Charging Stations and Mobility Data Generators for Agent-based Simulations

Pasqual Martí[a], Jaume Jordán[a], Javier Palanca[a], Vicente Julian[a]

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

Abstract

Current traffic congestion and the resulting carbon emissions are two of the main problems threatening the sustainability of modern cities. The challenges facing today's cities focus primarily on the optimization of traffic flow and the transition to electric vehicles. The latter aspect implies the need for an adequate deployment of the infrastructure of charging stations. The inherent complexity in today's cities and the difficulty in implementing new policies whose benefits are difficult to measure and predict has led in recent years to consider the enormous potential of simulation tools and in particular of the agent-based simulation (ABS). ABS allows the specification of complex models that reflect the complexity and dynamism of urban mobility. Current technology in ABS has evolved and matured sufficiently to provide very sophisticated tools but lacking facilities for a flexible and realistic generation of input data in the execution of the experiments. In line with this, this paper introduces two configurable generators that automatize the creation of experiments in agent-based simulations. The generators have been developed with the SimFleet simulation tool enhancing the simulation of realistic movements and location of vehicles, passengers and other users of the urban traffic system within a city. The generators proved to be useful for comparing different distributions of locations as well as different agent movement behaviors based on real city data.

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

1. Introduction

The International Telecommunication Union (ITU) and the United Nations Economic Commission for Europe (UNECE) defined, in 2015, a smart sustainable city as an innovative city that uses Information and Communication Technologies (ICTs) 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. Currently, the list of challenges for keeping current cities sustainable has grown, and consequently, so has the need to establish appropriate intervention policies with the lowest possible risk.

One way of researching how to deal with such challenges is through the use of simulators [1], and specifically agent-based simulation (ABS) [2]. ABS integrates an interesting number of properties which makes it useful for a wide range of domains, supporting structure preserving modeling of the simulated reality, parallel computations, simulation of proactive behavior, and flexible and dynamic simulation scenarios [3]. All these properties can be clearly observed in the domain of today's cities, where we find multiple autonomous entities that move around the city and make use of a collection of resources located according to certain policies, such as policies for the deployment of new electric vehicle charging stations. Currently, we can find a multitude of agent-based simulation tools [4], some of them specifically designed for traffic or urban mobility management. Using these tools, it is possible to see the effect that the changes would have on the city after defining them, thus avoiding a possible unsuccessful deployment. However, as current cities are very complex systems, it is necessary to have a complete simulator that allows experimentation with big and complex configurations inside the city. The more realistic the simulator, the more accurate and useful experiments would be for real-world applications.

In this work, we propose different ways to generate more realistic data as input for agent-based simulation experimentation, extending a previous work presented in [5]. More specifically, we focus the processes of generating more realistic data on two problems: the generation of possible locations for electric vehicle charging stations, and the generation of realistic movements of entities in a city to have an appropriate representation of its traffic flow. To do this, we use SimFleet simulator [6], which is able to place different varieties of agents with custom behaviors over real-world cities to develop and test any type of strategies. Over SimFleet we have developed two generators for the

above commented problems. On the one hand, a charging stations generator to create several distributions of these infrastructures, and make comparisons and simulations with well-informed charging stations emplacing systems such as the one in [7], which uses several data sources to feed a genetic algorithm that obtains solutions. On the other hand, a mobility data generator of entities that move in a city such as delivery transports, private vehicles, or taxi fleets, among others. Moreover, these mobility data generators makes use of real data of the city, which implies a more informed approach to the generation of realistic traffic in a city to be used in dynamic simulations. To do this, different generators of each type have been developed, where the most sophisticated ones are based on different AI techniques and are explained in detail further in this paper.

In order to illustrate the use of the proposed generators, the paper includes a case study in the city of New York, concretely in the Manhattan Island area. The study has been made using available data such as population, traffic, and tweets, from open data portals, or gathered with other tools like U-tool [8].

The rest of this paper is structured as follows. Section 3 presents the SimFleet simulator, on which the proposal presented in this paper has been developed. Sections 4 and 5 explain the two main generators proposed in this work, that is, the charging stations simulator and the mobility data generator. Section 6 illustrates through a case study the use of the proposed generators. Finally, some conclusions are presented in Section 7.

2. Related work

Agent-based simulation has become in recent years a key aspect for the development of more realistic simulations with high scalability. In the environment of urban mobility, there are many works that try to perform simulations to study aspects such as traffic, movement of citizens, crowds, emergency situations, or the optimal location of different services.

To support the modeling and development of these simulations, different tools have been appearing that facilitate the execution of experiments for the study of mobility both at urban and interurban level. A review on agent-based simulation tools for traffic and transportation can be consulted in [9].

In traffic simulation, one of the most well-known tools is SUMO [10]. SUMO is an open-source traffic simulation framework which includes net import and demand modeling components. SUMO helps in several research top-

ics such as traffic light algorithm, the choice of routes, and in the simulation of vehicular communication with other vehicles or with the infrastructure. The framework is used in different projects to simulate traffic management or autonomous driving. SUMO employs origin/destination matrices to describe the movement between traffic assignment zones in vehicle number per time in large-scale scenarios. Moreover, SUMO can be extended through new applications in order to extend how to generate traffic information for the simulation process.

Another well-known simulation framework is MATSim [11]. MATSim is a framework that allows the implementation of large-scale agent-based transport simulations. The framework is mainly employed for demand-modeling and traffic flow simulation. MATSim offers several extensions which enhance the functionality with additional features, one example is the package that allows to convert Google Transit Feed Specification (GTFS) data into a MATSim transit schedule. The GTFS is an extension of the General Transit Feed Specification which is a data specification that allows public transit agencies to publish their transit data in a format that can be consumed by a wide variety of software applications. Other example we can highlight is SIMmobility [12], which is a simulation framework that helps in the prediction of the impact of mobility demands on intelligent transportation services, vehicular emissions and transportation networks, or its specific version for logistics called SimMobility Freight [13]. VISSIM [14] is another well-known commercial simulator that provides an ecosystem of products that can be integrated to provide solutions to solve different mobility and transportation challenges. VISSIM is the only one that offers real-time knowledge acquisition. Lastly, Matisse 3.0 [15] is the last version of a microscopic simulator for agent-based intelligent transportation systems which includes intersection controllers and enables V2X and I2I [16] communication mechanisms.

Another widely used framework is AnyLogic[1]. AnyLogic is a general purpose simulation software but includes specific extensions for mobility management that allows to simulate aspects such as transportation planning, fleet management, and traffic flow. As facilities for data generation, AnyLogic allows the import of databases as well as the generation from scratch of different components of the simulation. There exists numerous examples of mobility simulation models in AnyLogic, in [17] a simulation model is used

[1]https://www.anylogic.com

as a decision support tool for estimating efficiency of vehicle schedules with time windows. Another example is the work presented [18] which is a study of passenger flow in urban subway stations which makes use of the Anylogic pedestrian library.

From another perspective, new agent-based simulators for the generation and testing of autonomous driving strategies have recently appeared. These simulators are based more on providing mechanisms for sensing, monitoring, communication and action at the level of autonomous vehicles in order to provide solutions for the problem of autonomous driving. One of the best-known examples is CARLA[2] [19]. CARLA allows controlling aspects such as traffic generation, pedestrian behaviors, weathers and vehicle sensors. Its main goal is to allow the learning of new driving policies or the training of new perception algorithms. Other similar simulation frameworks are AIRSim[3] [20] (a simulator developed by Microsoft as a platform to experiment with AI learning algorithms), TORCS[4] [21] (an Open Racing Car Simulator which is an open source 3D car racing simulator that is designed to enable AI-based strategies for competing drivers), and in [22] an urban traffic simulation framework is presented for helping the development and test of automated driving vehicles.

As can be observed there exist different agent-based simulation tools that offer several facilities for the generation of highly accurate simulations with high scalability. However, most of these tools do not offer specific facilities for a flexible and realistic generation of input data in the simulation. Usually, the analyzed approaches incorporate the possibility of generating third-party software or simply include database import modules. Accordingly, in our proposal we make use of different approaches, including AI techniques, that make use of real-world data as input to improve simulations both by positioning elements, such as electric charging stations, in a more informed way, and by generating new, more realistic input data, such as the most feasible traffic routes.

3. Extending the SimFleet simulator

SimFleet [6] is a simulator based on SPADE [23] (a multi-agent system

[2]http://carla.org

[3]https://microsoft.github.io/AirSim/

[4]https://sourceforge.net/projects/torcs/

development environment) specialized in testing different mobility strategies where vehicles that belong to different fleets interact in the simulation. This simulation tool has been chosen because of its features. It allows you to manage simulated fleets in an easy and very flexible way. This is thanks to the agent architecture provided by the SPADE platform, which allows every actor in the simulation to be a proactive and independent agent which can have its own strategy and behavior. Also, scaling the simulation is simple. In SimFleet each simulation counts for a number of clients (Customer agents), transport operators (Transport agents) and fleet managers (FleetManager agents), where Customer agents serve individuals who need to be shipped from their place of origin to their place of destination in the region. In order to do so, each Customer agent demands a single transport service offered by the Transport agent. Then, it is the duty of the FleetManager agent to get the clients in need of a transport service and the transport providers that might be required to provide such services into touch. In short, the FleetManager agent serves as a command and control center for transports. It acknowledges the incoming customer requests and forwards those requests to the relevant transport providers.

For passenger transportation across the region, SimFleet uses the OSRM[5] routing software to locate the shortest routes in the road network. A query to OSRM receives the origin and destination points and returns the shortest route between the two points.

A SimFleet user needs to develop the behavior of each agent in the simulation in order to define their negotiation policy. Throughout this research we have ignored the development and testing stages in order to concentrate on the simulation development, a function in which much of this sort of simulators have limitations, as presented in the related work.

We must clarify how the experiments are represented in order to understand the limitations of SimFleet in conjunction with the development of simulations. To load a simulation into SimFleet the user must write a JSON file where the details of each actor of the simulation (this is, the agents) are described (position, initial data, goals, etc.). These parameters may vary depending on the type of agent.

The only way to fill in the configuration file at the moment is to build each agent manually, providing values to their attributes. This presents a problem

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

for developing simulations with a huge amount of vehicles, consumers or packages. In addition, SimFleet is likely to be used by users to replicate static elements such as charging (or gas) stations and movements around a city in a simulated environment. On the one side, it may be of interest for urban planners to measure how each distribution can influence the mobility of the city by using a generator to put charging (or gas) stations in different configurations. Mobility modeling, on the other hand, involves introducing several agents that appear in the system during the simulation time, as well as the mobility information of agents around the area, based on real data measured from the city. Via the implementation of generators that simplify the development of simulation configurations, our work intends to solve those needs.

Two global generators are the key contribution of this work, enabling the setup of larger and more realistic simulations with SimFleet. The generators are an instrument not only for helping the user write big files, but also for creating realistic configuration files based on the actual target city details. These informed generators are designed to produce configuration files that are as similar to reality as possible (i.e. simulating vehicles mobility with real traffic data from the city).

The first generator is a generator for charging stations that populates the simulation area with a defined number of charging stations following a specified template. The second generator is a mobility data generator that fills the simulation space with various types of moving agents that can be pseudo-random or informed. In addition, in order to compare informed versions against them, entirely random versions of both generators were also introduced.

Next sections present these two generators in depth. First we present the charging stations generator, which allows selecting different approaches (from less informed to more informed) to place stations in the city map. Next, a mobility data generator is presented, which allows to create realistic movements along the city map.

4. Charging stations generator

The charging stations generator is in charge of placing a certain number of charging (or petrol) stations in the city according to a certain technique or distribution. The generator has the following main parameters: n charging stations to place; p charging poles to locate in the stations; and distribution

type, {*random*, $uniform$, *radial*, *genetic*}, that determines the technique used to place the stations in the city. The first three types of distributions correspond to non-informed charging station generators, i.e. they only use the parameters of the number of charging stations and poles to be placed on the city map according to the specified distribution (*random*, $uniform$, or *radial*). However, the so-called *genetic* distribution uses information about the city (population, traffic, and activity in social networks) to distribute the charging stations by means of a genetic algorithm that optimizes utility and cost.

The charging stations generator receives the number of charging poles as an input parameter. The charging poles are the spots that can be used by a vehicle in a charging stations, so a station consist of at least a charging pole. In the *genetic* distribution, the genetic algorithm receives as parameter the maximum charging poles per station, and hence, it allocates the charging poles p in the specified n stations depending on the utility and cost that the complete distribution provides. However, the *random*, $uniform$, and *radial* distributions allocate one pole in each station, and the remaining poles are placed using one of the following alternatives:

- In the first case, the list of stations is shuffled and the poles are distributed following the order of the list. This process is repeated shuffling again the list until all poles are distributed.
- The second case makes a pseudo-random distribution that allocates the remaining poles by selecting a random station and placing a random amount of poles in it. However, the random number of poles is limited to a percentage of the total poles to avoid uneven distributions.

The output of the charging stations generator is a GeoJSON file with the position and number of poles of each station. However, the position given by the generator is processed using the *getValidPoint* function of the service *nearest* of OSRM, which obtains a valid point situated in a street near the given coordinates. In the case of the genetic algorithm, this process is performed before to ensure that the set of Points of Interest (PoIs) that have to be provided to the genetic algorithm (it must be an amount of PoIs significantly larger than the stations to place) are already valid.

4.1. Random distribution

This distribution generates a set of n valid points in the city map that will be the positions of the charging stations. For each point, coordinates x

and y are randomly generated within the bounds of the city map defined by its polygon: x_{min}, y_{min}, x_{max}, y_{max}. The valid point of these coordinates is obtained and if it is not inside the city map, it is discard. So, until there are n valid points, this process is repeated.

Figure 1: Random distribution of stations.

Algorithm 1 shows a pseudo-code which operates as described. In addition, an example of a random distribution with 50 stations in Manhattan is shown in Figure 1. A random distribution is useful to serve as a baseline for comparisons with other more informed distributions.

Algorithm 1 Allocates n stations in random points within the *city_map*

Require: *city_map*, n
  $valid\_points \leftarrow [\,]$
  x_{min}, y_{min}, x_{max}, $y_{max} \leftarrow city\_map.bounds()$
  while $length(valid\_points) < n$ **do**
    $(x, y) \leftarrow randomPoint(x_{min}, y_{min}, x_{max}, y_{max})$
    if $city\_map.contains((x, y))$ **then**
      $valid\_points \leftarrow valid\_points \cup (x, y)$
    end if
  end while

4.2. Uniform distribution

In this distribution, the city map (Figure 2a) is divided uniformly[6] into rectangular cells of equal size. The *grid* (see Figure 2b) is a wider working are created from the bounds of the city map with the points of the polygon $\{(x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{max}), (x_{max}, y_{min})\}$.

The grid size can be obtained depending on the amount of stations (n) as specified in Equation 1. The number of rows and columns will be the square root if n is a perfect square. Otherwise, there will be more rows or more columns if the grid is higher or wider.

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

Nevertheless, as the shape of the city map can be very irregular and a significant part of the grid may be outside the boundaries, the user can also define the number of rows and columns to find a more suitable cell distribution instead of using the method of Equation 1.

Once the grid has been obtained, it is trimmed with respect to the city map and the cells outside the borders are discarded as in Figure 2c. The cells of the grid are traversed and a station is placed in the nearest valid point to the centroid of the cell. In the case of any remaining stations to be distributed, they would be placed at random valid points within randomly selected cells. Figure 2d shows an example of this distribution and Algorithm 2 describes its operation by means of pseudo-code.

There is an alternative version of this distribution in which all stations are directly placed in randomly chosen cells at a random valid point.

4.3. Radial distribution

The radial distribution aims to adapt the charging station infrastructure to certain urban areas which present greater activity towards its center in contrast to the outskirts. It makes use of a new parameter c, which defines the number of circles in which the city map will be divided.

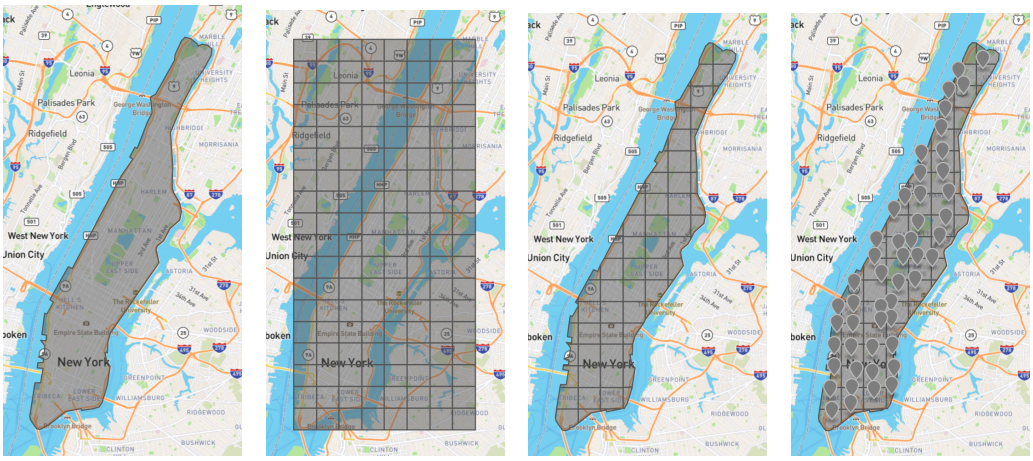
[6]The name "Uniform distribution" does not refer to a probability distribution but to how stations are divided in the city map.

Algorithm 2 Distributes n stations uniformly within the *city_map*

Require: *city_map*, n
  $valid\_points \leftarrow [\,]$
  x_{min}, y_{min}, x_{max}, $y_{max} \leftarrow city\_map.bounds()$
  $grid \leftarrow Polygon(x_{min}, y_{min}, x_{max}, y_{max})$
  $grid \leftarrow grid \cap city\_map$
  for all *cell* in *grid* **do**
    $(x, y) \leftarrow cell.centroid()$
    $valid\_points \leftarrow valid\_points \cup (x, y)$
  end for
  # Place leftover station in random points inside random cells
  while $length(valid\_points) < n$ **do**
    $cell \leftarrow randomCell(grid)$
    $(x, y) \leftarrow randomPoint(cell)$
    $valid\_points \leftarrow valid\_points \cup (x, y)$
  end while

The distribution procedure begins by defining two copies of a wider working area, created as detailed for the uniform distribution. The first copy gets divided into a configurable number of triangles, 8 by default, as it is shown in Figure 3a. These are created by joining the working area vertex and sides' middle points with the centroid of the city map. As for the second copy, it gets partitioned by c concentric circles, each with a larger radius than the previous. The initial radius r is calculated according to the dimensions of the map. Each circle gets trimmed against its previous one, starting with the last (and larger) created, so as to avoid overlap among them. The resulting polygons are also trimmed against the city map, obtaining an area with a central circle and many outer rings, as shown in Figure 3b. Bear in mind that we will be referring to both circles and rings just as circles from now on. Finally, the two modified areas are intersected, dividing each circle into up to 8 parts, obtaining a city map similar to that shown in Figure 3c.

Stations are assigned to the nearest valid point to the centroid of the polygons. To assign the stations as evenly as possible among each circle and the city map, both the number of stations per circle (n/c) and the subdivisions a circle has are taken into account. Each triangle is populated beginning from the inner circle and heading towards the outer. Once a station has been allocated to all polygons of a triangle, the next triangle will be picked



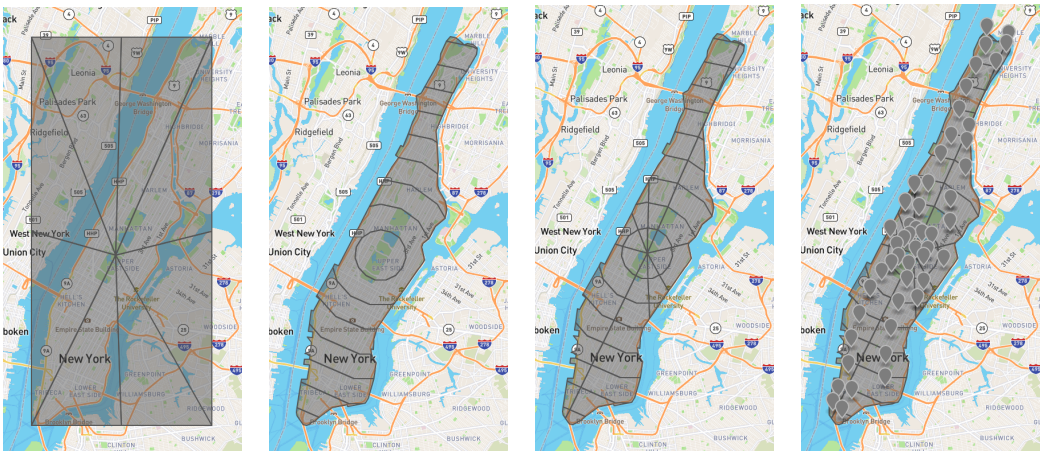
(a) City map (b) Working area or Grid (c) Grid trimmed against city map (d) Cells populated with stations

Figure 2: Uniform distribution of stations process.

with respect to the number of stations and the total number of subdivisions to evenly spread the stations within a circle. This process is described by Algorithm 3. An example of a finished distribution is shown in Figure 3d.

The procedure explained above allocates only one station inside each polygon. However, there may be a higher number of stations to allocate than polygons in the map, as the granularity of the division is decided by the user. For such cases, the leftover stations are positioned by arbitrary selecting a polygon and a valid point inside it.

A completely random variant of this distribution has also been introduced. The city map is split in the same manner and the ratio of stations per circle is considered as well. The allocation of stations, however, is performed in random valid points of arbitrary subdivisions of each circle.



(a) Triangle division of working area

(b) 8-circles division of city map

(c) Final city map division

(d) 50 stations between 8 circles

Figure 3: Radial distribution of stations process.

Algorithm 3 Distributes n stations following a radial pattern with c circles within the *city_map*

Require: *city_map*, n, c, *num_triangles*

valid_points $\leftarrow [\,]$

x_{min}, y_{min}, x_{max}, $y_{max} \leftarrow$ *city_map.bounds*()

working_area $\leftarrow Polygon(x_{min}, y_{min}, x_{max}, y_{max})$

triangle_area $\leftarrow$ *divideInTriangles*(*working_area*, *num_triangles*)

circle_area $\leftarrow$ *divideInCircles*(*working_area*, c)

working_area $\leftarrow$ *triangle_area* $\cap$ *circle_area* $\cap$ *city_map* # Calculate the number of stations to place in each circle

stations_per_circle $\leftarrow \lfloor n/c \rfloor$

while *length*(*valid_points*) $< n$ **do**

  # Select next triangle to populate

  triangle $\leftarrow$ *triangleSelection*(*stations_per_circle*, *num_triangles*)

  for all *cell* in *triangle* **do**

  $(x, y) \leftarrow$ *cell.centroid*()

  valid_points $\leftarrow$ *valid_points* $\cup (x, y)$

  end for

end while

Place leftover station in random points inside random cells

while *length*(*valid_points*) $< n$ **do**

  cell $\leftarrow$ *randomCell*(*grid*)

  $(x, y) \leftarrow$ *randomPoint*(*cell*)

  valid_points $\leftarrow$ *valid_points* $\cup (x, y)$

end while

4.4. Genetic algorithm distribution

An alternative to placing charging stations or petrol stations in a city in an intelligent way is to use genetic algorithms. Thus, this option of the generator is based on the genetic algorithm presented in the papers [24, 25]. In the previous *random*, $uniform$, and *radial* alternatives, the only necessary data are the map itself and the city limits together with the number of charging poles to be placed. This has the advantage that if no more data is available, a set of stations in the city can also be generated following the chosen distribution, which in some cases may be sufficient. However, these approximations may be unrealistic to the reality of the city, since a uniform or radial distribution may not correspond to the actual distribution and movements of the potential users of the stations.

Thus, when relevant data from the city is available the solution that the genetic algorithm can provide might be more realistic. Particularly, these data must be referred to the possible users of the stations. Hence, the data considered relevant to obtain the most accurate solution are:

- **Population or cadastral 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 sum of the population, traffic and activity data of the area covered by each of the charging poles (cp) of a solution or individual (ind) defines the utility of the solution. In addition, each of the data is balanced by a weight

ω to give it more or less importance.

$$utility(ind) = \sum_{\forall cp_i > 0 \in ind} (p_i \cdot \omega_P + t_i \cdot \omega_T + a_i \cdot \omega_A) \tag{2}$$

On the other hand, the distribution of charging stations in a city also implies high costs, so this becomes an additional criterion for the optimal distribution of charging stations. Therefore, the data to take into account in relation to costs are: cost of each station c_s and charging pole c_{cp} (additional charging pole to a station have a lower cost than the installation of the first point/station), cost per distance to transformer substation c_{dt}, and positions of transformer substations.

$$cost(ind) = \sum_{\forall cp_i > 0 \in ind} c_s + (cp_i \cdot c_{cp}) + (distenergy(s) \cdot c_{dt}) \tag{3}$$

Considering the population, traffic and social network data, that is, the utility of placing charging stations, and the costs of placing them according to their position and number, the genetic algorithm obtains a solution in which it optimizes its fitness function, which is formed by the utility and cost. Since this is a multi-criteria optimization, the genetic algorithm tries to obtain the maximum utility and the minimum cost to place the required charging stations and points.

The process of the genetic algorithm to generate the set of charging stations in the city is detailed below.

Firstly, the genetic algorithm starts from a set of Points of Interest (PoIs) that must be provided. This set of PoIs must be considerably larger than the number of stations to be placed so the genetic algorithm can select the final subset where the charging stations will be placed, and thus make sense for its application in this context. The set of PoIs can be specified by the user, or on the contrary, it can be created using different types of points such as those extracted from open data of the city of study. The set of PoIs provided defines the individual of the genetic algorithm, which is an array with the length of the set of PoIs. Then, each position of the array represents each PoI in the city, and the integer number inside will be the amount of poles to install in the PoI. This number can be from 0 to the maximum poles per station specified by the user. Therefore, the genetic algorithm generates different individuals simply by changing the integer numbers of the array and evaluating the fitness of each of them.

Secondly, all possible geographical data (in GeoJSON) must be provided so that the genetic algorithm can obtain informed solutions: cadastral data specified in number of inhabitants by areas or polygons (P); traffic data specifying a number for each street (at least the main roads) (T); geo-located activity in social networks (A); and transformer substations. In addition, the weights ω with which the cadastral, traffic and social network activity information is valued can be provided, along with the monetary costs per station (c_s), per charging pole (c_{cp}), and per distance to the transformer substation (c_{dt}).

Finally, the genetic parameters themselves such as the initial population of individuals and number of generations must be provided. In this case, the higher the values, the greater the exploration of the genetic algorithm will be; however, its computing time will also increase. These are some parameters that should be tested with low values in order to increase them according to the execution times. In our case, as the computation of fitness is relatively complex, we can start from 100 population and 50 generations, which should be computed in minutes at most. In addition, the probabilities of crossover and mutation, together with their operators, can be specified, however, the default values for this generator should be enough (see [25] for more information).

With all the parameters specified above, it is possible to run the genetic algorithm that will provide a distribution of charging stations in the city that is as optimal as possible with respect to fitness, i.e. maximizing the value of the utility, and minimizing the cost values. An example of 50 stations placed in Manhattan with the genetic distribution is represented in Figure 4. This figure shows both the location of the stations within the boundaries of the city map, as well as the Voronoi polygons that determine the areas of influence (along with a 300-meters radius circular area that intersects the Voronoi polygon) of each of the stations.

In this section, different charging station generators have been presented. The first generators, corresponding to the $random$, $uniform$, and $radial$ distributions, serve as a baseline or for situations when no city data is available. On the other hand, the generator based on the genetic algorithm, i.e., the distribution that we have named as *genetic*, has all the available information of the city where it is applied. This means that the charging station distribution solutions obtained by the genetic-based generator are potentially more realistic, that is, adapted to the demand for this type of service in the city. A detailed example of the genetic-based charging stations generators is

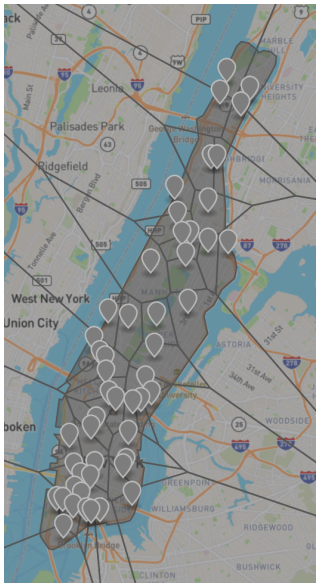


Figure 4: Genetic distribution of stations example.

shown in Section 6.1.

In addition, the electric vehicle charging station (or petrol station) generators presented in this section could also be used as a method for deciding the location of other types of services, either fixed or dynamic. For example, any other type of infrastructure such as bicycle stations or taxi ranks, as well as distribution of ambulances or emergency services, positioning of open-fleet vehicles or car-sharing services, among others.

5. Mobility data generator

With the mobility data generator, we aim to create either random or real-life inspired movement of agents for SimFleet's simulations. To generate realistic movement data, from the point of view of an agent-based simulator, means to create movements around the city which are inspired by its citizens and other users of the traffic system such as private vehicles, taxi fleets, etc. This movement can be adapted to any of the agent types that SimFleet offers by creating routes for them which have their origin and destination points located in certain areas of the city in which there is more activity. Both the areas and the type of activity are dependent on the data source we use. For instance, we could have data indicating the amount of population in a

specific neighborhood at a precise time of the day; similarly, we could gather the number of delivery vehicles that depart from a certain zip code area at a certain weekday. There are many possibilities and depending on data availability one could apply the principles of our generator to create mobility data for different situations.

Hereunder are presented random and informed versions of the mobility data generator. The random version simply creates valid routes and assigns them to the agents. The informed versions of the mobility data generators, which create realistic routes, are inspired by the work in [26], which presents an approach based on the creation of a mobility graph with real traces.

The mobility data generator develops routes of at least min_dist meters long for n agents of type t within the borders of a given city map. The delay parameter d determines the point of the simulation in which the agents will start their execution; by default, at the beginning of the simulation ($d = 0$). The number of agents per batch, $agents\_per\_batch$, is introduced to give different delays to groups of agents that will start executing simultaneously. This is most useful for scenarios with a great number of agents. If indicated, the first batch of agents will have a delay of d seconds; the following batch, a delay of $2d$, and so on. As indicated above, all generators are prepared to receive an existing SimFleet configuration file as input and fill it with agent definitions and their routes. This enables the use of the mobility data generator to introduce, in the same simulation, different types of agents in diverse quantities, with different delays and batch sizes, achieving a complex simulation scenario.

5.1. Random movement generator

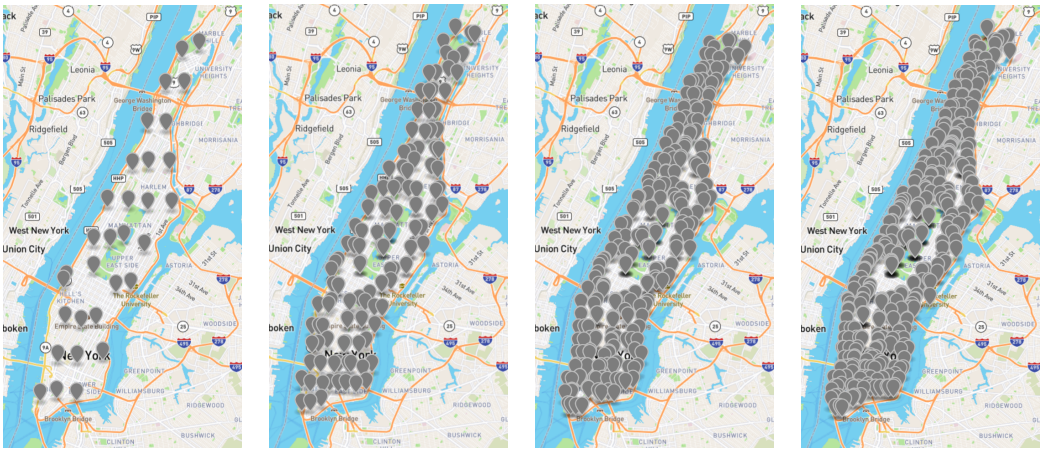
The movement of the random generator is created by designating a random route (random origin and destination points) for the agents to follow. Both the origin and destination points must be valid points inside the city map, and they must be a minimum of min_dist meters apart. This process is repeated to create and assign routes for n agents of type t. The origin point indicates the agent the location in which it will spawn, whereas the destination point determines the place where the execution will finish. Transport type agents can travel by themselves. However, if the agent is of customer type or a package, the movement will actually be performed by the transport agent that carries it after picking it up.

Figure 5: 50 randomly generated routes in Manhattan, indicated by a line connecting origin and destination points.

5.2. Informed movement generator

The mobility data generator's informed variant attempts to replicate more accurate movements across the city map. For this, it is important to provide relevant data to the generator on which to base the agents' routes. This data can be accessed from different sources; often open data portals the government of a city or nation provides to its inhabitants. For our generator, we used the following data (already presented in Section 4.4): population or cadastral information (P); traffic information (T); and Twitter activity (A).

The data is used to establish a probability distribution between a series of points available in the city map. The assignment of the routes' origin and destination will be carried out according to the distribution. The process starts by generating a collection of available points. As described for the uniform distribution (see Section 4.2), the city map (M) is split as a grid 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 closest valid point to the centroid of G_i. The grid is divided by a configurable number of rows and columns. Such a number directly affects the granularity of the system, as a larger number of cells implies more accessible points (see Figure 6). 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 (d) 40 rows and cols

Figure 6: Number of available points according to map division granularity.

By combining the city data with M, we join, for every polygon G_i, the population, traffic and Twitter activity volumes that occur inside 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 compute the likelihood associated with each point O_i as in Equation (4):

$$prob(O_i) = w_p \cdot \frac{p_i}{\sum_{j=1}^{n} p_j} + w_t \cdot \frac{t_i}{\sum_{j=1}^{n} t_j} + w_a \cdot \frac{a_i}{\sum_{j=1}^{n} a_j}; \text{ with } w_p + w_t + w_a = 1 \tag{4}$$

where w_p, w_t and w_a are weights that control the effect on the probability of each of the variables. By having $w_p + w_t + w_a = 1$ we obtain a probability distribution among points that ensures that the addition of the probability of each point is equal to 1. An intuitive example of this can be seen in Table 1 and Equation 5. Finally, the set of available points (S) and their resulting probabilities: $S = \{(O_1, p(O_1)), (O_2, p(O_2)), \ldots, (O_n, p(O_n))\}$; are taken into account for route generation.

$$prob(O_1) = 0.5 \cdot \frac{5000}{10000} + 0.3 \cdot \frac{1500}{8000} + 0.2 \cdot \frac{3500}{5000} = 0.44625 \tag{5}$$

When all points in S have an associated probability, the process to define the n routes starts (see Figure 7 for examples). The approach is very similar to the one described for the random mobility data generator, except this

Point (O_i)	Population in O_i (p_i)	Traffic in O_i (t_i)	Activity in O_i (a_i)	Probability ($prob(O_i)$)
O_1	5000	1500	3500	0.44625*
O_2	3700	4500	1000	0.39375
O_3	1300	2000	500	0.16
Total values	10000	8000	5000	**1.0**
Weights	0.5 (w_p)	0.3 (w_t)	0.2 (w_a)	

Table 1: Example of a probability distribution among 3 points. *Value obtained according to Equation 5.

time the origin and destination points are picked from S with respect to their probability and guaranteeing the *min_dist* between both points.

(a) 10x10 map (b) 20x20 map (c) 30x30 map (d) 40x40 map

Figure 7: 50 routes examples in Manhattan with different granularity.

5.3. Regression mobility data generator

The regression version of the mobility data generator creates a data model and makes use of it to enrich the simulations with a real-life inspired movement of agents.

In this work, using New York City as an example, we used the regression mobility data generator to reproduce a realistic taxi demand across the city map. For this, we employ the TLC Trip Record Data[7] of the city of New York.

[7] `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`

This dataset contains records of taxi services which are defined, among other parameters, by the service start date (month, day and time of the day) as well as an origin and destination taxi zone identifiers. With this information, we divided our simulation's city map into its corresponding taxi zones and created a regression model which can estimate the amount of demand in each taxi zone for a specific month, weekday and time of the day. Please refer to Section 6.2 for a detailed explanation of how the data was processed.

Therefore, the generated movement is represented by a certain number of customer agents that will spawn distributed among the different taxi zones according to the predicted value of each zone. Additionally, as the dataset includes not only the origin but the destination taxi zone of each service, we can also estimate a service destination. According to the observed services, we can assign a probability to every taxi zone that indicates how likely it is to be the destination of a service originated in a specific taxi zone at a certain month, weekday and time of the day. By pseudo-randomly choosing a destination according to such probabilities, we complete the routes of the customer agents.

The customers will be picked up and driven to their destinations by transport agents. Consequently, we have generated mobility data which is in line with real-life taxi demand as well as origin and destination areas. Optionally, we could develop a relocation service for transport agents, using the same regression model, which sent idle taxis to zones where more demand is likely to appear in the next minutes or hours; although that is outside of the scope of this work, as it has more to do with transport agent strategic behavior.

As can be seen, we have designed and implemented three approaches to mobility data generation with different levels of complexity. The random movement generator may be useful to establish baseline measurements in simulations, so as to compare them against other systems. As for the more informed versions, they make use of real-world data, processing it with different techniques to create realistic routes. Assigning such routes to agents in our simulations we obtain a better representation of the real urban traffic system. The regression mobility data generator employs the most complete approach, although in a very general perspective, as it can be used with many types of input data. In the following Section 6.2 we illustrate the use of the latter generator with a detailed example.

6. Case study

In this section, we present a case study on the island of Manhattan in order to illustrate the use of the charging stations generator as well as the mobility data generator described in Sections 4 and 5. Throughout the previous sections we have shown illustrative examples of how each of the generators works, i.e., each of the distributions of the charging stations generator ($random$, $uniform$, $radial$, $genetic$), and the mobility data generators (random, informed, and regression). Thus, in this case study we focus on what we can consider to be the most informed and potentially most realistic generator for each case. Thus, Section 6.1 details the use of the genetic algorithm-based charging station generator, and Section 6.2 details the use of the regression-based mobility data generator to generate a realistic demand for taxis in the Manhattan island.

6.1. Genetic generation of stations

In this section we present the use and results obtained from the charging stations generator based on the genetic algorithm of Section 4.4 on the case study located in Manhattan. Concretely, we specify the pre-processing of the Manhattan data as well as the parameters used to prepare the experimentation with the generator. Then, the results of executions with different number of charging stations are shown.

6.1.1. Data pre-processing

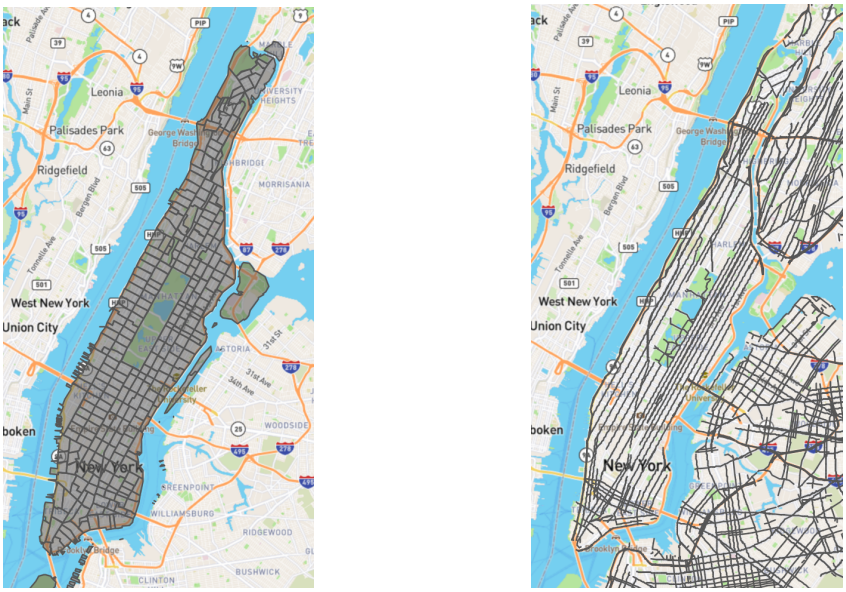
One of the crucial parts for a successful operation of the charging stations generator based on the genetic algorithm is the data on which it is based. For the genetic algorithm we will use data on population, traffic, and social media activity (in this case, Twitter). The population and traffic data have been extracted from the New York open data portal[8]. Regarding the Twitter data, it has been obtained using uTool [8], a tool that captures geo-located tweets during a specific period of time.

The population data of Manhattan has been extracted from two different datasets. On the one hand, the 2010 Census tracts[9] are used as the division of different areas of Manhattan. Then, this data is processed and merged

[8] `https://opendata.cityofnewyork.us/`

[9] `https://data.cityofnewyork.us/City-Government/2010-Census-Tracts/fxpq-c8ku`

with the census demographics at the neighborhood tabulation area (NTA) level[10]. Concretely, the number of population at 2010 of each NTA code from the last dataset is assigned to the corresponding polygon that defines the NTA (extracted from the first dataset). This produces a GeoJSON file with the population by areas of Manhattan (represented in Figure 8a) that can be provided to the genetic algorithm.



(a) Manhattan population by neighborhood tabulation areas

(b) Manhattan traffic by streets

Figure 8: Manhattan population and traffic.

Regarding the traffic data of Manhattan, two different datasets have also been used. On the one hand, the New York City street centerline dataset[11] has been used for having the polylines that define the streets of Manhattan. On the other hand, the traffic volume counts from 2014 to 2019[12] have been

[10] https://data.cityofnewyork.us/City-Government/Census-Demographics-at-the-Neighborhood-Tabulation/rnsn-acs2

[11] https://data.cityofnewyork.us/City-Government/NYC-Street-Centerline-CSCL-/exjm-f27b

[12] https://data.cityofnewyork.us/Transportation/Traffic-Volume-Counts-2014-2019-/

used to have the number of cars moving through the streets of Manhattan. These data have been processed to establish a correspondence between the street names represented in both datasets, as there is no compatible identification or code that we can use for our needs. All traffic volume counts, which are separated by date and hour from 2014 to 2019 are aggregated to have a single number per street. This aggregation gives a global picture of the traffic volume in Manhattan that allows the genetic algorithm to discriminate between different streets or areas. All this process ends up with a GeoJSON file (represented in Figure 8b) which is ready to be used by the genetic algorithm.

The geo-located tweets to use with these experiments are from 2017 to 2019, that is, 3 complete years. The total number of geo-located tweets is roughly 4.5 million for New York City. However, this volume of data is difficult for the genetic algorithm to process, as it would be very slow for each of the solution evaluations. It is therefore more appropriate to reduce it by using the *geohash* system[27], which allows the encoding of a geographical location using a string of characters. In our case, the advantage lies in using a certain precision (7 characters) to reduce the 4.5 million points at which the tweets are located to a smaller set. Hence, all tweets that have the same geohash are grouped together in the same "bucket", so that we end up with a set of points that represent each of the geohashes with the number of tweets that have been made in that area. Thus, by applying geohash with precision 7 we reduce the 4.5 million tweets of New York City to 41194 geohash areas, and specifically, 5467 are the geohash areas corresponding to the island of Manhattan that group together around 1.8 million tweets.

The genetic algorithm needs to start from a set of points of interest (PoIs) in the city where it is applied. This set of PoIs must be considerably larger than the number of stations to be placed in order for the genetic algorithm to determine which combination of points is the most appropriate. In other words, if the set of PoIs were very similar to the number of stations to be placed, a genetic algorithm would not be necessary and could be obtained by brute force. For these experiments, we composed a set of 409 PoIs in the main isle of Manhattan which are separated at least 150 meters. These PoIs represent areas in which a charging station could be considered given their interest and the activity generated around them. Examples include

---

`ertz-hr4r`

existing petrol stations, museums, monuments, tourist attractions, cinemas or theaters, shopping areas, restaurants, among others.

The parameters used to configure the genetic algorithm to obtain the charging stations to be placed in Manhattan are the following:

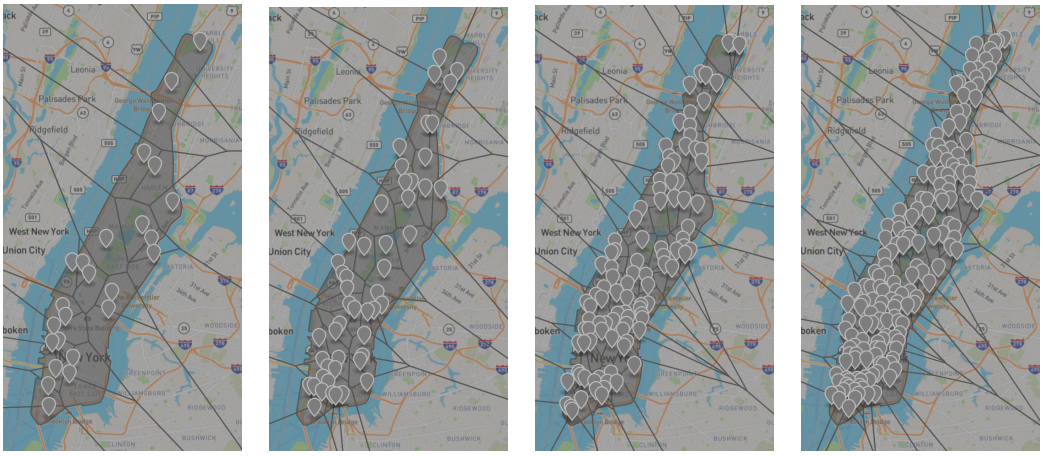
- Population of individuals is set to 100. This parameter determines the number of individuals (potential solutions) that are maintained and used during the evolution.
- The number of generations in which the genetic algorithm evolves the individuals is established at 50. The evolution is made by applying crossover and mutation operators with some probabilities, and selecting the best individuals of each generation.
- The weights to balance the importance of the population, traffic, and Twitter activity are: $\omega_P = 0.4$, $\omega_T = 0.3$, and $\omega_A = 0.3$, respectively. These weights are chosen by the user depending on the problem to be optimized. Different values may give better or worse solutions. In this case it has been decided, after a few tests, to give a little more weight to population versus traffic and social networks.
- The cost of each station is $c_s =$€40000, and the cost of a charging pole is $c_{cp} =$€10000.
- The influence radius that each PoI considers about the area that covers (with the intersection of the Voronoi polygon) is set to 300 meters.
- The crossover and mutation probabilities during the evolution are 0.5 and 0.2, respectively. The crossover operator is the graph operator presented in [24] (Section 3.4). The mutation operator is the uniform with a mutation probability of each gene of 0.05.

The rest of the parameters remain with default values since they are out of the scope of these experiments. Additionally, the transformer substations or any forbidden areas have not been considered. For more information about all the genetic algorithm parameters we refer the reader to [25].

6.1.2. Execution and results

With all the population, traffic, and Twitter data for Manhattan Island extracted and processed along with the set of PoIs, as well as the other parameters specified in the previous section, we can proceed to run the genetic

algorithm. Specifically, we have performed different runs in which we placed 25, 50, 100, and 200 charging stations in Manhattan. The representation of the charging stations that have been placed along with the resulting Voronoi diagram can be seen in Figure 9.



(a) 25 stations (b) 50 stations (c) 100 stations (d) 200 stations

Figure 9: Charging stations in Manhattan obtained by the genetic-based generator.

In the example with 25 charging stations in Figure 9a it can be seen that the charging stations are quite dispersed, although some are more concentrated in the southern area, probably due to the higher activity in the area. However, as there are few stations to be placed there are some areas that are certainly far away from any stations.

The example depicted in Figure 9b with 50 charging stations presents a distribution that apparently better covers the island of Manhattan, and especially the southern part, in this case, below Central Park. However, there are still some large areas without adequate charging station coverage.

The example of 100 charging stations in Figure 9c already shows a much larger coverage of the entire island of Manhattan, except for the part of central park, where it is not possible to place many charging stations, and also some areas in the north that have fewer stations than in the south, probably due to relatively less activity.

Finally, the example in Figure 9d with 200 charging stations already shows a much larger coverage of Manhattan compared to the previous examples. In this case, it becomes more evident that the southern area is fully populated

with charging stations, as well as the northern area. In fact, the only gaps that can be seen are in the central park area where it is not possible to place too many stations, and the amount of population and traffic in that particular area is much smaller as it is a significantly large park with no housing or roads.

stations	cost (€)
25	1,250,000
50	2,500,000
100	5,000,000
200	8,600,000

Table 2: Monetary cost of each of the charging station distributions in Manhattan.

Table 2 shows the monetary cost of each of the Manhattan charging station distributions seen in Figure 9. For the cases of 25, 50, and 100 stations, the monetary cost is the result of multiplying the number of stations by €50000, since in all three cases stations with only one charging point have been placed (the cost of one station has been set at €40000, and the cost of each charging point at €10000). For the case of 200 stations, 165 stations are placed, in which several of them have 2-3 charging points. This implies a cost of €40000 for each of the 165 stations (without considering the charging points), and then €10000 for each of the 200 charging points in total. This happens because the genetic algorithm tries to maximize the coverage of the stations to have more utility, so in cases with 100 stations or less, it only places one charging point per station. However, when it can place 200 charging points, it can afford to place more than one charging point per station.

Table 3 summarizes the results obtained concerning the utility and cost of running the genetic algorithm for 200 charging points in Manhattan. Each row of Table 3 corresponds with a pair of crossover and mutation probability values. Note that the sum of both values must not be greater than 1. In addition, results for values $cxpb = 1, mutpb = 0$ and $cxpb = 0, mutpb = 1$ have been excluded since they did not arrive at any feasible solution. The best results are obtained with the crossover probability at $cxpb = 0.5$ and the mutation probability at $mutpb = 0.2$. Although also the values $cxpb = 0.5, mutpb = 0.05$ and $cxpb = 0.75, mutpb = 0$ obtain similar results with which there is no significant difference. Thus, we can conclude that any of

cxpb	mutpb	utility	cost (€)
0	0.05	0.03068	9,130,000
0	0.2	0.03195	8,850,000
0	0.5	0.03351	8,650,000
0.25	0	0.03741	8,890,000
0.25	0.05	0.04110	9,410,000
0.25	0.2	0.03978	8,920,000
0.25	0.5	0.04317	8,760,000
0.5	0	0.04156	8,880,000
0.5	0.05	**0.04831**	**8,930,000**
0.5	0.2	**0.05230**	**8,600,000**
0.5	0.5	0.03923	8,930,000
0.75	0	**0.04943**	**8,970,000**
0.75	0.05	0.04263	8,930,000
0.75	0.2	0.04516	8,890,000

Table 3: Utility and cost results for different runs of the genetic algorithm varying the crossover (cxpb) and mutation (mutpb) probabilities for the case with 200 charging points.

these 3 combinations of crossover and mutation probability can obtain the best results for the problem we are dealing with.

6.2. Realistic route generation

To illustrate the use of our regression mobility data generator, in this section we describe a complete example in which the TLC Trip Record Data of New York City is used to train a regression model that predicts the taxi demand per taxi zone in a concrete date. The predictions are then used to reproduce the demand on SimFleet's simulations by generating routes for taxi service customers.

6.2.1. Data pre-processing

From the many parameters by which a taxi service is characterized, only the service start date and the origin taxi zone ID are kept. Then, the date is split in *month* (1-12), *weekday* (0-6) and *hour* (0-23). The minutes and seconds values of the service start date are discarded so as to group together all services which started during the same hour. Next, the taxi services get grouped by month, weekday, hour and origin taxi zone id and a new column *demand* is created by counting the services being joined together

(see Table 4). By doing this, the demand value indicates the number of taxi services that started during the indicated month, weekday and hour on the corresponding taxi zone. Dividing the demand value by the total demand (sum of all demand values) we obtain instead an estimation of the percentage of total demand that occurs in each zone.

Month	Weekday	Hour	Origin	Demand
1	0	0	4	41
1	0	0	13	59
1	0	0	24	48
...	...	...	...	...
6	6	23	261	18
6	6	23	262	12
6	6	23	263	76

Table 4: Clean taxi service dataset grouped by month, weekday, hour and origin taxi zone ID. The demand column indicates the number of trips departing from the origin taxi zone at the indicated date and time.

The process mentioned above can be applied to datasets with services from many months or even many years, as long as they get merged together in the final data model. Optionally, it would be possible to create an individual data model for each month with its corresponding regression model. We decided to build a data model with yellow taxi services of the months of January to June of 2019. Also, to restrain the simulation city map to a smaller area, we considered a restricted set of taxi zones inside the Manhattan borough as can be seen in Figure 10.

The last step of the pre-processing is to treat the origin taxi zone identifiers as categorical variables, which we do by encoding them into a one-hot encoding (see Table 5). The data is then ready to be divided into training and test sets and feed to our regression model.

6.2.2. Execution and results

We used an automated Machine Learning process (included in the TPOT tool [28]) to find the most adequate regression model to train with our data. This process automates the pipeline design of machine learning models by performing a search using genetic algorithms to combine and evaluate different models and hyper-parameters.

Figure 10: Restricted set of taxi zones of the Manhattan borough.

The pipeline found was composed by a Stacking Estimator with a Decision Tree Regressor followed by a Random Forest Regressor. These models are part of the scikit-learn toolkit [29]. The concrete parameters of each model, also tuned by the automated Machine Learning process, are presented in Table 6. This model achieved an accuracy of 0.95 over the test set, which we consider enough given its purpose, which is to generate realistic data.

Once the regression model has been trained, we can generate mobility data as a prediction of taxi demand over the different taxi zones. To do so we first define the simulation time span, the amount of time we want our simulation to model. In the following example, we model a simulation which takes place over a Monday (weekday 0) of January (month 1), from 9:00 to 14:00. In addition, we indicate a number of customers per hour, which will inform the simulator about the number of customer agents we want our system to spawn each simulation hour[13]. For our example, we set the number of customers per hour to 1000. Let it be noted that such a number could be indicated for longer time periods like many days or even a whole month; the simulator would then adjust the duration of the simulation accordingly.

[13]The real-time duration of an hour of simulation can be adjusted by the user.

Month	Weekday	Hour	4	13	24	...	261	262	263	Demand
1	0	0	1	0	0	...	0	0	0	1.093e-06
1	0	0	0	1	0	...	0	0	0	1.066e-07
1	0	0	0	0	1	...	0	0	0	1.230e-06
...	...	...	...	...	...	...	...	...	...	...
6	6	23	0	0	0	...	1	0	0	4.798e-07
6	6	23	0	0	0	...	0	1	0	3.198e-07
6	6	23	0	0	0	...	0	0	1	2.026e-06

Table 5: Dataset formatted to train a regression model. The origin taxi zone ID values have been converted to one-hot encoding and the demand presented as a percentage of the total demand.

Decision Tree Regressor		**Random Forest Regressor**	
max_depth =	10	max_features =	0.5
min_samples_leaf =	17	min_samples_leaf =	1
min_samples_split =	7	min_samples_split =	15
		n_estimators =	100
		accuracy =	0.95

Table 6: Best pipeline found for our dataset.

With the simulation characterized by the aforementioned parameters, the generator builds the samples to pass to the regression model. As our example takes place in different hours, the model builds, for each hour, samples of services which depart from every taxi zone considered in the simulation. If the simulation was set in different days or months, the generator would create samples in a similar manner to the one described, making sure every possible combination of the parameters is considered.

The samples are passed to the regression model, which outputs a percentage of demand for each one of them. As we mentioned on the data pre-processing (Section 6.2.1), our model was trained with data belonging to 6 months. This means the predicted demand is a percentage of the total demand of 6 months. Because of that, we decided to normalize the demand percentage across all samples with the same month, weekday and hour. For our example, that implies normalizing the demand across samples with the same hour, which gives us 6 sets of data, each corresponding to an hour (9:00 to 14:00), with normalized demand. Each of the 6 datasets indicates the per-

centage of demand to be expected in each taxi zone during the same hour. Following, the demand percentage is multiplied by the number of customers per hour (1000), finally obtaining the number of customers that will spawn in each taxi zone during every hour of the simulation. A representation of the dataset for the 9:00 hour can be seen in Table 7.

Month	Weekday	Hour	Origin	Demand	Customers
1	0	9	4	0.002154	2
1	0	9	13	0.013161	13
1	0	9	24	0.003504	3
...	...	...	...	...	...
1	0	9	261	0.003842	3
1	0	9	262	0.023182	23
1	0	9	263	0.026261	26

Table 7: Data predicted by our regression model. The demand column indicates the percentage of the hourly demand that occurs in a determined taxi zone. The customers column expresses such demand in terms of the number of customers departing from the determined taxi zone.

As it can be inferred, the normalization of data can be adapted to a concrete simulation setup; i.e., a concrete set of values for the month, weekday, hour and customer amount parameters. If a simulation was defined to take place over many days, and the number of customers was also indicated by day, the demand percentage could be normalized across a day instead of an hour.

As for the last step, the route creation, the generator defines a route for each customer which starts in a random valid location inside its origin taxi zone. The destination taxi zone can be chosen semi-randomly according to the observed taxi services (as commented in Section 5.3) or completely random among all considered taxi zones. Once the destination is set, a random valid point inside it is fixed. The origin and destination points are then passed to our routing service and the final route is obtained. During the simulation development, the customer agents will spawn in their origin points. To divide the demand of a single simulation hour into different time intervals the delay parameter (commented in Section 5.2) can be used.

7. Conclusions

This paper has presented a proposal for the generation of more realistic data in agent-based simulation tools related to mobility and transportation. Specifically, the data generation has been focused on the location of EV charging stations and on the generation of routes within the urban environment, but it can be easily adapted to generate other types of infrastructure or data to be required in the urban environment. For the development of the proposal, the simulation tool SimFleet has been used, on which different generators of each kind have been integrated.

We highlight the development of two more complex generators based on artificial intelligence techniques. In the case of the generation of charging stations, a genetic algorithm has been used to optimize the location of stations in the city based on information from open data and other data sources. In the case of route generation, a regression algorithm has been used to generate more realistic routes from historical mobility data. With both generators, the users can simulate different distributions over a city or metropolitan area by recreating mobility, using real data to analyze and compare each distribution. This improves the simulation results facilitating the decision making in municipalities.

Moreover, a case study has also been developed in order to illustrate the use of such generators in common. Specifically, an example has been developed on the island of Manhattan in New York using different sources of real data. Results allow to ensure the usefulness of these generators. As a future work we aim to propose an evolution of the proposed generators for their possible adaptation and integration in other simulation tools similar to SimFleet. The work of adaptation to other tools would mainly consist of transforming the input data generated in our proposal into the appropriate format in each case. As an example, work has begun on developing a data transformation algorithm for the MatSim tool, which requires the input information in xml format divided into different files. On the other hand, we also propose the integration of vehicle reallocation strategies for open fleets in the generators.

Acknowledgments: This work is partially supported by grant RTI2018-095390-B-C31 funded by MCIN/AEI/ 10.13039/501100011033 and by "ERDF A way of making Europe". Pasqual Martí is supported by grant ACIF/2021/259 funded by the "Conselleria de Innovación, Universidades, Ciencia y Sociedad Digital de la Generalitat Valenciana".

References

[1] H. Noori, Realistic urban traffic simulation as vehicular ad-hoc network (vanet) via veins framework, in: 2012 12th Conference of Open Innovations Association (FRUCT), IEEE, 2012, pp. 1–7.

[2] A. Drogoul, D. Vanbergue, T. Meurisse, Multi-agent based simulation: Where are the agents?, in: International Workshop on Multi-Agent Systems and Agent-Based Simulation, Springer, 2002, pp. 1–15.

[3] P. Davidsson, Multi agent based simulation: beyond social simulation, in: International workshop on multi-agent systems and agent-based simulation, Springer, 2000, pp. 97–107.

[4] S. Abar, G. K. Theodoropoulos, P. Lemarinier, G. M. O'Hare, Agent based modelling and simulation tools: A review of the state-of-art software, Computer Science Review 24 (2017) 13–33.

[5] P. Martí, J. Jordán, J. Palanca, V. Julian, Load generators for automatic simulation of urban fleets, in: International Conference on Practical Applications of Agents and Multi-Agent Systems, Springer, 2020, pp. 394–405.

[6] J. Palanca, A. Terrasa, C. Carrascosa, V. Julián, Simfleet: A new transport fleet simulator based on mas, in: International Conference on Practical Applications of Agents and Multi-Agent Systems, Springer, 2019, pp. 257–264.

[7] J. Jordán, J. Palanca, E. Del Val, V. Julian, V. Botti, A multi-agent system for the dynamic emplacement of electric vehicle charging stations, Applied Sciences 8 (2) (2018) 313.

[8] E. del Val, J. Palanca, M. Rebollo, U-tool: A urban-toolkit for enhancing city maps through citizens' activity, in: International Conference on Practical Applications of Agents and Multi-Agent Systems, Springer, 2016, pp. 243–246.

[9] A. L. Bazzan, F. Klügl, A review on agent-based technology for traffic and transportation, The Knowledge Engineering Review 29 (3) (2014) 375.

[10] M. Behrisch, L. Bieker, J. Erdmann, D. Krajzewicz, Sumo–simulation of urban mobility: an overview, in: Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation, ThinkMind, 2011.

[11] K. W Axhausen, A. Horni, K. Nagel, The multi-agent transport simulation MATSim, Ubiquity Press, 2016.

[12] M. Adnan, F. C. Pereira, C. M. L. Azevedo, K. Basak, M. Lovric, S. Raveau, Y. Zhu, J. Ferreira, C. Zegras, M. Ben-Akiva, Simmobility: A multi-scale integrated agent-based simulation platform, in: 95th Annual Meeting of the Transportation Research Board Forthcoming in Transportation Research Record, 2016.

[13] T. Sakai, A. R. Alho, B. Bhavathrathan, G. Dalla Chiara, R. Gopalakrishnan, P. Jing, T. Hyodo, L. Cheah, M. Ben-Akiva, Simmobility freight: An agent-based urban freight simulator for evaluating logistics solutions, Transportation Research Part E: Logistics and Transportation Review 141 (2020) 102017.

[14] M. Fellendorf, Vissim: A microscopic simulation tool to evaluate actuated signal control including bus priority, in: 64th Institute of Transportation Engineers Annual Meeting, Vol. 32, Springer, 1994, pp. 1–9.

[15] B. Torabi, M. Al-Zinati, R. Z. Wenkstern, Matisse 3.0: A large-scale multi-agent simulation system for intelligent transportation systems, in: International Conference on Practical Applications of Agents and Multi-Agent Systems, Springer, 2018, pp. 357–360.

[16] J. M. Lozano Domínguez, T. J. Mateo Sanguino, Review on v2x, i2x, and p2x communications and their applications: A comprehensive analysis over time, Sensors 19 (12) (2019).
URL `https://www.mdpi.com/1424-8220/19/12/2756`

[17] G. Merkuryeva, V. Bolshakovs, Vehicle schedule simulation with anylogic, in: 2010 12th International Conference on Computer Modelling and Simulation, IEEE, 2010, pp. 169–174.

[18] Y. Yang, J. Li, Q. Zhao, Study on passenger flow simulation in urban subway station based on anylogic, Journal of Software 9 (1) (2014) 140–146.

[19] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun, Carla: An open urban driving simulator, arXiv preprint arXiv:1711.03938 (2017).

[20] S. Shah, D. Dey, C. Lovett, A. Kapoor, Airsim: High-fidelity visual and physical simulation for autonomous vehicles, in: Field and service robotics, Springer, 2018, pp. 621–635.

[21] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, A. Sumner, Torcs, the open racing car simulator, Software available at http://torcs. sourceforge. net 4 (6) (2000) 2.

[22] C. Sippl, B. Schwab, P. Kielar, A. Djanatliev, Distributed real-time traffic simulation for autonomous vehicle testing in urban environments, in: 2018 21st International Conference on Intelligent Transportation Systems (ITSC), IEEE, 2018, pp. 2562–2567.

[23] J. Palanca, A. Terrasa, V. Julian, C. Carrascosa, SPADE 3: Supporting the New Generation of Multi-Agent Systems, IEEE Access 8 (2020) 182537–182549. doi:10.1109/ACCESS.2020.3027357.

[24] J. Jordán, J. Palanca, E. del Val, V. Julian, V. Botti, Localization of charging stations for electric vehicles using genetic algorithms, Neurocomputing (2020). doi:https://doi.org/10.1016/j.neucom.2019.11.122.

[25] J. Palanca, J. Jordán, J. Bajo, V. Botti, An energy-aware algorithm for electric vehicle infrastructures in smart cities, Future Generation Computer Systems 108 (2020) 454 – 466. doi:https://doi.org/10.1016/j.future.2020.03.001.

[26] A. Förster, A. Bin Muslim, A. Udugama, Trails - a trace-based probabilistic mobility model, in: Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWIM '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 295–302. doi:10.1145/3242102.3242134.

[27] G. Niemeyer, Geohash, http://geohash.org/site/tips.html, accessed on 23-01-2019 (2008).

[28] R. S. Olson, N. Bartley, R. J. Urbanowicz, J. H. Moore, Evaluation of a tree-based pipeline optimization tool for automating data science, in:

Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, ACM, New York, NY, USA, 2016, pp. 485–492. doi:10.1145/2908812.2908918.

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.