.

A simulation on SimFleet is defined by its configuration file. To automatically create experiments means to generate new configuration files or fill a previous one with data according to some parameters. For this, all generators include the option to get as input a configuration file and they are prepared to leave the present objects of such file unchanged while including the ones generated. Besides that, each generator works with a GeoJSON file that defines the area of the real world where our simulation will take place and thus they have to populate. We will call this file the **city map**, since, usually, simulations are performed within the borders of a city. Since it represents a real-world location, all geometries defined in the city map are indicated with latitude-longitude points; to manipulate them we will use the Python Shapely library[4].

3.1 Charging stations generator

The charging stations generator is used in order to test different distributions of charging (or petrol) stations of any kind over an area. The generator has many parameters; we will only present the relevant ones: n charging stations to distribute; p charging poles of the distribution; and distribution type, $\{random, uniform, radial\}$, that affects how stations are positioned inside the area.

Each station may have several charging poles. The allocation of charging poles between stations will be discussed further on. The generator outputs a file in GeoJSON format which indicates the type and position of each station. Such a position, however, can not be any point inside the city map but rather a valid point: a point belonging to a street or road. For obtaining valid positions, we make use of the function *getValidPoint*, which given a point returns the nearest valid point to it. This function uses the service *nearest* of OSRM, which returns the nearest point belonging to a street or road with respect to the specified coordinates.

Random distribution. In this type of distribution, n valid points are generated within the city map. Each of the points indicates the position of a charging station. For the generation of points, the bounds of the polygon that defines the city map are used: x_{min}, y_{min}, x_{max}, y_{max}. Using these values, random x and y coordinates are generated for each point. Then, the corresponding valid point

[4] `https://pypi.org/project/Shapely/`

is obtained. If the point is contained in the city map, it is stored as a station location; otherwise, it is discarded. This process is repeated until there are as many stored points as the number of stations.

A random distribution is interesting from the experimentation perspective to compare other more informed distributions against it.

Uniform distribution. This distribution divides uniformly[5] the city map (see Figure 1a) into equal size cells, generally with rectangular shape. To do so, it creates a wider working area, the *grid* (Figure 1b), defined from the bounds of the polygon representing the city map. If such bounds are x_{min}, y_{min}, x_{max}, y_{max}, the grid is the four vertex polygon defined by the points $\{(x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{max}), (x_{max}, y_{min})\}$.

The number of rows and columns of the grid is determined by the number of stations (n) and its width and height, following the criteria shown in Equation (1). If the number of stations is a perfect square, i.e., its square root is a positive integer, the grid will have the same amount of rows and columns, obtaining exactly n cells. However, in general, the number of rows and columns will depend on whether the grid is wider or higher, having one more column than the number of rows (or vice versa) accordingly.

$$\begin{cases} rows = cols = \sqrt{n} & n \text{ is perfect square} \\ rows = \lfloor\sqrt{n}\rfloor,\ cols = \lceil\sqrt{n}\rceil & height < width \\ rows = \lceil\sqrt{n}\rceil,\ cols = \lfloor\sqrt{n}\rfloor & otherwise \end{cases} \tag{1}$$

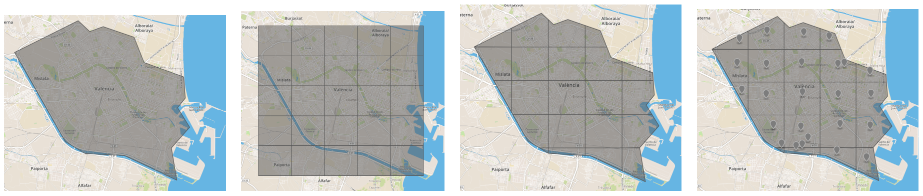
Once the grid is split, we trim each of the cells against the city map, which causes cells outside of it to disappear and those laying over its borders to get an irregular shape (see Figure 1c). These final cells are stored in a list of valid polygons where stations can be placed.

Next, an iterative process begins to allocate every station. The list of valid polygons is traversed and a station is placed in the closest valid point (using the *getValidPoint* function) to the centroid of the polygon. This process finishes once every polygon has one station in it or all stations have been allocated. After this, if there were still stations to allocate, the rest are positioned in a random valid point of a randomly chosen valid polygon (Figure 1d shows a possible outcome).

Besides this, we also implemented a random version of this distribution in which the city map is divided in the same way emplacing, however, each station in a random valid point within a randomly selected cell.

Radial distribution. This distribution was inspired by the radial distribution of activity within certain cities, which presents a higher rate towards its core and decreases as it gets closer to the peripheries. It requires an additional parameter c, which indicates the number of circles that will be used to divide the city map.

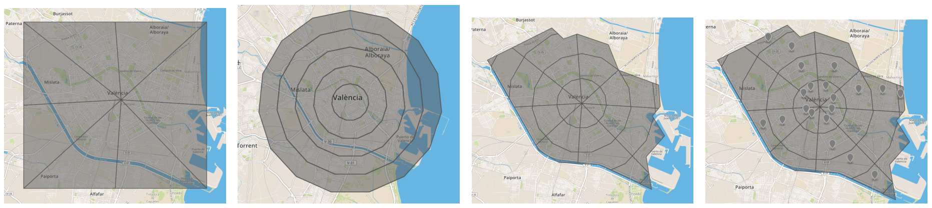
[5] The name “Uniform distribution” does not refer to a probability distribution but to how stations are divided in the city map.



(a) City map (b) Working area or Grid (c) Grid trimmed against city map (d) Cells populated with stations

Fig. 1. Uniform distribution of stations process

The division process begins by defining two copies of a wider working area, created as described for the uniform distribution. One of those copies will be divided into a series of triangles, 8 by default as can be seen in Figure 2a, by joining every vertex and sides' middle point with the centroid of the original city map. As for the other, it will be divided by c concentric circles with an initial radius r calculated taking into account the map dimensions. To avoid circle overlap, each circle is trimmed against the one created before it, starting by the last (and biggest) created. This obtains an area with a middle circle and many rings around it, as can be seen in Figure 2b. Take into account that from now on when we mention circles we will also be referring to the rings. Next, the two aforementioned areas are intersected with each other, dividing each circle into 8 polygons, and trimmed against the city map, obtaining a division of it as shown in Figure 2c.



(a) Triangle division of working area (b) 5 circle division of working area (c) Final city map division (d) 20 stations between 5 circles

Fig. 2. Radial city map division

Each station is then allocated in the closest valid point to the centroid of one of the polygons. The number of stations per circle (n/c), as well as the amount of polygons a circle has, is taken into account to allocate the stations as uniformly as possible within the city map and each circle. The algorithm populates each triangle starting from the inner circle and moving towards the outer. Once all polygons of a triangle have a station assigned, the algorithm will select the next triangle according to the number of stations and the total number of polygons to divide the stations uniformly within a circle. The final distribution can be seen in Figure 2d.

The number of stations may be greater than the number of polygons since the number of circles is determined by the user. The algorithm described above places only one station in each polygon. In this case, the rest of the stations are assigned a random position by randomly choosing a polygon and a valid point within it.

We also implemented a fully random version of this distribution that divides the city map in the same way, takes into account the number of stations per circle, but chooses randomly the polygons within a circle as well as a valid point within them.

Charging poles allocation. The amount of charging poles (spots for a vehicle to charge) we want in our configuration is one of the parameters of the station generator. There must be at least one charging pole in each station. After this, if there are more charging poles to be distributed they will be allocated according to one of the following methods:

The first one distributes the points evenly by traversing the list of stations, adding one point to each until all points have been allocated. Then, the list of stations is shuffled, so as not to benefit stations that are traversed first. In this way, the stations will have either p/n or $p/n - 1$ charging poles.

The alternative is a pseudo-random distribution of the remaining points that works choosing a random amount of points and a random station to which assign them. To avoid a too uneven distribution of poles, such a random amount can be limited by a parameter that indicates the maximum percentage of the total charging poles that a single station can have. For instance, using a maximum percentage of 30%, we ensure that no station will have more than $0.3 \cdot p$ charging poles.

3.2 Load generator of movements in a city

The load generator is used to create either a random or informed load on the simulation. Such load can be adapted to any of the agent types that SimFleet offers: electric vehicles, taxi fleets, customers for taxis, delivery vehicles, packages, etc. The relevant parameters of this generator are: agent type t; amount of agents n; minimum distance min_dist in meters; starting delay d in seconds; amount of agents per batch $agents\_per\_batch$.

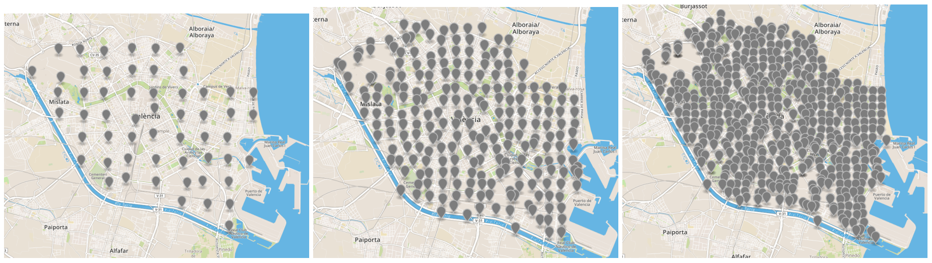
The generator aims to create a movement of at least min_dist meters of n agents of t type within the borders of a given city map. The delay parameter d determines at what time of the simulation the generated agents will start running; by default, it is 0. The amount of agents per batch is introduced to give different delays to sets of $agents\_per\_batch$ agents, which will begin its execution at the same time. This may be useful when generating a large number of agents. If indicated, the first batch of agents will have a delay of d; the second, a delay of $2d$, and so on. As we mentioned above, all generators are prepared to receive an existing SimFleet configuration file as input and fill it with agent data. This enables the use of the load generator to introduce, in the same simulation, different types of agents in various amounts, with different delays and batch sizes, to create a complex system.

Random movement generator. The random load is created by choosing a random route (random origin and destination points) for the agent to perform. Both random origin and destination points must be valid points of the area, and they must be at least *min_dist* apart from one another. This process is repeated to create n agents of type t. The origin point will determine where in the area the agent will spawn, whereas the destination point indicates where it will finish its execution. If the agent type is customer or package, the movement is performed by the corresponding transport vehicle that carries it after it gets picked up.

Informed movement generator. The informed version of the load generator aims to reproduce more realistic movements around the city map. For this, it is necessary to provide the generator with relevant data from which to ground the routes of the agents. This data can be obtained from diverse sources; often open data platforms that the government of a city or country makes accessible for its citizens. For our generator, we used the following data:

- **Population information**: It shows the amount of people that live in different zones of a city. The population information (P) is defined as: $P = \{(C_1, p_1), (C_2, p_2), \ldots, (C_n, p_n)\}$, where C_i is a closed polygon representing a zone in the city together with its population p_i.
- **Traffic information**: It shows the number of vehicles moving around a certain area. The traffic information (T) is defined as: $T = \{(R_1, t_1), (R_2, t_2), \ldots, (R_n, t_n)\}$, where R_i is a polyline that follows a street or road indicating the volume of traffic t_i.
- **Twitter activity**: Information about the amount of geo-located tweets, from the social network Twitter, tweeted from a certain location. This information can be used to determine where a representative percentage of the population is spending their time. The Twitter activity (A) is defined as: $A = \{(Q_1, a_1), (Q_2, a_2), \ldots, (Q_n, a_n)\}$, where Q_i is a point represented as a latitude-longitude tuple and a_i the number of tweets in such coordinates.

The information is used to create a probability distribution among the available points of the area. The selection of the origin and destination points will be performed according to such distribution. For this, we begin by creating a set of available points. The city map (M) is divided as if it was a grid similarly as explained in Section 3.1 for the uniform distribution, obtaining $M = \{(G_1, O_1), (G_2, O_2), \ldots, (G_n, O_n)\}$; where G_i is a closed polygon and O_i the nearest valid point to the centroid of G_i. The number of rows and columns of the grid is a configurable parameter and it determines the granularity of the system. A higher amount implies more cells in the grid which directly translates into more available points, as it can be seen in Figure 3. The more points, the more distributed will be the probability.



(a) 10 rows and cols (b) 20 rows and cols (c) 30 rows and cols

Fig. 3. Number of available points according to map division granularity

By merging the city data with M, we join, for every polygon G_i, the population, traffic and Twitter activity amounts that take place within its area: $M = \{(G_1, O_1, \{p_1, t_1, a_1\}), (G_2, O_2, \{p_2, t_2, a_2\}), \ldots, (G_n, O_n, \{p_n, t_n, a_n\})\}$ and calculate the probability associated to each point O_i as in Equation (2):

$$prob(O_i) = w_p \cdot \frac{p_i}{\sum_{j=1}^{N} p_j} + w_t \cdot \frac{t_i}{\sum_{j=1}^{N} t_j} + w_a \cdot \frac{a_i}{\sum_{j=1}^{N} a_j}; \text{ with } w_p + w_t + w_a = 1 \quad (2)$$

where w_p, w_t and w_a are weights that control the influence of each of the factors over the probability. Finally, the generator takes into account the set of available points (S) and their corresponding probability: $S = \{(O_1, p(O_1)), (O_2, p(O_2)), \ldots, (O_n, p(O_n))\}$ to generate the routes.

Once every point in S has its probability assigned, a process to create the n routes begins (see examples of Figure 4). This process is very similar to the one used in the random load generator, but this time the origin and destination points are chosen from S according to the probability distribution and ensuring the *min_dist* between both points.

4 Experimental Results

To present experimental results for our generators, we define and set up a simulation and then execute it in SimFleet with two different strategies.

4.1 Setup

The experiment defined takes place over the main area of the city of Valencia, Spain. We generated 20 electric charging stations distributed uniformly in the

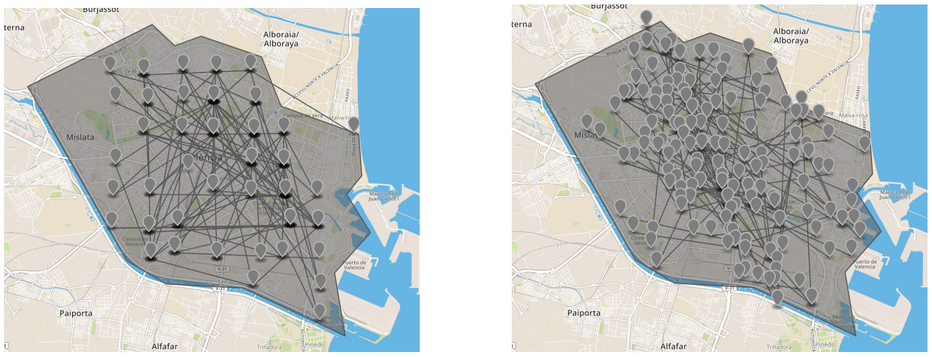

(a) 10x10 map (b) 30x30 map

Fig. 4. 100 routes examples

area. As for the load, we used the informed load generator to create a fleet of 30 electric taxis and 30 customers with a granularity of 30. Taxis were created with random values for their autonomy so that some of the taxis would be forced to charge in one of the stations. For the customers, the weights of Equation (2) were $w_p = 5/12$, $w_t = 1/4$, $w_a = 1/3$, increasing the impact of population and Twitter activity. The routes of the customers were defined by points that were at least 700 meters apart. For the taxis creation, the weights of Equation (2) were $w_p = 1/3$, $w_t = 5/12$, $w_a = 1/4$, giving more importance to traffic and population. Figure 5 shows the described experiment in SimFleet.

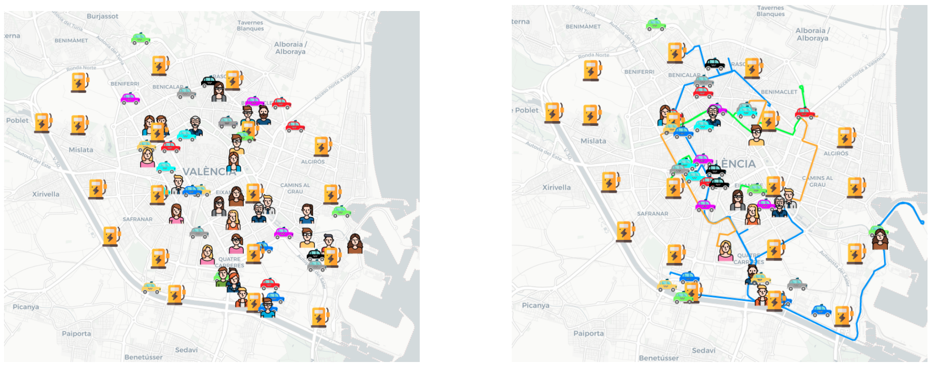

(a) Experiment setup (b) Experiment running

Fig. 5. Experiment shown in SimFleet

We executed this setup using two different strategies for the FleetManager and Transport agents. The first strategy is the *default* one provided by SimFleet:

All customers send their transport service requests to the FleetManager which, in turn, forwards them to every taxi in the fleet, ignoring their state. Taxi agents, if available, will accept the request of a random customer, taking into account that it may have already another taxi assigned. As for charging, taxis choose a random available station. In contrast, the *modified*, more informed strategy works in the following way: Customers still send their requests to the FleetManager but the latter will just inform the nearest taxi to the customer among the available taxis. Taxis will then attend requests of the customers closer to them, which will reduce the customers' waiting time. Also, if the taxi needed to charge, it would not go to a random station but the closest one to it.

4.2 Results

The metrics that SimFleet offers to evaluate simulations are the average waiting time of the customers until their assigned transport picks them up, the average total time the customer waits for their transport service to be completed, and the simulation time. Since the modified strategy aims to reduce customer waiting time and total time, we will use them as the metrics to improve.

Strategy	Avg. waiting time	Avg. total time	Avg. simulation time
Default	21.06	25.28	50.38
Modified	**11.50**	15.72	43.14
Improvement	45.39%	37.82%	14.37%

Table 1. Time (seconds) for the tested strategies

The results presented in Table 1 are an average of 5 different executions, since the agent behavior is not deterministic for neither of the strategies. As can be seen, the modified strategy achieves an average waiting time of 11.50, which implies an improvement of 45.39% over the time achieved by the default strategy. The generators achieve their objective of facilitating the configuration of experiments and the comparison of agent strategies in different simulations. Some video demonstrations of the generator setups in SimFleet are available online: strategy comparison[6], delay start and batches[7], and ecar fleet[8].

5 Conclusions

In this work, we have identified the need for enhancing SimFleet simulation potential and presented two tools to do so: the charging stations generator and the load generator. Knowing that one of SimFleet's purposes is the implementation

[6] `https://viewsync.net/watch?v=XNRLQTUlL-Y&t=0&v=QesxSMdEFLI&t=0`
[7] `https://youtu.be/9OgUOVmz4co`
[8] `https://youtu.be/EokHXBAzlL4`

and comparison of agent strategies, the generators are effective to help in the research of solutions for traffic congestion or any other type of challenge derived from city sustainability. Provided we have access to city data, with the use of the informed load generator, we can test different driver behaviors over realistic settings to identify problem sources and look for appropriate solutions.

As for the EV charging stations infrastructure, thanks to both generators we can simulate different distributions over a city and, using its real city data, recreate movement within it to analyze the performance of each distribution. This information can be of use for municipalities or other entities in charge of infrastructure creation.

The placement of a charging station should take into account current traffic trends but one must keep in mind that it may as well have an impact on traffic once the station is working. In future work, we aim to develop coordination strategies among transport and station agents in order to find ways of optimizing traffic and achieving a maximum global utility. Such strategies may be the basis for autonomous EV in the smart cities of the future.

Acknowledgments

This work was partially supported by MINECO/FEDER RTI2018-095390-B-C31 project of the Spanish government. Pasqual Martí and Jaume Jordán are funded by UPV PAID-06-18 project. Jaume Jordán is also funded by grant APOSTD/2018/010 of Generalitat Valenciana - Fondo Social Europeo.

References

1. Campo, C.: Directory facilitator and service discovery agent. FIPA Document Repository (2002)
2. Dong, J., Liu, C., Lin, Z.: Charging infrastructure planning for promoting battery electric vehicles: An activity-based approach using multiday travel data. Transportation Research Part C: Emerging Technologies **38**, 44–55 (2014)
3. Jordán, J., Palanca, J., Del Val, E., Julian, V., Botti, V.: A multi-agent system for the dynamic emplacement of electric vehicle charging stations. Applied Sciences **8**(2), 313 (2018)
4. Noori, H.: Realistic urban traffic simulation as vehicular ad-hoc network (vanet) via veins framework. In: 2012 12th Conference of Open Innovations Association (FRUCT). pp. 1–7. IEEE (2012)
5. Palanca, J., Terrasa, A., Carrascosa, C., Julián, V.: Simfleet: A new transport fleet simulator based on mas. In: International Conference on Practical Applications of Agents and Multi-Agent Systems. pp. 257–264. Springer (2019)
6. Skippon, S., Garwood, M.: Responses to battery electric vehicles: Uk consumer attitudes and attributions of symbolic meaning following direct experience to reduce psychological distance. Transportation Research Part D: Transport and Environment **16**(7), 525–531 (2011)
7. del Val, E., Palanca, J., Rebollo, M.: U-tool: A urban-toolkit for enhancing city maps through citizens' activity. In: International Conference on Practical Applications of Agents and Multi-Agent Systems. pp. 243–246. Springer (2016)