Final Assignment

A Multi Agent System for Autonomous Public Transport in Amsterdam

2016-2017

1 Introduction

"Autonomous cars will change the way we think about traffic. Today traffic is primarily regarded as the result of the independent actions of thousands of drivers. A view from above on any city would show large numbers of vehicles pursuing their own trajectories through the maze of roads. The cities' traffic management systems try their best to observe, identify and somewhat channel the grand flows.

At first glance, autonomous vehicles do not seem to change this situation very much. From above, self-driving cars will not be distinguishable from human driven cars and they too, will seek their individual paths through the maze of roads. The picture changes, however, when we consider fleets of self-driving cars. Recent statements by Ford, Uber, BMW and others clearly show that fleets of self-driving cars will emerge early and have the potential to capture a significant share of individual motorized mobility.

This introduces a crucial difference: Fleet vehicles no longer pursue their local optimum; rather than completing the individual trip as quickly as possible, fleet management will seek to maximize throughput for all of its vehicles – for the fleet as a whole. The operational goals of fleet management are therefore very much aligned with the traffic flow goals of a city as a whole."

As illustrated by this fragment[1], autonomous vehicles have the potential to result in a radically new transportation paradigm, in particular when they are viewed as multiple (distributed) entities that are part of a larger overarching system. Indeed, the ability of modern cars to sense information in their environment, communicate with other cars and systems, and autonomously make informed decisions makes that a group of autonomous cars can nicely be conceptualized as a multi-agent system.

The future of autonomous vehicles will not stop at personal use (and privately-owned ride-sharing services), but will also impact the role of public transportation[2]. Many cities[3] and countries, including the Netherlands[4], are currently considering introduc-

[1] Taken from http://www.driverless-future.com/?cat=26 (accessed January 11, 2017).
[2] http://transloc.com/will-autonomous-vehicles-kill-public-transit/ (accessed January 11, 2017).
[3] http://ieeexplore-spotlight.ieee.org/article/autonomous-vehicle-public-transportation-system/ (accessed January 15, 2017).

ing next-generation transportation services that make use of autonomous cars and buses.

In line with these developments, the current assignment addresses the design of a multi-agent system that acts as an intelligent public transport service for the city of Amsterdam. The system should be implemented in the multi-agent modelling environment NetLogo. This assignment involves many of the challenges addressed in the Multi-Agent Systems course, including reasoning, communication, coordination, group decision making, and possibly coalition formation, bargaining and argumentation.

The details of the assignment are introduced in Section 2. A week-by-week schedule describing the various milestones that should be achieved during the course as well as the criteria for grading are provided in Section 3.

2 Assignment

The main goal of the assignment is to develop a multi-agent system for public transport, which consists of a number of intelligent vehicles (buses)[5]. These vehicles should bring passengers to their desired destination as efficiently as possible (i.e., as quickly as possible, but also at the lowest possible cost; see below for details).

Map. The point of departure for this assignment is the map of Amsterdam. On this map, 24 locations have been marked that will play the role of bus stops (see Figure 1). In this figure, several connections between these bus stops have been highlighted in pink. Autonomous vehicles are assumed to travel via these connections from bus stop to bus stop. Hence, for this assignment the transportation network of Amsterdam is represented as an undirected, not fully connected graph with 24 vertices and various edges between them. In addition, edges are labeled with a number that represents the distance between two bus stops. For example, the distance from bus stop *Dam* to bus stop *Evertsenstraat* is exactly 10 *patches*[6], whereas the distance from *Dam* to *Centraal* is 3.16 *patches* (which equals $\sqrt{10}$, as *Centraal* is located 1 patch to the east and 3 patches to the north of *Dam*). Obviously, the larger the distance, the more time it will take for a vehicle to travel along that edge[7].

---

[4] http://www.treinreiziger.nl/actueel/binnenland/ov_visie_2040:_sneller_en_slimmer_openbaar_vervoer-147845 (accessed January 15, 2017).

[5] Throughout this document, the terms *agent*, *bus* and *vehicle* are interchangeable.

[6] A *patch* equals approximately 500 meters (see the section on the NetLogo environment).

[7] For simplicity, edges do not align 100% with actual streets.

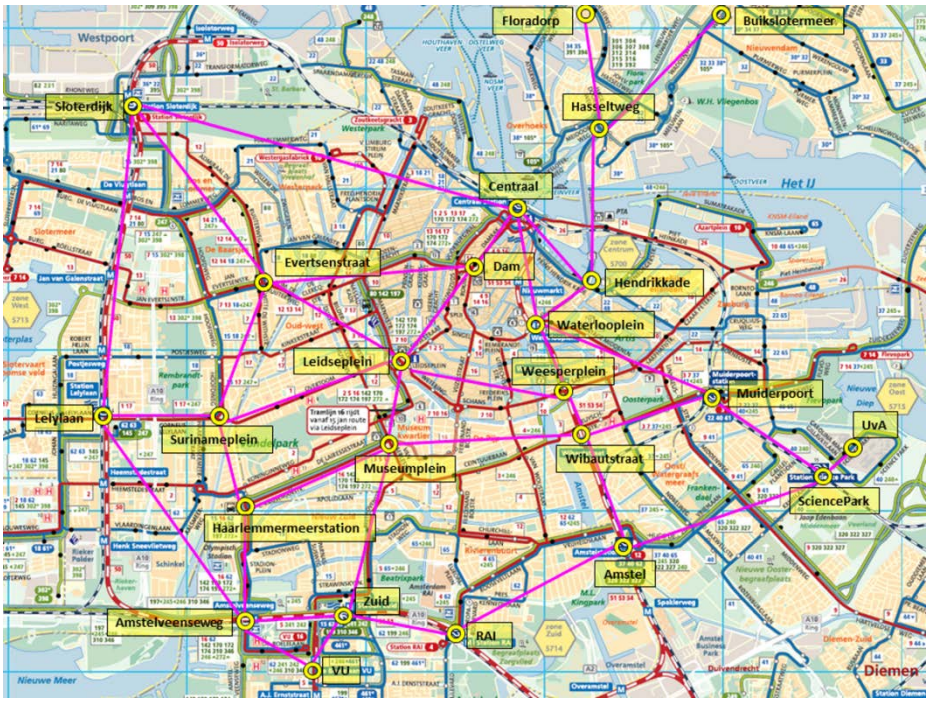

Fig. 1. Map of Amsterdam

Vehicles. Vehicles are intelligent agents with the ability to autonomously travel through the city and transport passengers between the various bus stops. Basically, they can perform three types of actions, namely *pick up passengers* at bus stops, *travel* to bus stops, and *drop off* passengers at bus stops. To acquire information about the environment, they can *observe* the number of passengers (and their destinations) at all bus stops. This information is assumed to be publicly available via a central database that is fed by sensors in the bus stops[8]. Furthermore, vehicles may also *communicate with each other* in order to exchange information. However, communication is not free of charge: there are certain *communication cost* associated with each message that is sent.

Finally, vehicles are allowed to *add new vehicles* to the system. This can be done, for instance, when the amount of passengers in the city is too large to be transported in a satisfactory manner. However, the decision to add new vehicles should be made with care, since there are cost involved, and new vehicles are used for the entire simulation (i.e., they cannot be 'returned' if they are not used anymore). In particular, each vehi-

[8] So, vehicles do not need to be physically present at a particular bus stop to observe the number of passengers at that stop. However, information about the number of passengers *inside a vehicle* is assumed to be local. In case other vehicles require this information, it needs to be communicated explicitly to these other vehicles.

cle has a number of fixed characteristics: a carrying *capacity* (= a maximum number of passengers), its *lease cost* (in Euros), and its *cost per patch* (in Euros), which include, among others, expenses for gasoline. Table 1 shows an overview of the three types of vehicles that are available. New vehicles can be acquired at any time at the expense of their lease cost, after which they are added to the fleet of the public transport company. Hence, the lease cost need to be paid only once per new vehicle, while the cost per patch are paid all the time that a vehicle is moving.

Table 1. Vehicle types[9]

vehicle type	capacity	lease cost	cost per patch
small	12	6000	1
medium	60	12000	1.5
large	150	20000	2

Scenario. A scenario has the length of one full day, i.e. 24 hours. To simulate the fact that passengers with various travel destinations depart from the various bus stops during the day, the notion of *scenario* is introduced. A scenario is defined as a time-stamped sequence of entries of the form shown in Figure 2. Each entry, which can be visualized as a row in a table, represents the new passengers that depart from a particular bus stop at a particular time, along with their travel destinations. For example, the first entry denotes that at Time 0:00, the following passengers depart from bus stop *Amstel*: 5 passengers with destination *Amstelveenweseg*, 1 passenger with destination *Buikslotermeer*, and so on. The second entry shows similar information for Time 0:15 (hence, the temporal granularity of the entries is 15 minutes), and so on until Time 23:45. After this, a similar list of entries is given for the passengers departing from bus stop *Amstelveenseweg*. These entries are provided for all 24 bus stops.

Time 0:00	**Number of people for destination**					
	Amstel	A'weg	B'meer	Centraal	Dam	…
Amstel	-	5	1	8	4	

Time 0:15	**Number of people for destination**					
	Amstel	A'weg	B'meer	Centraal	Dam	…
Amstel	-	6	0	10	5	

Time 0:30	**Number of people for destination**					
	Amstel	A'weg	B'meer	Centraal	Dam	…
Amstel	-	3	1	15	5	

…

Time 23:45	**Number of people for destination**					
	Amstel	A'weg	B'meer	Centraal	Dam	…
Zuid	12	8	4	10	5	

Fig. 2. Example scenario fragment

[9] The cost are not intended to be 100% realistic.

To start working on the assignment, the file *passengers-location_day1.csv*, describing a scenario of 1 full day, is provided via Blackboard[10]. This scenario has been based on real data provided by Google Maps.

It is important to note that passengers are treated as *objects* in this assignment, and not as agents. This means that passengers will not make intelligent decisions themselves as to whether or not they will take a particular bus. Instead, the vehicles are the entities that will make decisions about when to pick up, transport, and drop off passengers, and as long as these actions are physically possible, they will always succeed (so passenger will never 'refuse' to be picked up). Hence, passengers should perhaps rather be seen as 'cargo'.

NetLogo Environment. This assignment should be implemented in the NetLogo Environment. As a starting point, two files have been provided via Blackboard, called Amsterdam.nlogo and agents.nls[11]. The file Amsterdam.nlogo simulates the physical environment, (i.e., the city of Amsterdam), as well as all vehicles and passengers. This file also visualizes the map of Amsterdam (highlighting the 24 bus stops as shown in Figure 1[12]) and the movements of the vehicles. Formally, the environment is represented as a grid with 30x40 cells (*patches* in NetLogo terms), which correspond to areas of approximately 500x500 meters. This file should *not* be modified in any way during the assignment. The file agents.nls is used to specify the decision making behavior of the agents, i.e., the autonomous vehicles. Initially, this file only contains a template, but during the assignment it should be filled with increasingly complex strategies.

A simulation is started by running the code in the file Amsterdam.nlogo. Upon running a simulation, first a scenario is read from the available .csv file(s) (as described in the Scenario section) to have information about when passengers depart from the various bus stops. Next, the initial fleet of the public transport company is created, which by default consists of just 1 small vehicle, located at bus stop *Centraal*. To initialize this agent (e.g., to set initial values for its internal states, like beliefs or goals), the function *init-buses* in the file agents.nls is called. After that, a main loop is executed which continuously calls the function *execute-actions* in the file agents.nls for all agents that are currently in the fleet and realizes the effect of these actions on the environment. Also, if there are any requests to add new vehicles, the main loop creates and initializ-

[10] For the evaluation phase, different scenarios will be used (see Section 3). Also, note that is it possible to run a scenario for more than one day. To this end, one simply has to create additional files, named *passengers-location_day2.csv*, and so on.

[11] The file agents.nls can be opened in NetLogo by first opening the code of the main file and then selecting 'Includes' --> agents.nls.

[12] The color of the logo represents the amount of passengers waiting at a particular bus stop: white means that the bus stop if empty. The darker the shade of grey, the more passengers are waiting.

es these new agents. Finally, it increments the simulation with one 'tick'[13]. This process is repeated until the end time of the simulation is reached.

As a result of this structure, the intelligent decision making strategies for all agents should be implemented within the functions *init-buses* and *execute-actions* in the file agents.nls. To implement these strategies, you will need functionality to interact with the world (e.g., making observations, traveling, and picking up and dropping off passengers) and with other agents (sending and receiving messages). To this end, a number of pre-defined functions are available (see Table 2), which enable interaction between the files agents.nls and Amsterdam.nlogo.

Table 2. Pre-defined agent functions

Observations	
get-distance [bus_stop bus_stop]	return the distance in patches between two bus stops (returns real)
get-passengers-at-stop [bus_stop]	for a given bus stop, return all passengers with their destinations (returns a list of tuples of the form [passenger_id bus_stop])
Actions[14]	
add-bus [vehicle_type]	add a vehicle of a certain type and place it at *Centraal* station
travel-to [bus_stop]	travel to an adjacent bus stop
pick-up-passenger [passenger_id]	pick up a passenger from current bus stop
drop-off-passenger [passenger_id]	drop off a passenger at current bus stop
send-message [bus_id message]	send a message to another bus (to have it added to its inbox)
Information Types	
bus_stop	{0, 1, 2, …, 23} where 0=Amstel, …, 23=Zuid
bus_id[15]	{24, 25, 26, …}
vehicle_type	{1, 2, 3} where 1=small, 2=medium, 3=large
passenger_id	{0, 1, 2, …}
message	*any arbitrary string*

Also, each agent has a number of default local variables (see Table 3), which can be found under agents-own in the file agents.nls. These local variables include static variables (such as the agent's ID or capacity) as well as dynamic variables (such as its inbox with incoming messages and number of passengers). Note that it is **not allowed** to change these variables, as they will be used by the environment as well. However, you are free to add your own local variables here, which you can modify. In addition, please be aware that it is also **not allowed** to use global variables within agents.nls (even though this is technically possible in NetLogo), as this conflicts with the agent paradigm, which assumes that each agent maintains its own, local information sepa-

[13] A tick is a time step in the simulation, which corresponds to 1 minute in real time.

[14] Note: in case an agent tries to execute an action that is physically not possible (e.g., trying to travel to a bus stop that is not adjacent, to pick up a passenger that is not at your current location, or to drop off a passenger that you are not carrying), the function does nothing and a warning message is displayed in the background window.

[15] The bus IDs start counting at 24, because the bus stops are implemented as agents as well.

rately. Hence, in case an agent has a certain belief that it wants to share with other agents, it should explicitly communicate this information (by sending messages) to other agents. The only global variable that can be used is the variable graph, which represents the topology of the network (as a list of lists that indicate for each bus stop to which other bus stops it is connected).

Table 3. Local agent variables (read only)

Static variables	
bus_id	integer that denotes the ID of the agent
bus_type	vehicle type of the agent (1=small, 2=medium, 3=large)
Dynamic variables	
inbox	history of all incoming messages (represented as a list of *[tick sender message]* triples)
bus_passengers	list of passengers that the agent is currently carrying (represented as a list of *[passenger_id bus_stop]* tuples)
previous_stop	ID of the last bus stop the agent visited (if none, return -1)
current_stop	ID of the current bus stop of the agent (if none, return -1)
next_stop	ID of the bus stop the agent is traveling to (if none, return -1)

One important property of the implementation structure introduced above is that the functions *init-buses* and *execute-actions* in the file agents.nls contain code that describes the strategies of all agents (as opposed to a situation where we would have separate files or functions for each individual agent). The consequence of this is that you can use these functions to implement generic strategies (that apply to all agents) as well as more specific strategies (that apply to one or a limited number of agents). Technically, generic strategies are implemented by directly including the desired code within the functions. Specific strategies are implemented by adding the condition *if (bus_id = X)* to the desired code (where *X* is the identifier of the agent).

Evaluation Criteria. The performance of a solution is measured in three different ways. The first, straightforward criterion is the *average travel time* over all passengers within the simulation. Here, the travel time is calculated by counting the number of ticks between the moment that a passenger was 'created' (i.e., shows up at the bus stop according to the scenario as shown in Figure 2) and the moment that the passenger is 'delivered' (i.e., is dropped off at the desired destination). For each passenger that is not delivered by the end of the simulation, its travel time is calculated by taking the time the passenger was already traveling plus a penalty of 180 ticks (= three hours).

A second criterion to measure the performance of a solution is the *total amount of money spent* by the public transport company. The total amount of money starts at 0, and is increased every time money is spent[16]. There are two types of actions that cost

[16] The is no limit to the amount of money that can be spent.

money, namely *adding new vehicles* and *traveling*. The cost for these actions depend on the type of vehicle, and are defined by the *lease cost* and *cost per patch*, respectively (see Table 1). Note that money can only be spent, not earned (the entire notion of earning money is left outside the scope of this assignment).

The third criterion to measure the performance of a solution is by counting the *total number of messages sent* during the simulation. The lower the number of messages that are sent, the better the solution.

The status of all three evaluation criteria can be observed directly during a simulation via three separate graphs in the interface[17].

3 Schedule and Grading

A rough schedule for the assignment is given in the table below.

Table 4. Schedule

Week	Content
1	Install and explore NetLogo, and make the 'vacuum cleaner assignment' that is available on Blackboard.
2	Implement a first version of the multi-agent transportation system in which vehicles pick up passengers, travel to the final destination and drop off passengers according to a fixed schedule (no intelligence / coordination needed).
3	Implement a version in which vehicles exchange messages with each other according to a self-defined protocol/ontology (e.g., to inform them about the passengers they are carrying).
4	Implement a version in which a simple form of coordination takes place (e.g., not all vehicles go to the same bus stop).
5	Implement a version in which vehicles make group decisions and/or form coalitions (e.g., vote about whether or not to add new vehicles).
6	Implement a version in which vehicles use at least one strategy for competition (e.g., to negotiate or bid for passengers).
7	Implement final version of the multi-agent system.

As indicated in this schedule, the first week is meant to get familiar with the NetLogo environment. In this week you will make a tutorial as well as simple assignment about a smart vacuum cleaner. This assignment will not be graded explicitly (see below for the exact formula to calculate the grade). In the weeks that follow, you will gradually improve your multi-agent transportation system. In week 2-6, you will add new functionality to your system in a step-wise manner. Each intermediate version needs to be

[17] Note that this number might increase a lot in the last time step of the simulation, as this is the moment when the penalty of 180 ticks per remaining passenger is added.

handed in (by submitting your agents.nls file via Blackboard), but will also not be graded explicitly. In week 7, you will hand in the *final version* of your implementation as well as a *report* about your system. These two elements will be graded in detail. Also, in week 7 there will be a closing session in the form of a 'competition' in which the solutions of all groups are compared to see which system has the best performance. For this evaluation, similar (but *not* identical) scenarios will be used as the one provided at the start of the assignment!

For each group, the final grade for the assignment is calculated as follows:

- During week 1-6, each intermediate solution will be graded with a pass (= *0.5 points*) or a fail (= *0 points*)
- Hence, when all weekly requirements have been met, you already have *3 points*
- For the report you can receive a maximum of *3 points*
- For the quality of the solution you can receive a maximum of *4 points*.
- Finally, up to *1 bonus point* is awarded to the three systems with the best performance

The *report* should contain a conceptual description of the system, including an explanation of the various agent concepts (communication, coordination, group decisions, negotiation, …) that have been used. The report should not exceed 10 pages. It will be evaluated based on readability and completeness. Readability refers to correct use of the English language, writing understandable text and making use of a proper scientific style. For completeness, it should become clear what you have done, why, and what the results are. In other words, can the reader understand what you have done.

- *Readability*: Is the spelling correct? Are sentences understandable? Has a scientific writing style been applied? Are figures, tables and articles properly referenced?
- *Completeness*: Have all design choices been motivated? Can a reader understand from the documentation how the system works?

The *quality of the solution* refers to the way in which the various agent concepts and strategies are designed and implemented. The more ambitious the strategies for communication, coordination, negotiation, etc., the higher the grade. Also, it is important that the implementation of the strategies is correct and consistent with the description in the report.

Bonus Points. To decide which groups obtain bonus points based on the *performance of the system*, the following procedure is taken. First, all solutions are ranked based on their *average travel time*, *total amount of money spent*, and *total number of messages sent* (as defined in Section 2). Hence, three separate rankings are created. Next, for each group, the average position over the three ranking is calculated, where the position of the first ranking (for *average travel time*) counts double. So for example, if a

group ends up at position 10, 40 and 20, respectively, the average position of this group is calculated as (10+10+40+20)/4 = 20. Based on these average positions, the final ranking is determined. The group that ends up first in that final ranking is the winner, and receives *1 bonus point*. The second and third group receive *0.75 and 0.5 bonus points*, respectively.