Comparing the Performance of Dynamic and Semi-flexible Demand-Responsive Transport Models under Varying Demand Scenarios

van Lent, Goos M.

Award date:
2023

Awarding institution:
Universitat Politècnica de Catalunya

Link to publication

Escola de Camins
Escola Tècnica Superior d'Enginyeria de Camins, Canals i Ports
UPC BARCELONATECH

Comparing the Performance of Dynamic and Semi-flexible Demand-Responsive Transport Models under Varying Demand Scenarios

MASTER FINAL THESIS

Final Thesis developed by:
Van Lent, Goos Milan

Directed by:
Badia Rodríguez, Hugo

Master in:
Sustainable Urban Mobility Transitions

Barcelona, 28th of June, 2023

Department of Civil Engineering

Comparing the Performance of Dynamic and Semi-flexible Demand-Responsive Transport Models under Varying Demand Scenarios

Sustainable Urban Mobility Transitions – Master's Thesis

Goos van Lent, Barcelona, June 28th, 2023

Supervisor: Hugo Badia Rodríguez

Universitat Politècnica de Catalunya

Abstract: As authorities worldwide are cutting public transportation funds, finding cost-effective means to provide public transport in underserved areas has become ever important. One of such means is Demand-Responsive Transport (DRT), a type of transport where supply is adjusted based on passenger demand. Although DRT schemes have existed since the 1960s, the rise of portable electronic devices like smartphones have enabled DRT schemes which can instantly respond to new passenger requests. These systems are classified as dynamic DRT, and while they can provide more tailored transport to passengers, these systems often become overwhelmed in higher-demand scenarios. To that end, semi-flexible DRT systems have been emerging, where only part of a vehicle trip can be adapted according to demand. In this paper, both dynamic and semi-flexible DRT models are compared with each other through a programme which simulates DRT model behaviour based on passenger requests. Using different stop density and demand level scenarios, it is assessed how well both models perform under these varying scenarios. Through the construction of relevant metrics, user and operator benefit are assessed, and recommendations are made which DRT model can best serve as a blueprint for affordable public transport in different types of areas. From the simulation results, it is determined that dynamic DRT systems are only feasible in areas with very low travel demand (between 4 and 8 trip requests per hour), and with a limited amount of bus stops where passengers can be picked up or dropped off. For higher demand levels, semi-flexible DRT systems perform better regarding pickup and drop-off time reliability and regarding the minimization of excess travel time for passengers.

Keywords: Demand Responsive Transport, DRT, semi-flexible DRT, dynamic DRT, medium-demand public transport, DRT simulation, delay, detours, vehicle load

Table of Contents

Introduction

As national and local governments alike are aiming to reduce car use among its population, multiple ways are being sought to reduce overall car dependency. The promotion of alternative transportation modes like cycling and public transport often make sense in an urban setting. However, in rural settings, the introduction or enhancement of public transport is often too costly for the level of demand it can serve. Therefore, in recent years, local authorities have sought to find a more cost-effective way of serving areas with low transport demand. Demand-Responsive Transport (DRT) is a form of public transport which has been increasingly utilized in areas where regular public transport is not deemed cost-effective. DRT uses, in contrast to regular public transport, flexible routing, flexible time tabling, or both, in order to be able to either serve more transportation demand for similar costs, or serve similar transportation demand for fewer costs.

Although DRT has been a concept for several decades now, the rise of information technologies has made DRT more accessible to use for end-users. While 20th-century DRT systems made use of calling services in order for end-users to be able to make a reservation, nowadays users can access information and book a trip through mobile applications and mobile browsers. This has enabled the development of dynamic DRT systems where users can book their trip on a bus in real-time, making the bus modify its route in real-time in order to fulfil the new request. While this has greatly reduced the barrier for DRT use and increased flexibility for users, it has also made DRT services less predictable and therefore less reliable for users. Also, it has been noted that dynamic DRT systems might not be able to serve as much demand as static DRT systems, where the vehicle itinerary is always pre-determined. This has given rise to 'semi-flexible' DRT systems, which still make use of a predefined route, but allow certain deviations from this central 'trunk' based on passenger demand.

Semi-flexible DRT systems have different definitions, varying from a limited number of optional stops outside of a fixed bus route, up until a complete set of optional stops, with only a departure stop, a terminal stop and a stop sequence defined. In this thesis, the performance of this latter definition of semi-flexible DRT systems will be compared to the performance of dynamic DRT systems.

In contrast to literature assessing the differences between regular public transport and dynamic DRT systems, the performance of dynamic DRT systems against semi-flexible DRT systems has not been studied extensively. This thesis will therefore provide new insights to what degree external factors influence DRT system performance. Under varying demand levels and pickup and drop-off node densities, it is assessed how both DRT systems perform. Through the construction of two simulations, one for each DRT system, it will be studied how both models process different levels of demand and different pickup and drop-off spot densities. Both models will be compared on reliability, passenger aggregation and passenger time spent on the vehicle. From these insights, it can be assessed under which demand levels and population densities each model performs best.

Definition of Concepts

DARP: Dial-A-Ride Problem

DRT: Demand-responsive Transport

Expected trip request rate: the expected value of the average number of trip requests per hour in a simulation. For example, when this rate is 8 trip requests per hour, it is expected that on average, 8 trip requests per hour are made in the simulation.

Dynamic DRT: model of DRT where a vehicle can change its itinerary freely based on real-time requests.

Model: system of DRT used. In this paper, a distinction is made between the dynamic and semi-flexible DRT model.

Node density: the number of pickup and drop-off nodes in the DRT area.

RPH: Requests Per Hour. In the simulations, this indicates the expected number of random requests to be generated per hour

Scenario: instance of DRT under certain parameters. These are the node density and the expected trip request rate.

Semi-flexible DRT: model of DRT where a vehicle needs to visit nodes in a sequential order, also known as the trunk route. This model also forces the vehicle to operate in separate itineraries (headways).

Simulation: one calculation of a specific DRT scenario, returning a traversed route and a list of served trip requests.

Trip request: a request in a simulation with a pickup node, drop-off node, time of pickup and number of passengers.

Trunk route: the sequence of nodes to visit in the semi-flexible DRT model.

TSP: Traveling Salesman Problem.

Vehicle load: number of passengers in a vehicle at a specific time.

VRP: Vehicle Routing Problem.

Literature Review

In recent years, there has been an increase in research on varying DRT models. This section provides an overview of the state of the art in research on the viability of different DRT schemes.

Demand-Responsive Transport

Demand-responsive Transport, commonly abbreviated to DRT, is the concept of transport services which adapt their routing and/or scheduling based on passenger demand. These transport services can range from mobility services at events to tailored medical transportation to full-fledged public transport systems. The rule of thumb is, however, that overall demand should be low (Papanikolaou & Basbas, 2020), in order to not overwhelm the system with requests, which it cannot all fulfil at once. Therefore, DRT systems are most commonly found as rural public transport or as transport for people with reduced mobility.

History of DRT

Transport tailored to fulfilling individual travel needs outside of mass transit dates back to the 20th century. With national and local governments recognizing the needs of the elderly and physically impaired, it became apparent that a tailored mobility solution needed to be created for this specific target audience.

As far as literature shows, research on DRT algorithmics and systems began in the 1960s with the Harvard CARS project (Wilson et al., 1969). The first real strides towards DRT systems began in North America and the UK in the 1970s (Ho et al., 2018). Under the name of 'paratransit', or 'dial-a-ride' certain cities and regions began offering limited services to people for whom regular public transport was not a feasible option (Lave & Mathias, 2000). However, this was a very specific service, viewed completely separately from existing public transport, offering limited service, and for which only a limited number of people was eligible. After the Americans with Disabilities Act of 1990 (ADA), it was made mandatory to offer unconstrained complementary paratransit service, after which paratransit became widespread in the US (Lave & Mathias, 2000).

DRT also developed in Europe during the 1970s (Coutinho et al., 2020). These systems were often inspired by their North American counterparts. Over time, services also began to spread to areas where regular public transport was not financially viable, expanding the target audience towards anyone who lives in sparsely-populated areas.

However, DRT remained focused on the elderly and physically impaired throughout the 1990s and 2000s. Palmer et al. (2004) define DRT as "means by which 'comparable transportation services' are provided to mobility impaired individuals".

DRT as public transport in sparsely populated areas

Even though the focus of DRT has mostly been on serving mobility-impaired people, it has increasingly been used as a substitute for, or complement to, public transport (Ho et al., 2018). Especially in times of austerity measures by governments and general unwillingness to subsidize low-demand public transport (Gomes et al., 2015), alternative solutions have to be sought to maintain an acceptable level of public transport in these areas.

DRT services have been trialled as a public transport substitute in several cases. A case in a rural area near Amsterdam, the Netherlands (Coutinho et al., 2020) showcase that the average distance driven per passenger decreased, but that ridership overall also decreased.

In order to decrease operational costs of DRT systems, models have been proposed where DRT acts as a feeder for the mobility impaired to fixed-route bus services (Posada et al., 2017). However, this comes with a big cost to users: changing twice between transport modes for one trip is inconvenient, especially for the physically impaired.

The Dial-A-Ride Problem

With the dispersion of DRT systems, the so-called 'dial-a-ride problem' (DARP) arose with it. Cordeau & Laporte (2007) state that:

"The Dial-a-Ride Problem (DARP) consists of designing vehicle routes and schedules for n *users who specify pickup and delivery requests between origins and destinations."*

The DARP is a variant of the well-known Traveling Salesman Problem (TSP), which aims to obtain the shortest route which visits all nodes in an area. Because the DARP is an extension of the TSP, several constraints are set in order to resemble a real-life dial-a-ride system. These include:

-Vehicle capacity (Parragh et al., 2012), (Wong & Bell, 2006)
-Route duration (Wong & Bell, 2006), (Jain & Van Hentenryck, 2011)
-Set routing starting and ending points (Ho et al., 2018)
-Driver working hours and lunch breaks (Parragh et al., 2012)
-Time window constraints (Xiang et al., 2006), (Jain & Van Hentenryck, 2011)
-Maximum waiting time constraints (Hunsaker & Savelsbergh, 2002)

Under (part of) these constraints, research has been done on the DARP for several decades. Wong & Bell (2006) state that solution methods for the DARP can be broadly separated into exact methods and heuristic algorithms. Exact methods aim to make a mathematical formulation of a DARP which can then algorithmically be found an optimal solution for. However, this is often not possible in practice due to excessive computing time. Heuristic methods iteratively try to find a good-enough solution through an algorithm which generates an acceptable solution within an acceptable amount of time. For practical implementations, heuristic methods like Integer Linear Programming (ILP) or simulations often can provide an insight in the performance of systems.

Static versus dynamic dial-a-ride problems

DARPs, and to an extension DRT models, can generally be categorized into static and dynamic systems (Cordeau et al., 2007). In static systems, all trip requests are known beforehand, while in dynamic systems, users are able to request a trip at any time, making it possible for vehicle routes to be adapted even when the trip has already started (Cordeau et al., 2007). These two different systems differ significantly in terms of passenger capacity, reliability, and flexibility. Depending on the conditions of a certain service area, and the priorities of the operator and local authorities, decisions are made on which kind of DRT system to use.

Cordeau et al. (2007) identify that most transport on demand problems have the conflicting objectives of maximizing the number of requests served, minimizing operational costs and minimizing user inconvenience. Operators and governments need to account for all these factors, making the choice of which type of public transport to use very case-specific.

Recent research has suggested that a deviation from fully dynamic, DRT models might be beneficial for system reliability and affordability. (Sörensen et al., 2021) identify that, even in very rural areas, DRT trips tend to aggregate around central axes, or trunks. Therefore, it is interesting to look at DRT models which exist between static and dynamic models: semi-flexible models.

Semi-flexible DRT

The traditional setup of DRT systems is fully flexible, meaning that vehicles can take any possible route between the given pick-up and drop-off points, whether these are door-to-door services or fixed bus stops. However, this might increase traveling time significantly, as new users can request a ride at any moment from any available point. In order to increase reliability of pick-up time and service availability, additional constraints, on top of the usual constraints, can be introduced. These are:

-A fixed 'trunk route', which connects all the pickup and drop-off nodes in the system. Each vehicle itinerary is based on this route; however, it will only visit a node if there is demand for it. This means that the trunk is essentially an order of nodes to be visited.
-A fixed headway: because the vehicle traverses this trunk, the service is divided in itineraries. For example, a vehicle can have two itineraries per hour, one in each direction. This makes the headway one hour in each direction.

By enforcing these constraints, passenger trip requests will automatically be assigned to a certain itinerary. It is possible for passengers to make requests while the vehicle has already started its itinerary; however, the passenger can only travel in the same itinerary if the vehicle has not passed the pickup point of the passenger yet.

Both constraints reduce the flexibility of the system, but increase reliability, and potentially financial feasibility of the system. Therefore, this type of DRT will be called 'semi-flexible': vehicles can operate flexible routes and times within the bounds of the routing graph and the timeslot given.

Current literature on semi-flexible DRT systems

In existing literature, semi-flexible DRT is most often defined as a system where a bus in principle follows a fixed route, but is allowed to make certain deviations on certain sections based on demand (Koffman, 2004; Li et al., 2023; Mishra & Mehran, 2023). This is different than the version used in this paper, since the version in this paper does not contain mandatory routes; just a sequence of stops which has to be traversed if they have demand. However, both systems are still comparable, since there still is a mandatory starting and end node, of which the path is altered based on demand.
Existing literature suggests that semi-flexible DRT, as defined in most literature, is cheaper to operate than dynamic DRT (Li et al., 2023; Mishra & Mehran, 2023), and can act as a cost-effective means of supplying high-quality public transport, often in combination with fixed public transport or bicycle sharing systems (Bruzzone et al., 2020). However, literature about the exact benefits of semi-flexible DRT is sparsely available, especially for users themselves. Direct comparisons between user costs and benefits regarding both DRT systems could not be found in literature.
In order to determine which type of public transport fits best in an area, or more specifically, which type of DRT to choose, it is important to have knowledge about both operator and user

costs (Papanikolaou et al., 2017), of which especially the latter is often unknown. Therefore, a direct comparison between dynamic and semi-flexible DRT can provide new insights in order for local authorities and public transport operators to make more informed decisions about which type of DRT to use.

Linear programming approach for DRT system modelling

Currently, DRT model performance is often assessed through linear programming (Alonso-Mora et al., 2017; Li et al., 2021, 2023; Mehran et al., 2020; Mishra & Mehran, 2020). Linear programming entails the maximization or minimization of a certain value based on predefined constraints. In DRT modelling, this is especially useful, since the minimization of either operator costs or user costs are mostly sought after, as well as the maximization of benefits.

Operator and/or user costs are aimed to be minimized according to a set of constraints. These constraints should represent real-life conditions as much as is feasible. For example, constraints which are often implemented are constraints on the operator side include time windows (Mishra & Mehran, 2020), vehicle fleet size and vehicle capacities (Cordeau & Laporte, 2007), (Mishra & Mehran, 2020), headways, mandatory stops (Li et al., 2023) and depot locations (Li et al., 2021). On the other hand, constraints on the user side include preferred pickup times (Li et al., 2021), maximum waiting time and delay constraints (Alonso-Mora et al., 2017). These constraints all need to be satisfied while minimizing costs defined by total distances driven (Cordeau & Laporte, 2007), total operational hours (Mishra & Mehran, 2020), and the sum of all delays (Alonso-Mora et al., 2017).

While constraints can be fairly easily defined in linear programming, actually calculating certain costs, especially vehicle distances driven, is not as straightforward. For this, many different algorithms exist which can find a good routing scheme in an acceptable amount of runtime that serves a certain number of locations. Such algorithms often employ an initial heuristic, to construct an initial feasible route for an acceptable cost, and then iteratively improves on it with a metaheuristic. VRP algorithms often include construction heuristics (Konstantakopoulos et al., 2022). Construction heuristics construct a feasible path by finding the lowest cost arcs (usually the shortest path) for small routes, and connecting them together to create overall low-cost routes (Hoos & Stützle, 2005). These heuristics often employ local search heuristics as a metaheuristic to find improvements in local (small) parts of the route.

In order to define operator and user costs between dynamic DRT and semi-flexible DRT, linear programming will also be used to define the theoretical model and constraints of both DRT models. Through a Python program, both DRT models will be simulated. Using the Google OR-tools library as a route searching tool for dynamic DRT simulation, and as a tool to determine the shortest trunk distances in semi-flexible DRT, data passenger trip handling and vehicle routing in both models can be obtained and used for constructing metrics.

Research Approach

Two different DRT models are analysed in this paper: dynamic and semi-flexible DRT models. They are simulated in a virtual area of 1 by 1, with pickup and drop-off nodes spread randomly in the area. In this section, both models will be described in detail.

Dynamic DRT

Dynamic DRT is defined as the model in which a vehicle tries to serve all the requests which are currently submitted to the system. In other words, it always follows a route calculated to serve all pending requests in the shortest amount of time possible. This has the consequence that the complete route can change once one new request has entered the system. Due to this principle, each time a new request enters the system, a new route has to be calculated for the vehicle that satisfies two constraints:

- To satisfy each trip request, a pickup node should always come before its drop-off node
- The route to serve all requests should be as short as possible

After a new route has been calculated, the bus follows the new route until a new request comes in. At that moment, the route is recalculated to serve all pending and unserved requests, as well as the new request, and the process repeats.

For example, in figure 1, a simulation is started with 10 initial requests. For these requests, a route is calculated. The bus starts at node 20.

In figure 2, the simulation at time t=1100 is shown. Three requests have been completed, three are underway and four are yet to be started. At this time, a new request enters the system, with pickup node 19 and drop-off node 17. A new route is calculated based on this new request, as well as the nodes that already had to be visited based on requests that are underway and requests that still need to be started. This results in the route in figure 3. his route will then be followed until a new request comes in, after which the process is repeated.

PickUp	DropOff	People
19	21	1
20	19	1
2	13	1
12	1	1
9	10	1
12	14	1
8	21	1
21	22	1
4	12	1
23	5	1

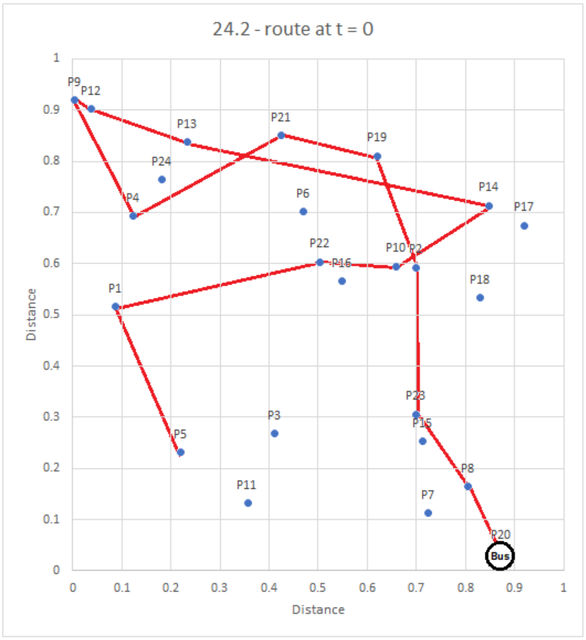
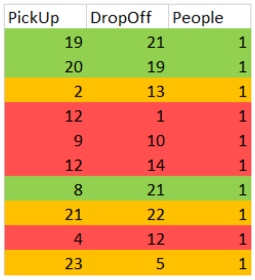
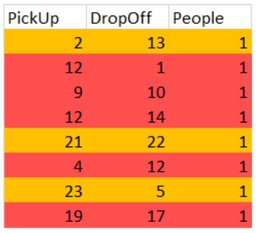


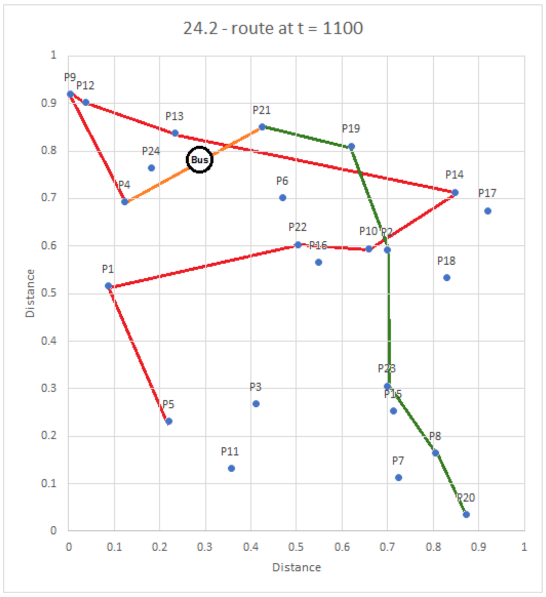
Figure 1: Example of the initial route and requests of a dynamic DRT simulation.

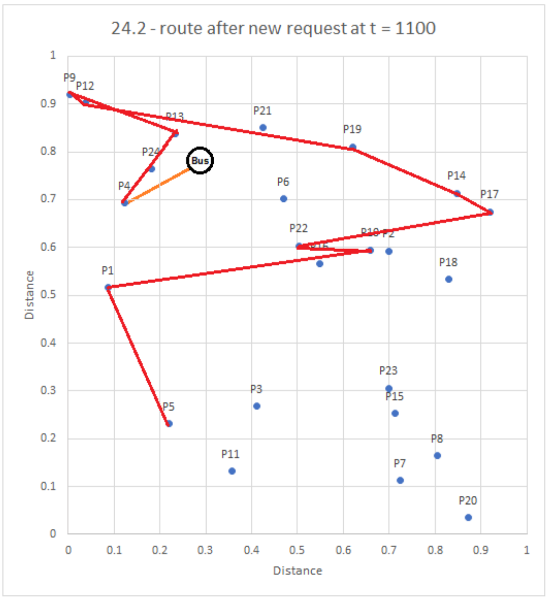


PickUp	DropOff	People
19	21	1
20	19	1
2	13	1
12	1	1
9	10	1
12	14	1
8	21	1
21	22	1
4	12	1
23	5	1



PickUp	DropOff	People
2	13	1
12	1	1
9	10	1
12	14	1
21	22	1
4	12	1
23	5	1
19	17	1





Figures 2 (left) and 3 (right): Served (green), pending (orange) and yet to be served requests (red) at the moment a new trip request enters the system. The bus has completed the green segments, is traversing the orange segment and has yet to traverse the red segments. The route to be travelled (red) is altered due to the new request entering the system.

This processing and entering of trip requests can in theory go on endlessly. For modelling purposes, a 6-hour window is simulated.

Semi-flexible DRT

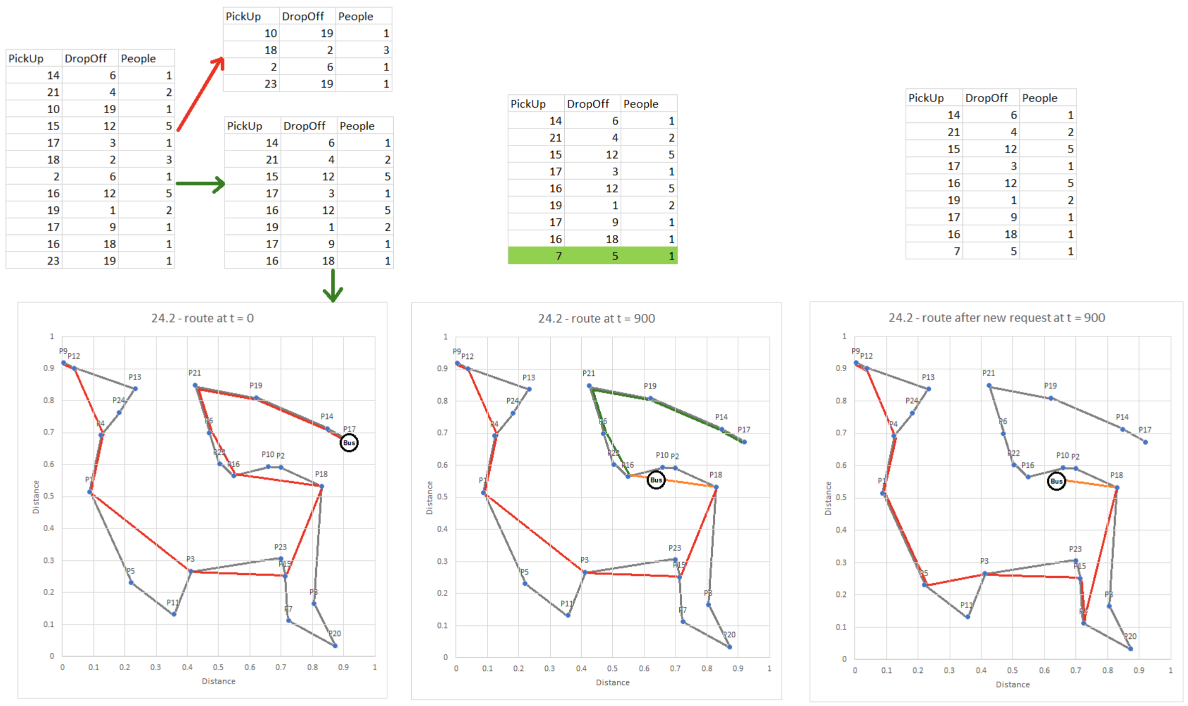
The semi-flexible DRT model makes use of the same virtual service areas as dynamic DRT, with the same nodes, but also has a trunk route defined. This route has been obtained by running a TSP over the field of nodes, and therefore is a short distance to serve all nodes in the field. The vehicle in the system travels between the first and last node in this system, and vice versa, in sequential itineraries. The vehicle serves each trip request in the order of the trunk sequence. If a trip request is in the reverse order compared to the trunk sequence, it will only be satisfied in the next itinerary.

Given these rules, three constraints apply to semi-flexible DRT:

- To satisfy each trip request, a pickup node should always come before its drop-off node
- To adhere to the trunk node sequence, a pickup node should always become before its drop-off node in the sequence
- An itinerary should always run from the first node in the trunk sequence to the last.

The second constraint has the result that when a new trip request is generated, but the vehicle has already passed its pickup node in the running itinerary, the trip request can only be satisfied in the second itinerary after it.

An example of request handling in the semi-flexible DRT model can be found in figures 3, 4 and 5. In figure 3, an initial set of requests enters the system. Only the requests which adhere to the trunk sequence are processed into the route. After t = 900, the system receives a new request,



PickUp	DropOff	People
14	6	1
21	4	2
10	19	1
15	12	5
17	3	1
18	2	3
2	6	1
16	12	5
19	1	2
17	9	1
16	18	1
23	19	1

PickUp	DropOff	People
10	19	1
18	2	3
2	6	1
23	19	1

PickUp	DropOff	People
14	6	1
21	4	2
15	12	5
17	3	1
16	12	5
19	1	2
17	9	1
16	18	1

PickUp	DropOff	People
14	6	1
21	4	2
15	12	5
17	3	1
16	12	5
19	1	2
17	9	1
16	18	1
7	5	1

PickUp	DropOff	People
14	6	1
21	4	2
15	12	5
17	3	1
16	12	5
19	1	2
17	9	1
16	18	1
7	5	1

Figures 3, 4 and 5: Example of the initiation of an iteration of a semi-flexible DRT simulation.

from node 7 to node 5. Since the bus is still located before node 7, and node 5 comes after node 7 in the trunk sequence, it is able to be processed in this itinerary. The route is adapted to serve the new request as well.

After the route has been completed, the vehicle waits at node 9 (the last node of the completed itinerary and the first node of the new itinerary) until the predefined start time of the new itinerary. Then, the unserved request from the previous itinerary can be processed, as well as any newly entered requests that adhere to the trunk sequence.

Both simulations will result in outputs and used to construct performance metrics, which is explained in the methodology.

Methodology

In order to make the comparison between the dynamic and semi-flexible DRT systems, two separate simulations were made in Python. These simulations aim to simulate the generation of trips requests in a theoretical area with a given number of nodes, where passengers can be picked up and dropped off. Through periodically inserting new trip requests, vehicle trip itineraries are continuously recalculated, constantly giving updated pickup- and drop-off-times for each trip request. When a trip request is fulfilled, i.e., when the vehicle arrives at its drop-off-point, the definitive drop-off time is known, resulting in a time discrepancy between the initially given pickup and drop-off times and their actually realized pickup and drop-off times.

In this section, the methodology for obtaining these results will be described in detail.

The Concept

Both models will be simulated in a theoretical area of 1 by 1 distance, with a given number of nodes (representing the points where the vehicle can pick up and drop off passengers) randomly distributed within it. In each simulation, it is simulated how passenger requests over the course of a set amount of time are handled by each DRT model. To simulate these requests, each simulation starts with a given number of requests (to simulate requests made before the service day starts), and afterwards processes a list of randomly created requests, created at random times at a specific spawn rate. After a new trip request has entered the system, routes are recalculated, and pickup and drop-off times changed.

In order to account for randomness, multiple areas with a random distribution of nodes are created. The location of the nodes is defined as a distance matrix in the Python program, containing the distance between each pair of nodes in the system.

In order to simplify simulations and comparisons, it is assumed that one vehicle serves each area. The service time window is 6 hours; in semi-flexible DRT this is approximated as 12 itineraries of half an hour each.

The Simulation

The simulations simulate two different DRT models through assessing the given input, generating random trip requests and returning the completed travel itinerary, together with the exact time each trip request was entered, served and completed. **Trip requests** are defined as a request for a trip from a node A to a node B, for a given number of passengers, on a given timestamp for when the request was made. These trip requests are made in a theoretical square **area** of distance 1 by 1. This area contains a certain number of **nodes**, which are randomly distributed over the area. This area is served by a **vehicle** which aims to service all trip requests.

In order for the simulation to be able to be set up, it requires a set of inputs, which are defined below.

Input

Both simulations require several inputs. First of all, it takes a distance matrix, which defines all the distances between each node pair. Since Google OR-tools requires a node to be assigned as a ‘depot’, a theoretical node ‘0’ is added to this distance matrix, with a distance of 0 to each node. In this way, a route can always start from any node, since any node is distance 0 from the theoretical depot. A distance matrix, in this example of only 5 nodes, is represented as follows:

```
[[0, 0, 0, 0, 0, 0],
[0, 0, 4, 3, 7, 2],
[0, 4, 0, 5, 4 ,7],
[0, 3, 5, 0, 1, 3],
[0, 7, 4, 1, 0, 8],
[0, 2, 7, 3, 8, 0]]
```

All node distributions that are used in the simulations can be found in appendix H.

Besides the distance matrix, each simulation has certain parameters that can be altered: these are:

- The expected number of requests per hour
- The amount of initial random requests that need to be created before the simulation starts
- The amount of service hours; i.e., the length of the simulation. In semi-flexible DRT this is defined as the duration of one headway multiplied by the defined amount of itineraries to be simulated.
- The maximum number of passengers per trip request
- The chance distribution of the number of people per trip request: if the maximum amount of people per request is 5, a [75, 15, 5, 3, 2] gives a 75% chance of a request containing 1 passenger, a 15% chance of a request containing 2 passengers, and so forth.

The simulations in this paper use a set number of 10 random trip requests to be defined before each simulation. The simulation duration is always 6 hours, or 12 times 30 minutes for semi-flexible DRT simulations. The maximum number of passengers per trip request is always 5.

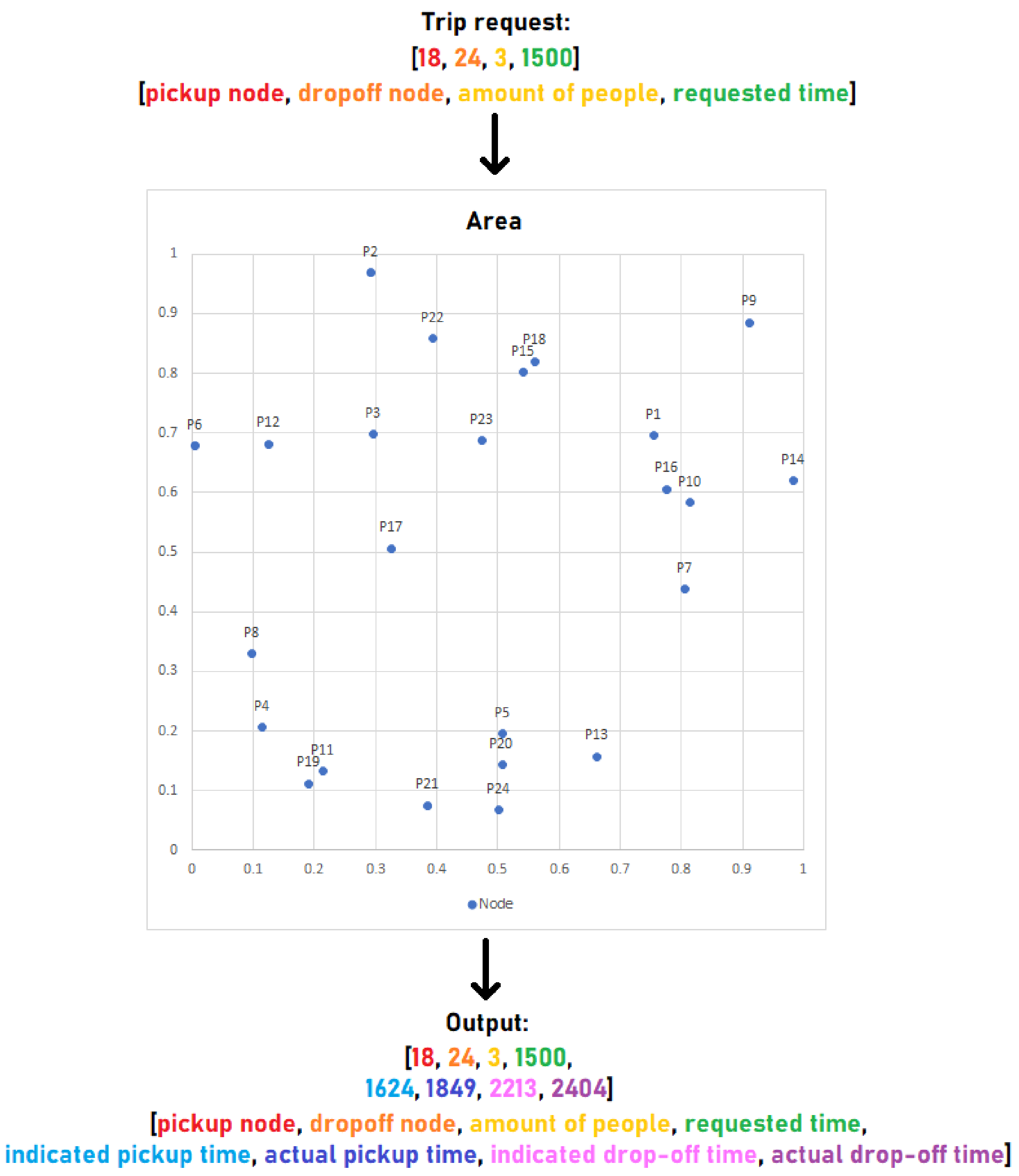
Output

After the simulation has been run, each trip request has the following attributes:

Pickup node	The node where the vehicle picks up the passenger(s).
Drop-off node	The node where the vehicle drops off the passenger(s).
Number of people	The number of passengers in the request.
Requested time	The time on which the request was made. For modelling purposes, this is also the requested pick-up time.
Indicated pickup time	The initial time given by the simulator for when the passenger(s) would be picked up
Actual pickup time	The actual time the vehicle picked up the passengers in the simulation. This can be different from the indicated pickup time, since other requests might be process between the entry of this particular request and the time the vehicle arrives.
Indicated drop-off time	The initial time given by the simulator for when the passenger(s) would be dropped off
Actual drop-off time	The actual time the vehicle picked up the passengers in the simulation. This can be different from the indicated pickup time, since other requests might be process between the entry of this particular request and the time the vehicle arrives, as well as during the time the passenger(s) are present on the vehicle.

Table 1: Output of processed trip requests in both DRT simulations.

In figure 6, it is visualised how each trip request are represented before and after the simulation:



Trip request:
[18, 24, 3, 1500]
[pickup node, dropoff node, amount of people, requested time]
Area
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
P17
P18
P19
P20
P21
P22
P23
P24
Node
Output:
[18, 24, 3, 1500,
1624, 1849, 2213, 2404]
[pickup node, dropoff node, amount of people, requested time,
indicated pickup time, actual pickup time, indicated drop-off time, actual drop-off time]

Figure 6: Visualisation of the processing of a request.

Each trip request consists of the pickup node, the drop-off node, the number of passengers and the requested pickup time before the simulation. Once the whole simulation has been run, each trip request also contains the information of when the request has been handled. From this information, metrics are constructed in a later stage of the thesis to be analysed.

Besides a list of completed trip requests, a complete route for the DRT vehicle is also generated. This consists of a list of each visited node, in chronological order. This list contains three other indicators: the total travel time up until that point, the load of the vehicle after unloading and picking up passengers at that node, and the total number of served passengers. A simple example is the following:

Pickup node	**Drop-off node**	**People**	**Requested time**
3	7	1	0
7	9	1	0
3	7	1	0
9	6	1	0
3	7	1	0
6	5	1	0
3	2	1	0
6	1	3	0
5	12	2	0
2	11	2	0

Node	**Distance**	**Load**	**Passengers**
9	0	1	1
6	332	4	5
2	650	6	7
5	865	7	9
3	1264	11	13
7	1321	9	14
12	1483	7	14
1	1669	4	14
11	1761	2	14
2	2315	1	14
9	2555	0	14

Tables 2 and 3: Example of a simple trip request input and route output.

In this example, 10 requests are made before the start of the service, hence their request time of 0. In the simulation, the quickest path to serve all these requests is calculated, as well as the load of the vehicle.

Two different simulations: dynamic versus semi-flexible DRT

In order to compare dynamic DRT with semi-flexible DRT, two different simulations are run under identical input parameters. These simulations are similar to each other, but contain important differences.

Simulation of dynamic DRT

Dynamic DRT is simulated through a continuous simulation: given a set of initial requests, the program starts with the processing of these requests and computing a first route based on these requests. Afterwards, it iteratively accepts new requests one-for-one, based on their requested time. This causes the route to be updated every time a new request has been processed. In the meantime, the vehicle keeps visiting nodes, completing requests in the meantime, which are then marked as ‘served requests’. After running the simulation for a certain time, an extensive list of ‘served requests’ exists, each with their own time marks and delays.

The simulation software works in the following way:

- Input parameters are chosen
- The program starts
- The program generates the first list of initial requests
- These initial requests are processed to generate the initial route
- New requests are iteratively inserted according to their requested time, generating a new route each time
- After a set number of hours, defined in the input parameters, the program stops and returns the full route and fulfilled requests.

The conceptual scheme for the simulator can be seen in figure 7.

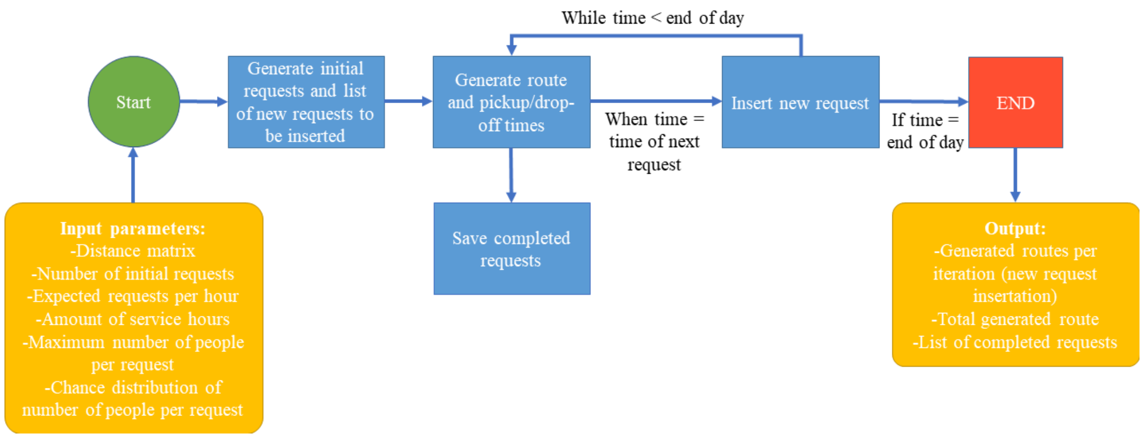

While time < end of day
Start
Generate initial requests and list of new requests to be inserted
Generate route and pickup/drop-off times
When time = time of next request
Insert new request
If time = end of day
END
Save completed requests
Input parameters:
-Distance matrix
-Number of initial requests
-Expected requests per hour
-Amount of service hours
-Maximum number of people per request
-Chance distribution of number of people per request
Output:
-Generated routes per iteration (new request insertation)
-Total generated route
-List of completed requests

Figure 7: The conceptual scheme of the dynamic DRT simulation

Technical summary of the dynamic DRT simulation

The dynamic DRT simulation has been made in Python, using the Google OR-Tools software suite (Google for Developers, n.d.). Google OR-Tools is callable in Python through a wrapper function, and is capable of solving TSPs and VRPs. The 'Vehicle Routing with Pickups and Deliveries' example code (Google for Developers, 2023b) from the Google OR-Tools website has been used as a template for the simulation code, since the dynamic DRT simulation also needs to process pickup and delivery requests.

Google OR-Tools in Python can solve VRPs with pickups and deliveries through processing a distance matrix of the distances between each pair of nodes, and a list of pickup and drop-off requests. Using a predefined search heuristic, the OR-Tools solver attempts to find an as short as possible route which serves all nodes. Through defining a constraint that for each pickup and drop-off node pair, each pickup node should be visited before the drop-off node, it is ensured that each pickup-and-delivery pair is served.

For the dynamic DRT simulation, the original OR-Tools example code has been changed to be able to generate new requests, insert new requests iteratively, to update the route after each newly entered request and to keep track of served requests, its pickup and drop-off times, and the load of the vehicle at all times. Because OR-Tools does not natively support multiple visits of the same node, dummy nodes are created for each node with the same location. Each node location therefore contains four 'pickup' dummy nodes and four 'drop-off dummy nodes: in the final output, these are represented simply by their common 'real' node.

The used Python code can be found in appendix I, with a pseudocode representation presented below.

```
#Input variables:

distanceMatrix = Matrix with distances between all node pairs
numberOfRequests = Number of initial requests at the start of the
program.
requestsPerHour = Expected number of requests to be generated during
the program, per simulated hour.
amountOfRunningHours = Number of hours the simulation should run.
maxPeoplePerRequest = Maximum amount of people per request. The
analysis uses 5.
requestWeights = Chance distribution of amount of people per requests.
The analysis uses [75, 15, 5, 3, 2] means a 75% chance that a random
request has 1 passenger, 15% of having 2 passengers, etc.

Main Procedure():
      initialRequests = Create initial random
      requests(numberOfRequests)
      newRequests = Create list of random requests to be inserted
      during the simulation(requestsPerHour * amountOfRunningHours)

      Calculate initial route for initialRequests()
      For each new request in newRequests:
            Determine new request time
            Determine the route travelled before the new request time,
            add to finalRoute
            Determine which previous requests have been completed
            before the new request time, add to completed requests
            Insert new request in requests
            Calculate new route, pickup times and dropoff times for
            requests

      Return finalRoute and completed requests
```

Pseudocode for the dynamic DRT simulation

As the search heuristic, Parallel Cheapest Insertion has been chosen to provide a viable initial solution. Guided Local Search (GLS) has been chosen as the metaheuristic to escape local cost minima. GLS has been proven to provide good solutions for TSPs and VRPs in an acceptable amount of time (Beullens et al., 2003; Kilby et al., 1999; Voudouris & Tsang, 1999).

Simulation of semi-flexible DRT systems

The semi-flexible DRT system is simulated in a different manner than dynamic DRT systems. Since the semi-flexible DRT system is not a continuous system, but rather a system which has alternating itineraries in each direction, between two predefined nodes which act as the starting or ending node, each itinerary (one bus trip from the starting node to the final node and vice-versa) is simulated separately. For each itinerary, the simulator determines from the given requests which requests can be served and which ones have to be served in a future itinerary. The software then produces a full itinerary, from a given starting node to a given end node.

The semi-flexible DRT simulation makes use of predefined trunk node sequences, which have been obtained by running a standard TSP algorithm (Google for Developers, 2023a) over each set of nodes

In the semi-flexible DRT system simulation, at first, a list of initial random requests is generated, just like in the dynamic DRT simulation. Based on these initial requests, a first travel itinerary is made. After this, random requests are periodically inserted, just like in the first simulation. When a request is inserted, it is assessed on two requirements:

-Whether the pickup node of the request is located before the drop-off node in the current trunk sequence

-Whether the pickup node is located later in the trunk sequence than the current location of the vehicle.

If the request satisfies both conditions, it is added to the current itinerary. If not, it is kept separately for processing in a later itinerary.

After the first itinerary has ended, the vehicle departs for the next itinerary after a set headway time. Before this itinerary starts, the trunk sequence reverses, since the new itinerary is in the reverse direction. Before the vehicle departs, the existing requests, which could not be processed before, are added if they comply to the trunk sequence (i.e., the pickup node comes before the drop-off node in the trunk sequence). Afterwards, the itinerary starts and the same methods are repeated as in the first sequence. This simulation is repeated for a set number of itineraries.

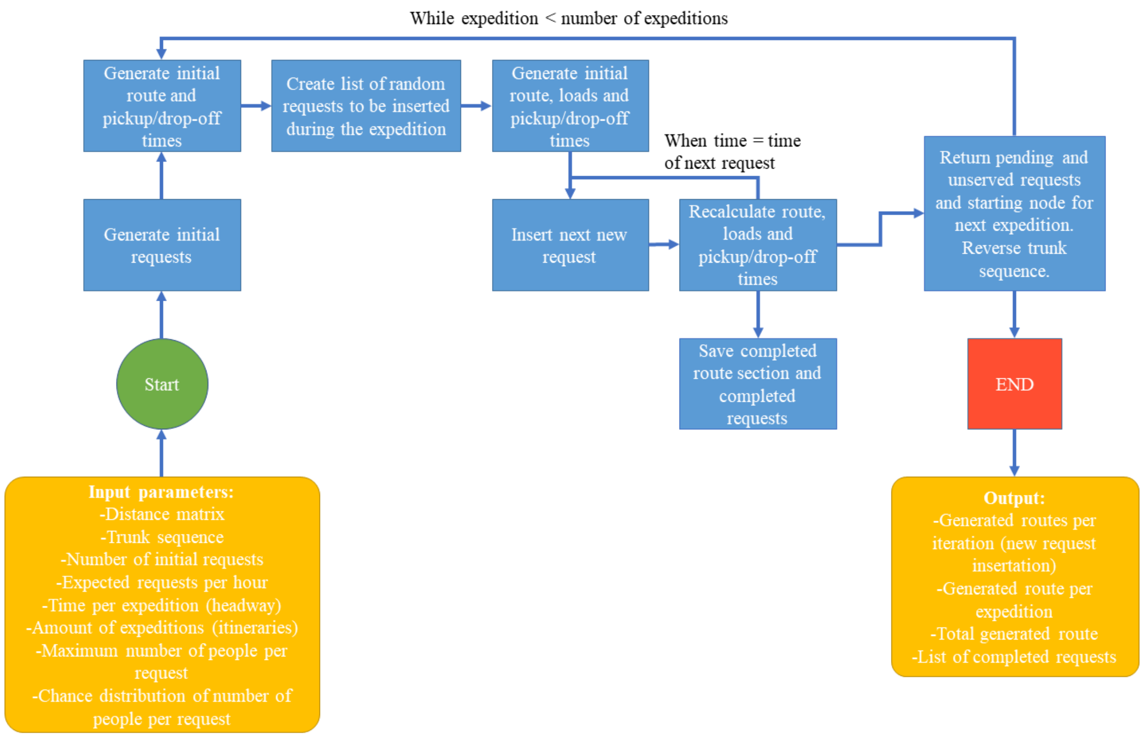

While expedition < number of expeditions
Generate initial route and pickup/drop-off times
Create list of random requests to be inserted during the expedition
Generate initial route, loads and pickup/drop-off times
When time = time of next request
Return pending and unserved requests and starting node for next expedition. Reverse trunk sequence.
Generate initial requests
Insert next new request
Recalculate route, loads and pickup/drop-off times
Save completed route section and completed requests
Start
END
Input parameters:
-Distance matrix
-Trunk sequence
-Number of initial requests
-Expected requests per hour
-Time per expedition (headway)
-Amount of expeditions (itineraries)
-Maximum number of people per request
-Chance distribution of number of people per request
Output:
-Generated routes per iteration (new request insertation)
-Generated route per expedition
-Total generated route
-List of completed requests

Figure 8: the conceptual scheme of the semi-flexible DRT simulation

```
#Input variables:

■ **distanceMatrix** = Matrix with distances between all node pairs
■ **sequence** = Trunk sequence of the node area
■ **numberOfRequests** = Number of initial requests at the start of the program.
■ **requestsPerHour** = Expected number of requests to be generated during the program, per simulated hour.
■ **expeditionTime** = Scheduled time for one travel itinerary in one direction
■ **expeditions** = Number of itineraries to be scheduled
■ **maxPeoplePerRequest** = Maximum amount of people per request. The analysis uses 5.
■ **requestWeights** = Chance distribution of amount of people per requests. The analysis uses [75, 15, 5, 3, 2] means a 75% chance that a random request has 1 passenger, 15% of having 2 passengers, etc.

Main Procedure():
      ■ initialRequests = Create initial random requests(numberOfRequests)
      ■ Calculate initial route for initialRequests()

      ■ For each new expedition in expeditions:
            ■ Retrieve the previous unserved requests from previous expeditions
            ■ newRequests = Create list of random requests to be inserted during the expedition(expeditionTime)

            ■ For each new request in newRequests:
                  ■ Calculate new route, pickup times and dropoff times for pending and previous unserved requests and the new request, from the starting node
                  ■ Append the part of the route that comes before the insertion of the next new request to the final route
                  ■ Determine which requests are serviced, pending and unserved
                  ■ Store the serviced requests
                  ■ Save the starting node for the next iteration

            ■ Print the route to Excel
            ■ Reverse the sequence for the next expedition

      ■ Return the final route and completed requests and print them in Excel
```

Semi-flexible DRT route calculation

The main part of this code that differs from the dynamic DRT simulation is the different method of route calculation. Instead of using a search heuristic, the route is simply calculated using the main trunk sequence. For example, node setup 12.1 is a setup with 12 randomly distributed nodes in the field. It has a trunk which has been calculated using a simple TSP solver:

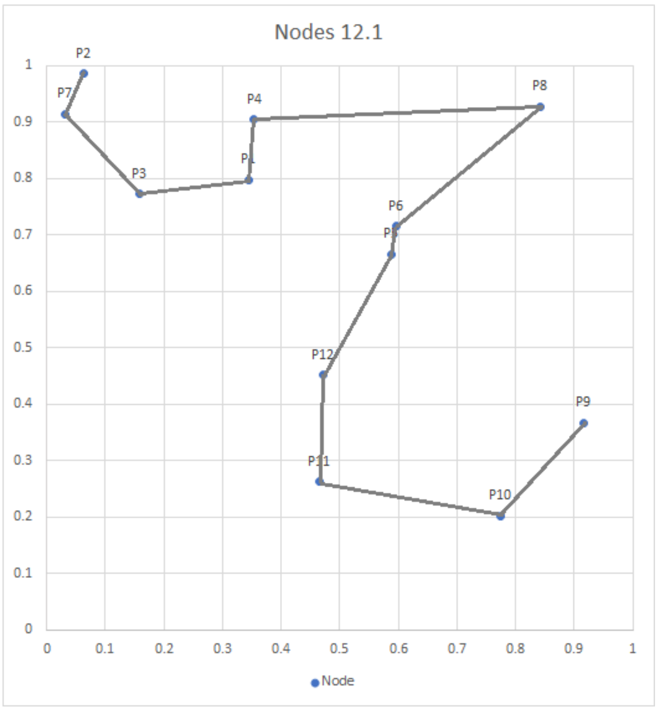


Figure 9: Node field 12.1 with its main trunk sequence as the grey line.

The trunk sequence in this case is [2, 7, 3, 1, 4, 8, 6, 5, 12, 11, 10, 9] for vehicle expeditions which start at node 2. Every time a new request is added to be processed, it is assessed if the request can be processed (i.e., in the right pickup and drop-off order, as well as starting at a point where the vehicle has not passed yet). If this is true, the new request is added to the list of requests to be served in the route.

The route is then simply determined as the order of all pickup and drop-off nodes in the sequence. If the list of requests to be served is the following, in the form of [pickup node, drop-off node, number of passengers]:

[4, 12, 1]
[3, 8, 2]
[8, 9, 2]
[1, 6, 1]

Then the pickup and drop-off nodes are nodes 1, 3, 4, 6, 8, 9 and 12. In the trunk sequence order, this gives an order of [2, 3, 1, 4, 8, 6, 12, 9]. Note that the starting node (2) and final node (9) are always included. This will then be the route the vehicle will travel in this particular expedition, which is visualised in figure 10. The occupancy of the vehicle at each point will be [0, 2, 3, 4, 4, 3, 2, 0].

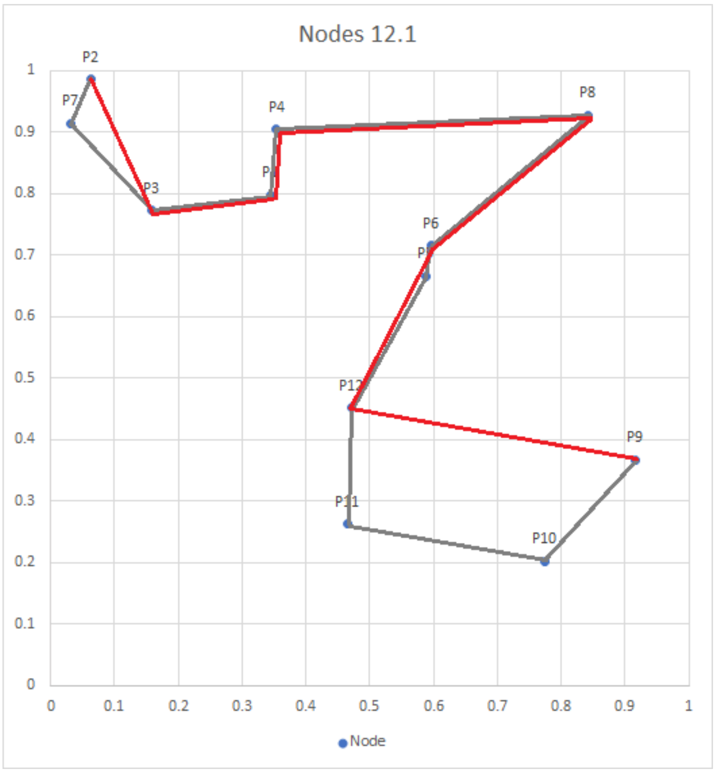


Figure 10: Node distribution 12.1 with its trunk sequence (grey) and example route (red).

This route will hold for this itinerary until a new request enters the system, which will then be handled according to the process described in the Research Approach. The total simulation will run for 12 itineraries of 30 minutes each, to make the runtime 6 hours, just like in the dynamic DRT simulation.

Case studies

In order to be able to assess the influence of varying demand scenarios, both simulations are run under a set of different scenarios. Firstly, a set of different node distributions is created. These node distributions vary from low node densities to high node densities: the node distributions have either 12, 16, 20, 24, 28, 32 or 36 nodes in their field. In order to account for the influence of the random distribution of the nodes, four random node fields are created for each node density scenario. This means that a total of 28 node fields are used in the simulation, four per node density.

Separate simulations are also run for different demand levels. These vary from low demand (4 new trip requests per hour) to high demand (24 new trip requests per hour), with increments of 4. These demand levels are expected demand levels: each minute, there is a certain chance that a new request is generated based on this value. For example, if the expected number of new trip requests per hour is 12, then each minute in that hour has a 12/60 = 1/5 chance of having a new trip request.

This gives a grand total of 168 scenarios, which are each simulated four times in the dynamic DRT simulation and four times in the semi-flexible DRT simulation. A schematic overview of the scenario structure is given in figure 11.

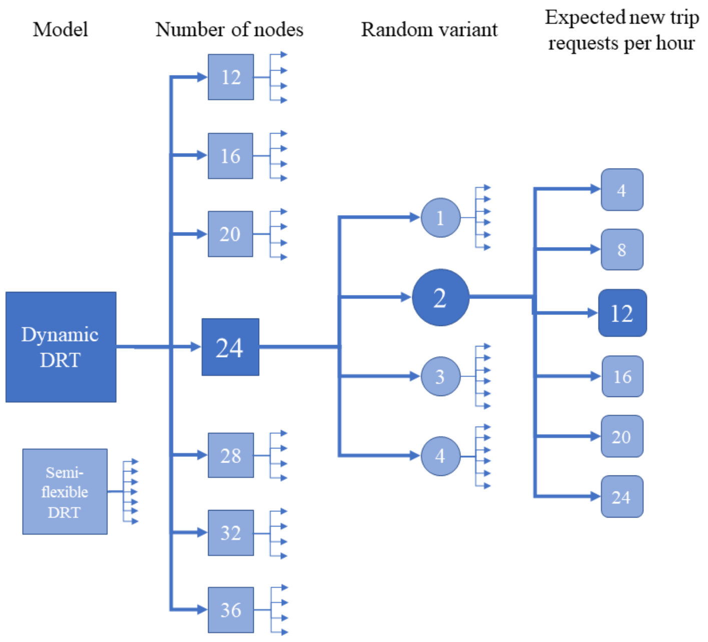


Figure 11: Schematic representation of the different simulation scenarios. This highlights the scenario for dynamic DRT, 24 nodes, variant 24.2, with 12 expected new trip requests per hour.

Each random variant of each node density scenario can be seen in appendix H, along with their trunk route for the semi-flexible DRT simulations.

Analysis of the results

Both simulations result in lists of both the traversed route in each iteration of the simulation, as well as a list of all served requests, like illustrated in tables 1, 2, and 3. The traversed routes contain the vehicle load at all times, and the served requests contain their calculated pickup and drop-off times

From the route itineraries, the following metrics are generated:

- **Vehicle load:** the number of people in the vehicle at all times. From this, the vehicle load can be visualised over time
- **Distance driven:** the total distance driven by the vehicle per itinerary. This is calculated as the sum of all straight-line driving times between each sequentially visited node. This is similar to the total itinerary time: however, this excludes waiting times when the vehicle is idle.
- **Distance ratio**: the ratio between the driven distance and the sum of all straight-line distances for all trip requests. This sum is the theoretical distance that the vehicle would drive if it would serve every request straight from its pickup to its drop-off point. This is used to assess how efficient the DRT routing is: the lower the ratio, the less distance travelled compared to the theoretical distance sum, and therefore the more efficient the DRT routing is.

These data will be used to generate several key metrics per trip request:

- **Pickup time delay:** the difference between the initially indicated pickup time and the actually realized pickup time.
- **Drop-off time delay:** the difference between the initially indicated drop-off time and the actually realized drop-off time.
- **Waiting time:** the difference between requested time and the true pickup time. This indicates how long one has had to 'wait' between requesting the trip and actually being picked up by the vehicle
- **Travel time:** the actual time spent in the vehicle, defined as the difference between the true drop-off time and the true pickup time.
- **Excess travel time:** defined as the time spent in vehicle compared the straight-line distance between the pickup and drop-off point. Remember that distance is the same as time in this simulation, indicating that the detour time is the extra time spent in the vehicle compared to a theoretical straight line-trip between both nodes. The metric is represented as a percentage: the extra percentage of distance driven. For example, if the straight-line time between two nodes is 4 minutes, and the trip time was 10 minutes, the detour ratio is 150%, since the realized travel time is 150% higher than the straight-line time. By definition, this metric cannot be lower than 0%.

These metrics are obtained for each different node density and demand level scenario. Based on these metrics, it can be assessed how they influence the performance of each DRT model.

Results

Simulations have been run for both the dynamic and semi-flexible DRT models, for every pickup and drop-off point amount, random node configuration and demand density. Each scenario has been simulated 4 times to account for randomness, which amounts to a grand total of 1344 simulations, which is 672 simulations per DRT model.

Note that, to maintain clarity, in this section examples of the results will be given, but not all results. All results can be consulted in appendices A through G.

Route itinerary metrics

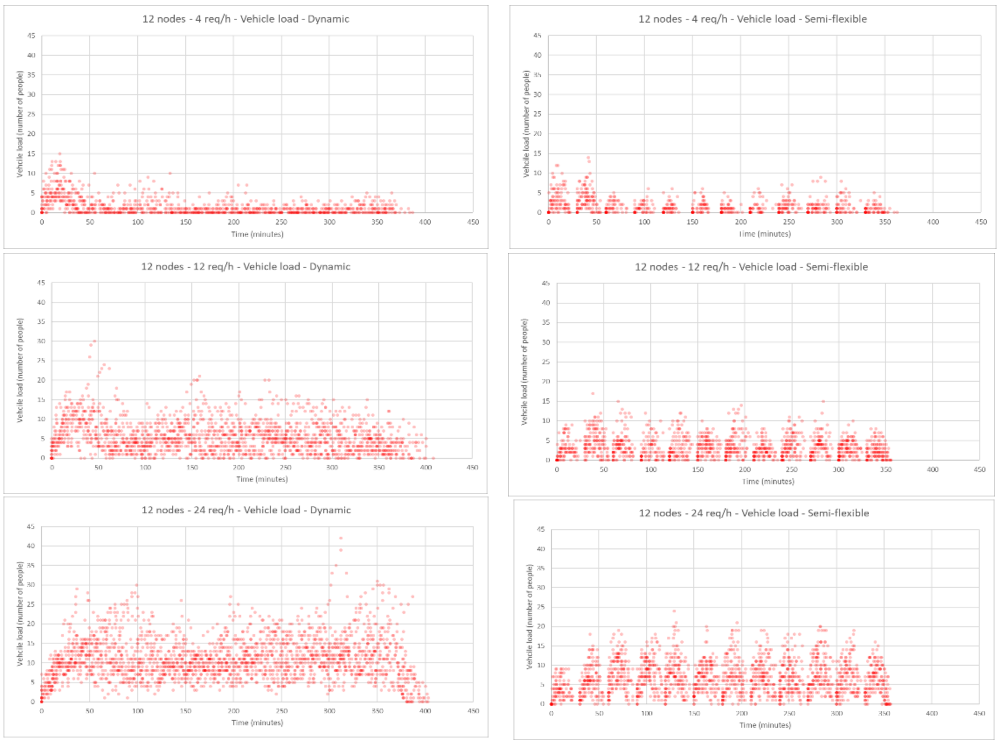
Vehicle load

Vehicle loads vary between the number of nodes in the system and the trip request level. In this section, the vehicle loads over time will be shown for a low node density (12 nodes), medium node density (24 nodes) and a high node density (36 nodes), and for a low trip request level (4 expected trip requests per hour), medium trip request level (12 expected request trips per hour) and a high trip request level (24 expected requests trips per hour). The vehicle loads are shown as scatter plots, with each point representing a vehicle load of one specific route in one specific simulation at a point in time. By combining these graphs into one for each scenario, a trend can be observed for vehicle loads over time.

Low node density

In the scenario with a low node density (12 nodes), vehicle loads are similar between the two DRT models. In the low demand scenario, after an initial peak load after processing the initial requests, the vehicle load is fairly constant, with the load seldomly exceeding 5. In the middle demand scenario, loads are also constant. However, in the dynamic model, the upper limit is a bit higher than in the semi-flexible model, with the semi-flexible model rarely exceeding a vehicle load of 10 people, while this is 15 people for dynamic DRT. In the high-demand scenario, the semi-flexible model keeps loads constant, roughly lower than 20, while the dynamic model shows a curve-like trend, with loads peaking between the 20 and 25 load marks.

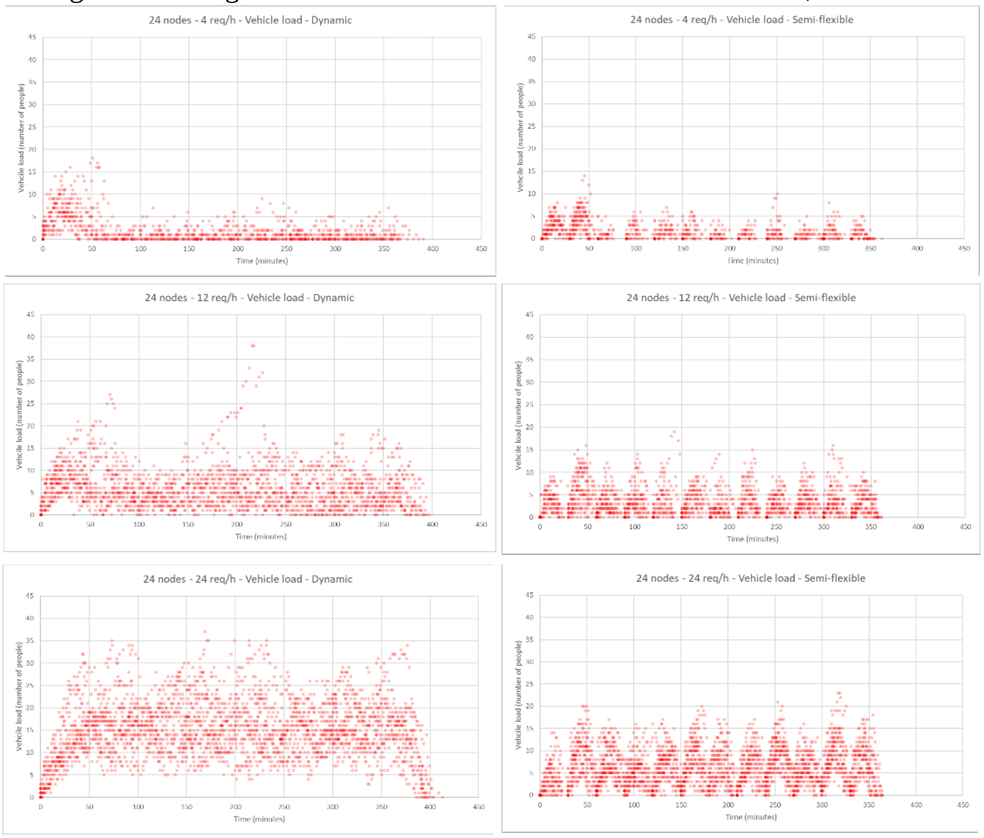
For the low node-density scenario, this implies that vehicle load peaks are higher for medium to high trip request levels, requiring more vehicle capacity in these higher demand scenarios. For an assumed minibus capacity of 24 (Martínez et al., 2015), this means that in high-demand scenarios, minibuses may not provide sufficient capacity when the number of pickup and drop-off nodes is limited.



Figures 12-17: Vehicle loads over time for 12 nodes, both DRT models.

Medium node density

In the medium node-density (24 nodes) scenario, the vehicle load shows a similar progression as in the low node-density scenario. However, for the dynamic DRT model, the vehicle load peaks are higher in the high-demand case. In the semi-flexible DRT model, this is not the case. This

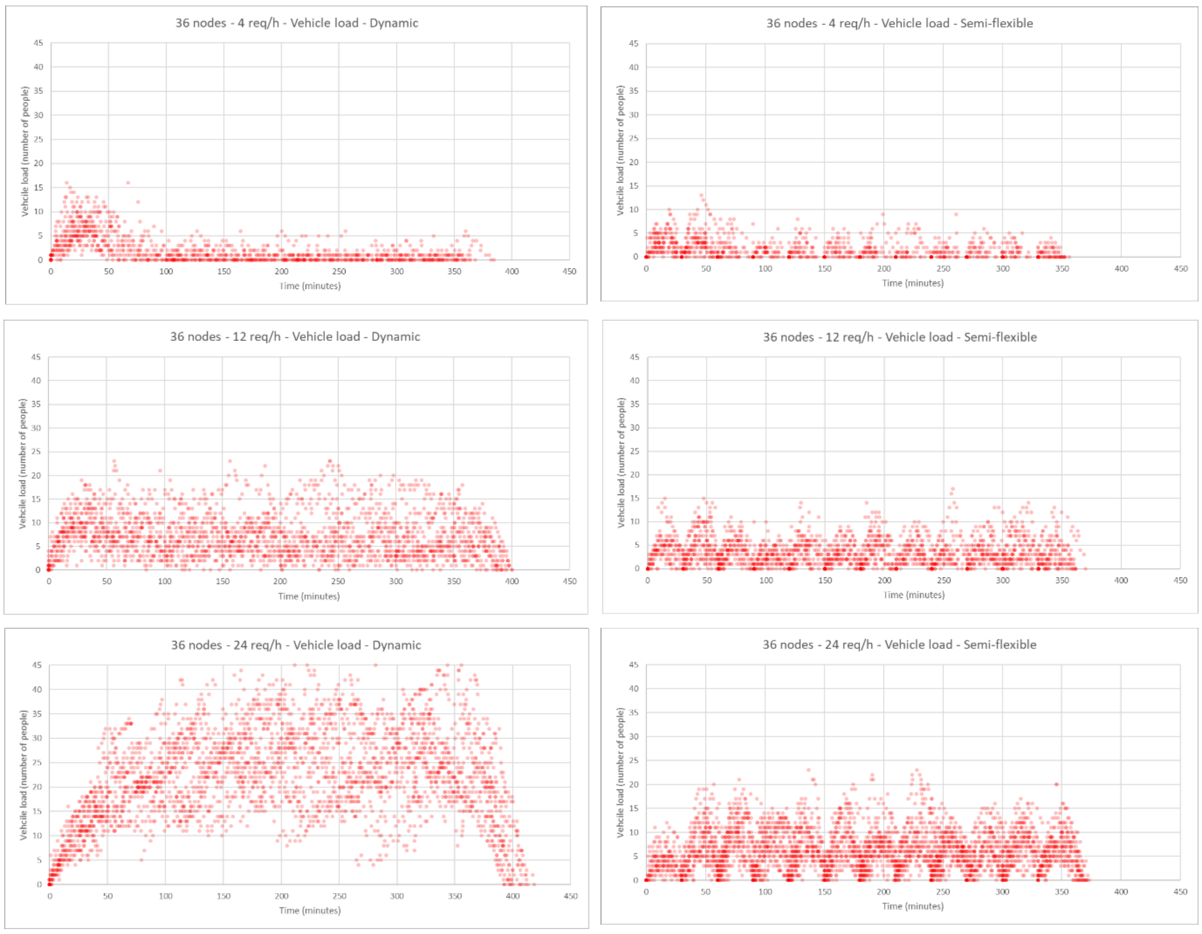


Figures 18-23: Vehicle loads over time for 24 nodes, both DRT models.

implies that for medium node densities, even moderate demand levels might cause vehicle loads to exceed the capacity of a minibus in dynamic DRT systems. In high demand scenarios, vehicle load peaks can be very high, while the vehicle loads show a lot of variation over time. This indicates that, besides the need for higher-capacity vehicles, people are aggregated over time in a less efficient way than in semi-flexible DRT.

High node density

In the high node density scenario, vehicle loads stay constant for both DRT models for low and medium demands. However, when demand is high, vehicle loads tend to disperse a lot from each other in the dynamic DRT model. Around half of the data points represent a vehicle load of more than 25 passengers during its itinerary. In comparison, the semi-flexible DRT model still keeps loads fairly constant, with peaks loads between 15 and 20 passengers. These graphs indicate that peak vehicle loads in dynamic DRT start to grow faster than the growth of travel demand in high node density scenarios.

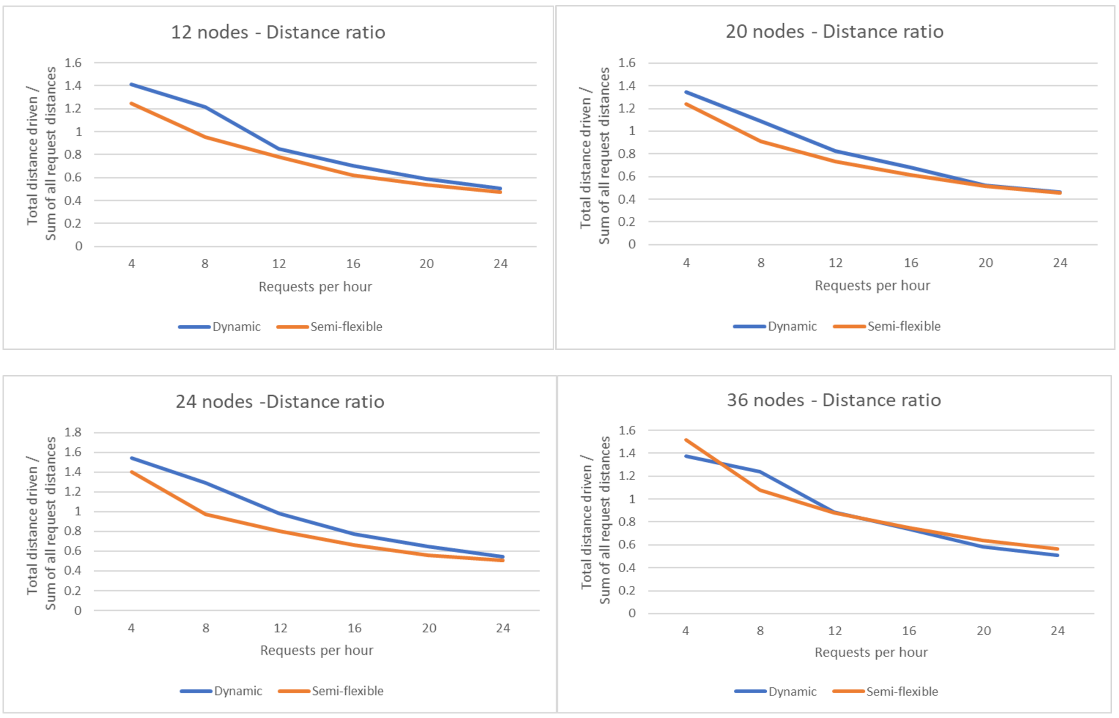


Figures 24-29: Vehicle loads over time for 36 nodes, both DRT models.

For the vehicle load metric overall, it is shown that dynamic DRT and semi-flexible DRT both aggregate passengers in a similar manner for low-to-medium node densities and low-to-medium demand levels. However, when demand levels exceed 12 requests per hour, semi-flexible DRT clearly aggregates the passengers more efficiently, leading to lower vehicle load peaks. Dynamic DRT would need bigger vehicles than minibuses to not exceed vehicle capacity, which is not the case for semi-flexible DRT.

Total distances driven

For both DRT models, the calculated distance ratio (the driven distance versus the distances between all pickup and drop-off node pairs) decreases, as expected, when the demand level increases, due to fewer ‘empty’ trips and higher passenger aggregation This means that both models more efficiently serve demand when demand increases. Between both models, the semi-flexible model seems to perform slightly better for lower demand scenarios than the dynamic model, except for the high node density scenario. However, this difference seems too small to be significant.



Figures 30-33: Distance ratios for various node-density scenarios.

Trip request metrics

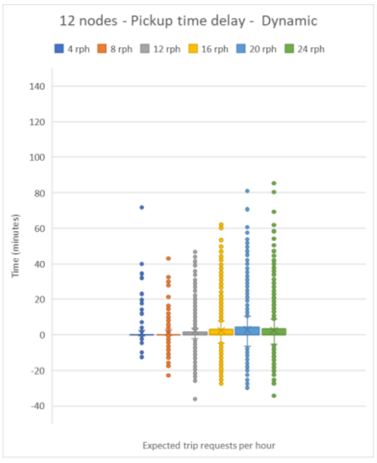
The trip request metrics are the metrics constructed from the pickup and drop-off time values obtained from each fulfilled trip request. These metrics are shown as boxplots to show their value distribution.

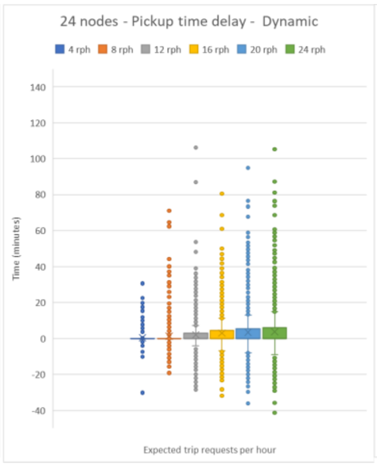
Pickup time delay

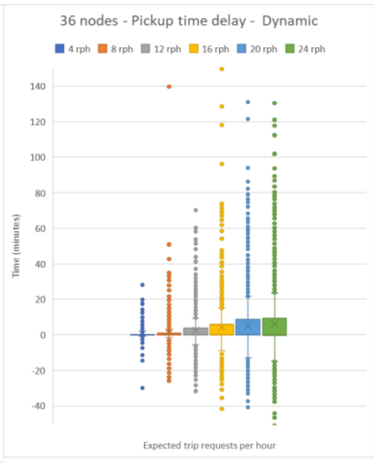
The difference between the initially indicated pickup time and the true pickup time are shown for dynamic DRT in figures 34-39. It can be seen that when demand is low (4 or 8 trip requests per hour), the pickup time delay is mostly very small, with most values being on or around 0 minutes. However, when looking at the outliers, there are some instances where the pickup time delay is much greater, even for low demand, going as far as a 70-minute delay.

For higher demands, the boxplots widen for all node density scenarios. In the 12-node scenario, the distribution of values has a limited increase, while in the 24-node and 36-node scenario, the distribution of values widens significantly. In the high-node density, high trip request rate scenario, only half of requests have their true pickup time within 10 minutes of the initially indicated pickup time.

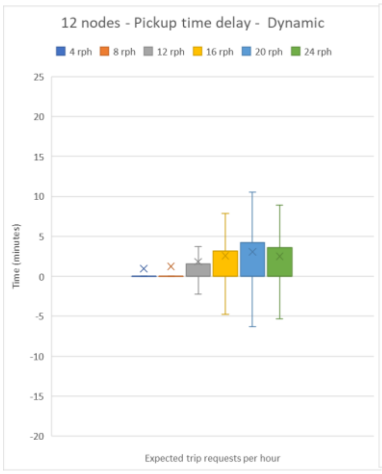
For semi-flexible DRT, pickup time delays are mostly non-existent, and only some outliers show a discrepancy of a few minutes between the pickup time values, as can be seen in appendix C. This shows that, while dynamic DRT has a similar pickup time reliability as semi-flexible DRT in low-demand scenarios, indicated pickup times become very unreliable in medium-to-high demand scenarios, making its deployment in these scenario types difficult.

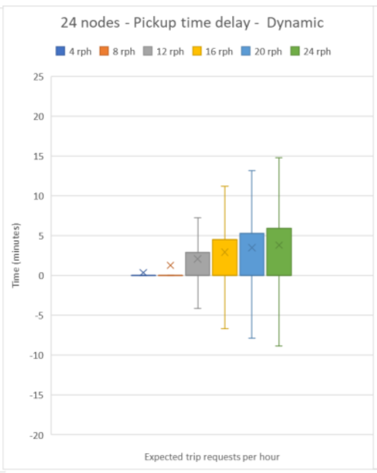


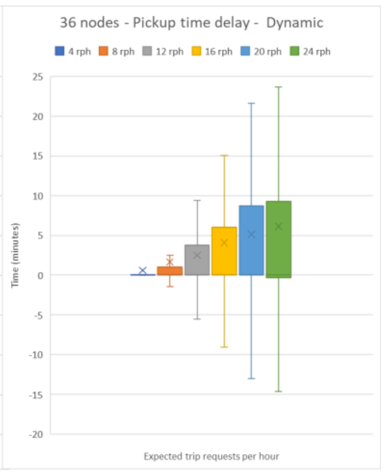




Figures 34, 35 and 36: Pickup time delay distributions (including outliers) for dynamic DRT.





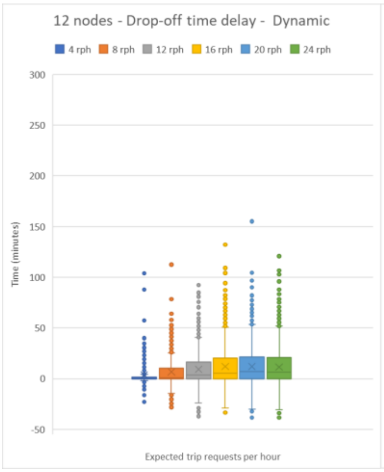


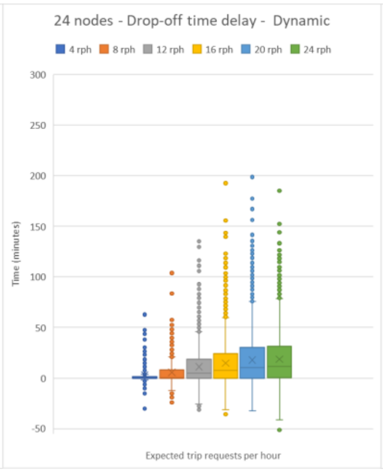
Figures 37, 38 and 39: Pickup time delay distributions (excluding outliers) for dynamic DRT.

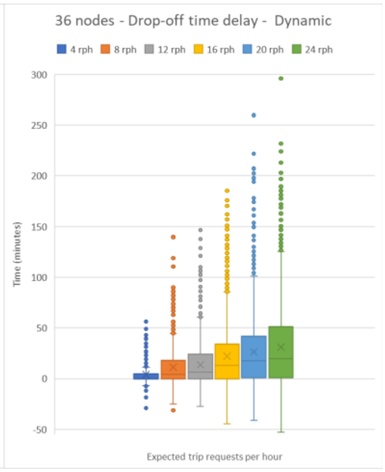
Drop-off time delay

For the drop-off time delays, the values show a similar trend as for the pickup time delays, but on a larger scale. For dynamic DRT, true drop-off times only generally stay within 10 minutes of the initial drop-off time for the 4 requests-per-hour scenario. The distribution of drop-off time delays widens with every increase in trip requests per hour. For low node density scenarios, the drop-off time delays tend to plateau under 50 minutes. However, for higher node density scenarios, the drop-off time delays tend to increase even further. Even in a low-node density (12 nodes) and low demand (8 trip requests per hour), only half of requests have their drop-off time stay within 10 minutes of the expected drop-off time. For higher node densities and demand rates, these values go far beyond, with outliers as far as 300 minutes.

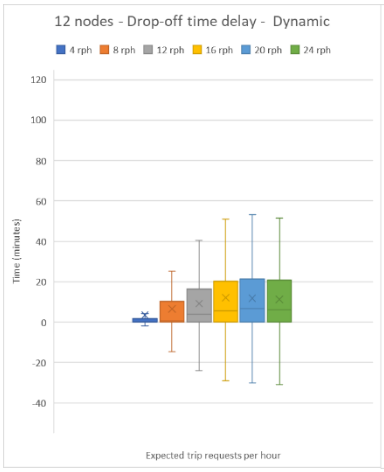
For semi-flexible DRT, drop-off time delays generally stay within 1 minute, even for high-demand scenarios, as can be seen in Appendix D. Outliers mostly stay under 10-minute differences. For dynamic DRT, drop-off time delays stay limited only for very low demand scenarios (4 expected requests per hour). This means that dynamic DRT is, regarding delays, only viable in very low demand scenarios. In higher demand scenarios, dynamic DRT systems would have unacceptable delays compared to semi-flexible DRT systems.

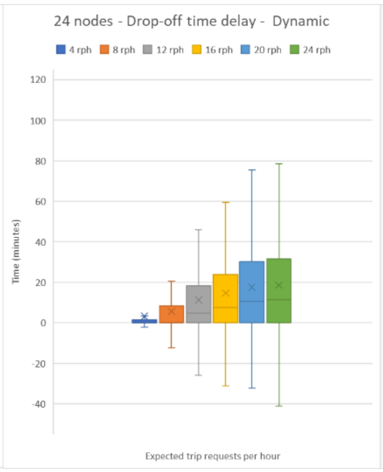


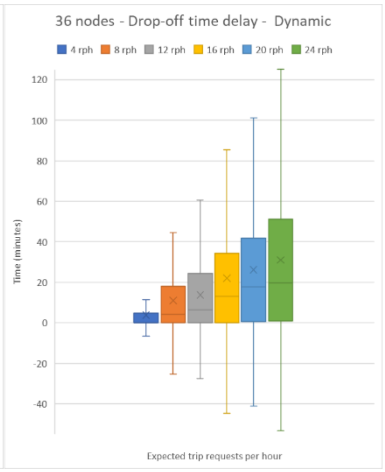




Figures 40, 41 and 42: Drop-off time delay distributions (including outliers) for dynamic DRT.



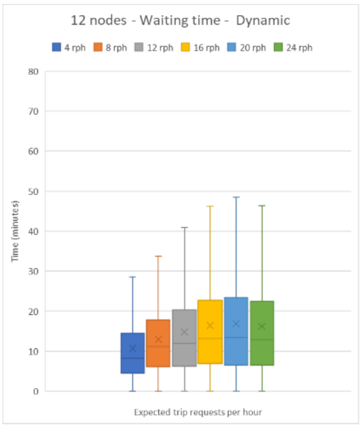


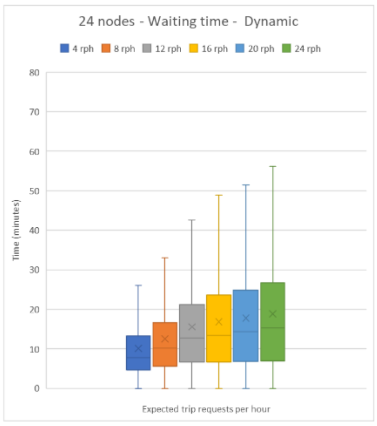


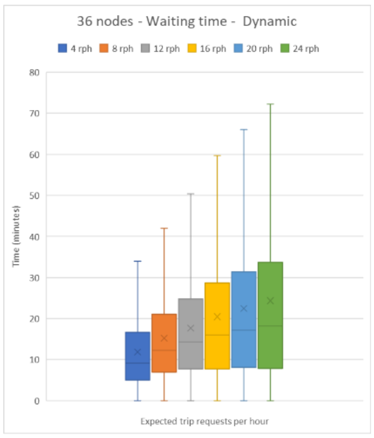
Figures 43, 44 and 45: Drop-off time delay distributions (including outliers) for dynamic DRT.

Waiting time

In dynamic DRT, waiting times tend to grow faster the higher the node density is in the system. This can be seen in figures 46, 47 and 48, where waiting times tend to stay limited for low-demand scenarios, even with a high node density. However, when trip demand rates increase, waiting times increase as well.

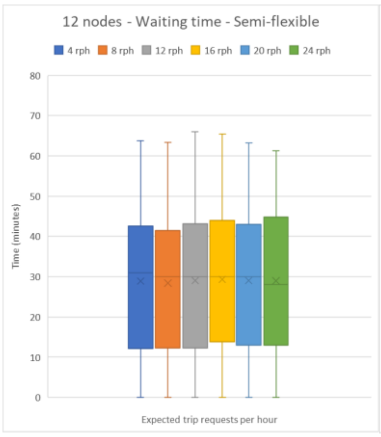


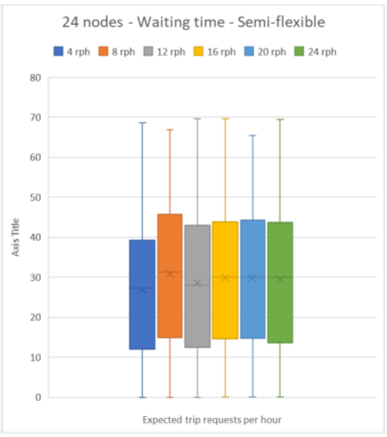


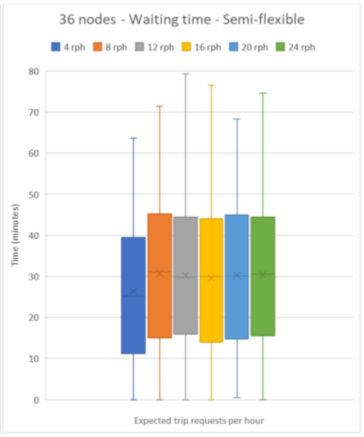


Figures 46, 47 and 48: Waiting time distributions (excluding outliers) for dynamic DRT.

For semi-flexible DRT (figures 49, 50 and 51), waiting times stay constant, independent of trip demand rate and node density. They tend to be higher than all scenarios for dynamic DRT, however, only being roughly equal to the high node density, high trip demand scenario in dynamic DRT. In all semi-flexible DRT scenarios, however, there are no outliers, in contrast to dynamic DRT, as can be seen in appendix E.





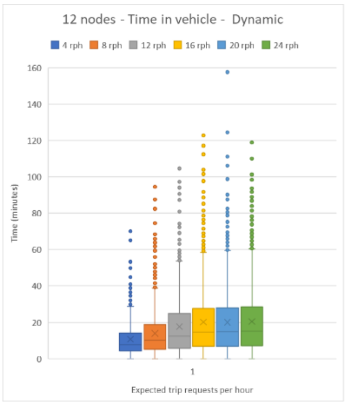


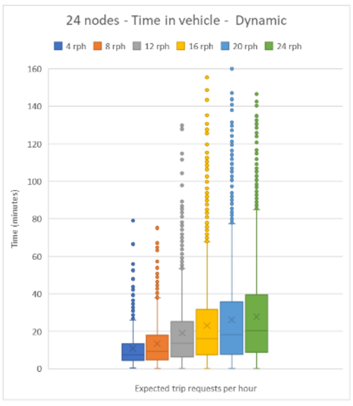
Figures 49, 50 and 51: Waiting time distributions for semi-flexible DRT.

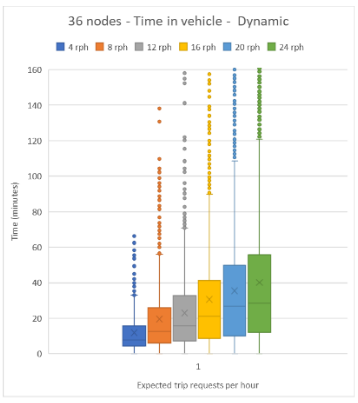
Overall, waiting times are higher for semi-flexible DRT, making this system less suitable for cases where it is desirable that trip requests are fulfilled as soon as possible. However, waiting times are independent of node density and demand levels in semi-flexible DRT. Dynamic DRT is more suitable when trip requests need to be fulfilled as soon as possible. Low-demand scenarios are especially suitable for this, since waiting times are shortest in these scenarios.

Trip duration

The time spent in the vehicle, i.e., the actual journey time for each trip request, is heavily dependent on the node density in mid- to high trip request rate scenarios for dynamic DRT. In low demand scenarios, half of trips are under 16 minutes, even in high node-density scenarios. For mid-to-high demand scenarios, the trip lengths plateau for the low node-density scenario (12 nodes), while for 24 nodes and 36 nodes, the trip times increase dramatically, with outliers as high as 260 minutes for the high-demand, high node density scenario.

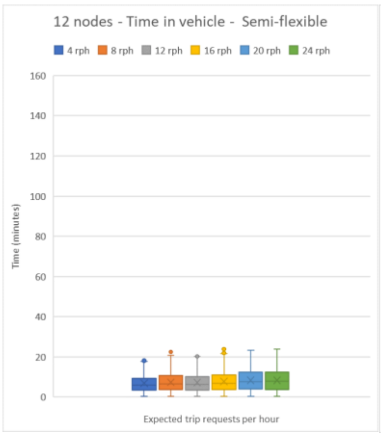


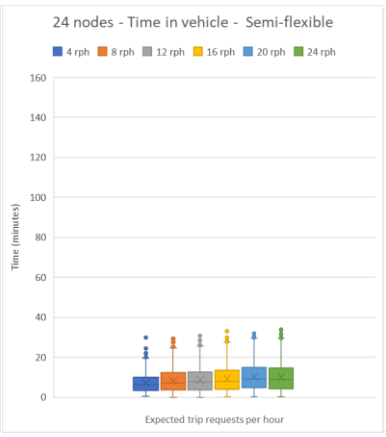


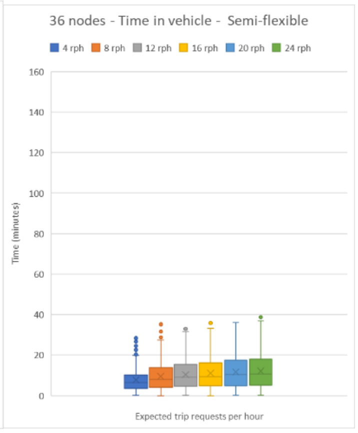


Figures 52, 53 and 54: Trip duration distributions (including outliers) for dynamic DRT. NB: some outliers extend outside the graphs.

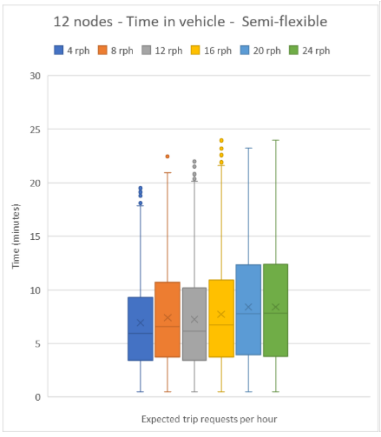
When comparing this to semi-flexible DRT, a stark difference can be seen. In every demand and every density scenario, the trip duration distributions are lower than for dynamic DRT, as can be seen in figures 55-57 and 58-60. As can be seen in figures 58-60, the trip durations remain fairly constant for each demand scenario in the low-node density scenario, while it tends to increase in the high-node density scenario. However, values stay far below that of their dynamic DRT counterparts, as is seen in figures 55-57.

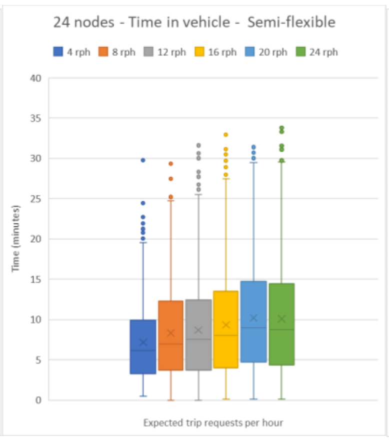


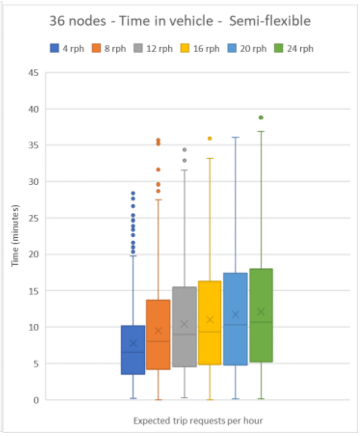




Figures 55, 56 and 57: Trip duration distributions (including outliers) for semi-flexible DRT.





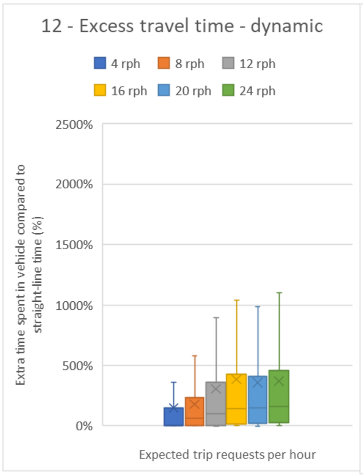


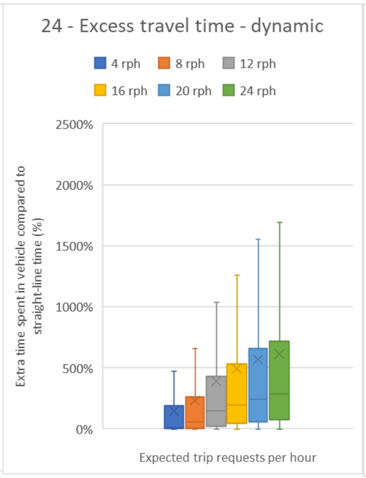
Figures 58, 59 and 60: Trip duration distributions (including outliers) for semi-flexible DRT, zoomed in.

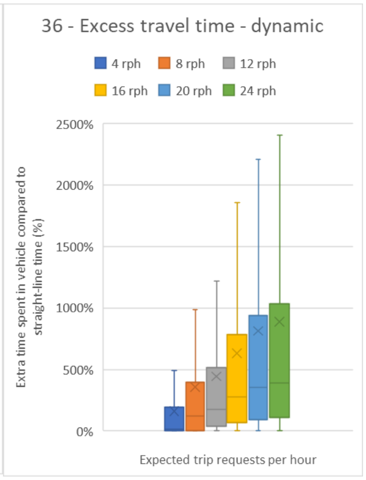
These figures show that trip durations are significantly higher in dynamic DRT, especially in medium-to-high demand scenarios and for higher node densities. In order to limit trip durations, it is not advisable to use dynamic DRT in higher demand scenarios, especially when regarding outliers.

Excess travel time

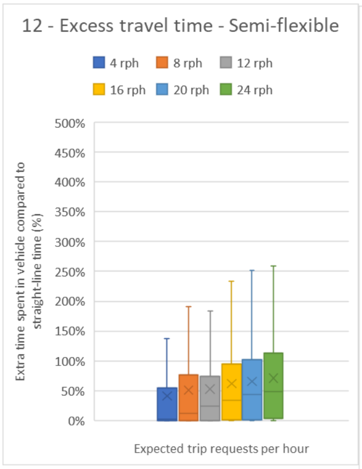
The detour ratios for each trip request show a much greater spread in dynamic DRT than in semi-flexible DRT. In dynamic DRT, trips have a trip detour ratio of a maximum of 500% only for the lowest demand scenario (4 trip requests per hour). In higher demand scenarios, this increases, especially in mid- to high-node density scenarios. For low node-density scenarios, the detour ratio distribution remains limited. However, these ratios still tend to be four to five times as high as the ratios in the same scenarios for semi-flexible DRT. In that model, the detour ratios also increase when trip demand increases, but consistently stay about four to five times as low as detour ratios in dynamic DRT.

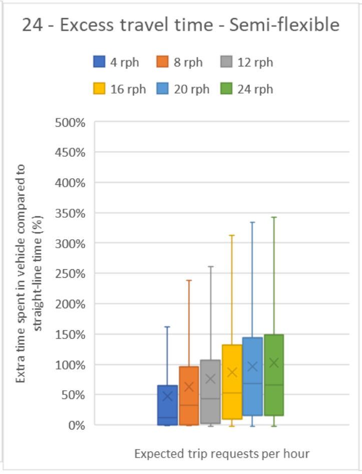


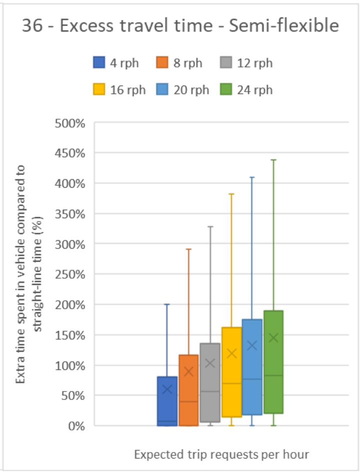




Figures 61, 62 and 63: Excess travel time distributions for dynamic DRT (excluding outliers).







Figures 64, 65 and 66: Excess travel time distributions for semi-flexible DRT (excluding outliers).

Outliers exist in both systems, but tend to be extremely high in dynamic DRT, as can be seen in appendix G. Therefore, dynamic DRT is not only systemically causing long detours for trips, but also occasionally causing extremely long detours. Regarding excess travel time, it is therefore advisable to only use dynamic DRT in very low demand scenarios, as excess travel time is comparable in only this scenario to semi-flexible DRT.

Overall Findings

From the results, it can be deduced how each DRT model performs under certain scenarios. Key Performance Indicators (KPIs) have been created from the observed metrics and are listed below.

Vehicle loads

Both dynamic DRT and semi-flexible DRT have comparable peak vehicle loads in low and medium demand scenarios. However, in high-demand scenarios, vehicle loads increase significantly in dynamic DRT, especially when the node density is also high. This indicates that semi-flexible DRT services can be operated with minibuses, if the capacity is assumed to be 24 people (Martínez et al., 2015). In high-demand DRT systems, this capacity is exceeded, regardless of the number of nodes in the system, and therefore, a bigger minibus or regular bus would be needed.

Total distances driven

In both DRT models, the relative distance driven compared to all pickup-drop-off node distances decreases when the trip demand increases, due to aggregation of passengers. Between the two models, there seems to be no significant difference in this relatively driven distance; perhaps this is due to dynamic DRT tending to detour more, while semi-flexible DRT has to perform more empty trips due to driving to the mandatory final stop of each itinerary.

Reliability

In cases of low demand, both DRT models perform well for pickup time reliability, except for the odd outlier in dynamic DRT. However, regarding drop-off times, only with very low demand (around 4 trip requests per hour), dynamic DRT can provide a reliable arrival time for most trip requests. In terms of reliability, this means that dynamic DRT only performs on a similar level to semi-flexible DRT when demand is very low. Semi-flexible DRT models are therefore preferred for any case above four expected requests per hour: they maintain a high reliability even in high-demand scenarios.

Waiting times

Regarding waiting times, dynamic DRT performs clearly better in this scenario than semi-flexible DRT, with median waiting times being around 10 minutes, compared to around 30 minutes for semi-flexible DRT. This is probably due to the set itinerary schedule of semi-flexible DRT, while dynamic DRT can instantly respond to a new trip request. This indicates that dynamic DRT is more suitable when an instant booking option is preferred; for semi-flexible DRT,

Travel time

Individual trips more often experience long detours in dynamic DRT than in semi-flexible DRT, for any demand scenario. This might not be as big of a problem for short trips, especially when considering the often-shorter waiting time beforehand; however, in bigger areas with longer overall distances the additional travel time might become unacceptable. This indicates that dynamic DRT should not be deployed in areas that are relatively large; and if so, only for very low demand scenarios.

Conclusions and Discussion

In conclusion, the overall picture indicates that dynamic DRT performs similarly to semi-flexible DRT in very low demand scenarios (around 4 expected trip requests per hour) in systems with a low to medium node density. When taking the benefits of dynamic DRT into account (freedom of movement and shorter waiting times), dynamic DRT can be a feasible solution in these kinds of scenarios. In scenarios with medium to high demand (8 expected trip requests or higher), dynamic DRT becomes impractical, mainly due to reliability problems and excessive travel times. It is therefore recommended, when DRT is the preferred system of public transport in an area, to implement DRT as a semi-flexible model when demand levels are expected to regularly exceed 4 requests per hour. As a benchmark, 6 requests per hour could be used as a soft border between choosing dynamic DRT and semi-flexible DRT. In any case, it is advisable to avoid dynamic DRT for systems with a high number of pickup and drop-off nodes. Dynamic DRT is therefore found to be a system better suited to very small-scale systems with low demand, as can be found in rural areas. For example, taxi DRT services could be dynamic in very rural areas, as well as bus services with very low demand. Semi-flexible DRT can be implemented in areas where demand is higher, like clusters of towns, suburban areas or in cases where a public transport system is needed as a feeder system for a higher capacity bus or rail system.

The results presented in this paper give an insight in how the two DRT models perform in different demand and vehicle stop density scenarios. As semi-flexible DRT has been shown to perform better in higher-demand scenarios, it can be asserted with more certainty that semi-flexible DRT provides a good ‘in-between’ alternative between dynamic DRT and fixed bus line public transport. As literature shows that DRT is often more cost-effective than fixed-line public transport, semi-flexible DRT has the potential to provide adequate, reliable and scalable public transport with a lower cost than fixed-line public transport.

As this is a simulation of theoretical cases, this study can be interpreted as a general insight into the performance of dynamic and semi-flexible DRT systems. For specific use cases, these findings can be used as a rule of thumb, but further analysis on the specific area is always advisable. As each area has different mobility patterns, geography and modal splits, the choice of public transport models should always be made regarding the local context.

In order to gain further insights for different scenarios, future research can analyse semi-flexible DRT under different scenarios. For example, multiple vehicles in one system could be simulated, as well as how semi-flexible DRT systems behave as a feeder system for fixed-line public transport. Also, comparisons between semi-flexible DRT and fixed-line public transport could give more insights in the demand and density levels at which semi-flexible DRT systems become less feasible than fixed-line DRT systems. Such insights could further the theoretical basis on which local authorities and public transport operators make their decisions, and bring more cost-effective and efficient public transport to everyone.

References

Alonso-Mora, J., Samaranayake, S., Wallar, A., Frazzoli, E., & Rus, D. (2017). On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences of the United States of America*, *114*(3), 462–467. https://doi.org/10.1073/PNAS.1611675114

Beullens, P., Muyldermans, L., Cattrysse, D., & Van Oudheusden, D. (2003). A guided local search heuristic for the capacitated arc routing problem. *European Journal of Operational Research*, *147*(3), 629–643. https://doi.org/10.1016/S0377-2217(02)00334-X

Bruzzone, F., Scorrano, M., & Nocera, S. (2020). *The combination of e-bike-sharing and demand-responsive transport systems in rural areas: A case study of Velenje*. https://doi.org/10.1016/j.rtbm.2020.100570

Cordeau, J. F., & Laporte, G. (2007). The dial-a-ride problem: Models and algorithms. *Annals of Operations Research*, *153*(1), 29–46. https://doi.org/10.1007/S10479-007-0170-8/METRICS

Cordeau, J. F., Laporte, G., Potvin, J. Y., & Savelsbergh, M. W. P. (2007). Transportation on Demand. *Handbooks in Operations Research and Management Science*, *14*(C), 429–466. https://doi.org/10.1016/S0927-0507(06)14007-4

Coutinho, F. M., van Oort, N., Christoforou, Z., Alonso-González, M. J., Cats, O., & Hoogendoorn, S. (2020). Impacts of replacing a fixed public transport line by a demand responsive transport system: Case study of a rural area in Amsterdam. *Research in Transportation Economics*, *83*, 100910. https://doi.org/10.1016/J.RETREC.2020.100910

Gomes, R., Pinho de Sousa, J., & Galvão Dias, T. (2015). Sustainable Demand Responsive Transportation systems in a context of austerity: The case of a Portuguese city. *Research in Transportation Economics*, *51*, 94–103. https://doi.org/10.1016/J.RETREC.2015.07.011

Google for Developers. (n.d.). *Google OR-Tools*. Retrieved June 7, 2023, from https://developers.google.com/optimization

Google for Developers. (2023a, January 16). *Google OR-Tools: Traveling Salesperson Problem*. https://developers.google.com/optimization/routing/tsp

Google for Developers. (2023b, January 16). *Google OR-Tools: Vehicle Routing with Pickups and Deliveries*. https://developers.google.com/optimization/routing/pickup_delivery

Ho, S. C., Szeto, W. Y., Kuo, Y. H., Leung, J. M. Y., Petering, M., & Tou, T. W. H. (2018). A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological*, *111*, 395–421. https://doi.org/10.1016/J.TRB.2018.02.001

Hoos, H. H., & Stützle, T. (2005). TRAVELLING SALESMAN PROBLEMS. *Stochastic Local Search*, 357–416. https://doi.org/10.1016/B978-155860872-6/50025-1

Hunsaker, B., & Savelsbergh, M. (2002). Efficient feasibility testing for dial-a-ride problems. *Operations Research Letters*, *30*(3), 169–173. https://doi.org/10.1016/S0167-6377(02)00120-7

Jain, S., & Van Hentenryck, P. (2011). Large neighborhood search for dial-a-ride problems. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial*

Intelligence and Lecture Notes in Bioinformatics), 6876 LNCS, 400–413. https://doi.org/10.1007/978-3-642-23786-7_31/COVER

Kilby, P., Prosser, P., & Shaw, P. (1999). Guided Local Search for the Vehicle Routing Problem with Time Windows. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, 473–486. https://doi.org/10.1007/978-1-4615-5775-3_32

Koffman, D. (2004). *Operational experiences with flexible transit services*. https://books.google.com/books?hl=nl&lr=&id=dhnhHALLlawC&oi=fnd&pg=PA15&dq=Operational+Experiences+With+Flexible+Transit+Services&ots=3bDGK_MkDq&sig=WwE-x5pk0ueRHCC7OhDihnWTGAM

Konstantakopoulos, G. D., Gayialis, S. P., & Kechagias, E. P. (2022). Vehicle routing problem and related algorithms for logistics distribution: a literature review and classification. *Operational Research*, *22*(3), 2033–2062. https://doi.org/10.1007/S12351-020-00600-7/TABLES/4

Lave, R., & Mathias, R. (2000). State of the Art of Paratransit. *Transportation in the New Millennium*.

Li, X., Liu, W., Qiao, J., Li, Y., & Hu, J. (2023). An Enhanced Semi-Flexible Transit Service with Introducing Meeting Points. *Networks and Spatial Economics*, 1–41. https://doi.org/10.1007/S11067-022-09583-8/FIGURES/20

Li, X., Wang, T., Xu, W., Li, H., & Yuan, Y. (2021). *A novel model and algorithm for designing an eco-oriented demand responsive transit (DRT) system*. https://doi.org/10.1016/j.tre.2021.102556

Martínez, L., Viegas, J., Science, T. E.-T., & 2015, undefined. (2015). Formulating a new express minibus service design problem as a clustering problem. *Pubsonline.Informs.Org*, *49*(1), 85–98. https://doi.org/10.1287/trsc.2013.0497

Mehran, B., Yang, Y., & Mishra, S. (2020). Analytical models for comparing operational costs of regular bus and semi-flexible transit services. *Public Transport*, *12*(1), 147–169. https://doi.org/10.1007/S12469-019-00222-Z

Mishra, S., & Mehran, B. (2020). Assessment of delivery models for semi-flexible transit operation in low-demand conditions. *Transport Policy*, *99*, 275–287. https://www.sciencedirect.com/science/article/pii/S0967070X20302626

Mishra, S., & Mehran, B. (2023). Optimal design of integrated semi-flexible transit services in low-demand conditions. *IEEE Access*. https://doi.org/10.1109/ACCESS.2023.3260727

Palmer, K., Dessouky, M., & Abdelmaguid, T. (2004). Impacts of management practices and advanced technologies on demand responsive transit systems. *Transportation Research Part A: Policy and Practice*, *38*(7), 495–509. https://doi.org/10.1016/J.TRA.2004.05.002

Papanikolaou, A., & Basbas, S. (2020). *Analytical models for comparing Demand Responsive Transport with bus services in low demand interurban areas*. https://doi.org/10.1080/19427867.2020.1716474

Papanikolaou, A., Basbas, S., Mintsis, G., & Taxiltaris, C. (2017). A methodological framework for assessing the success of Demand Responsive Transport (DRT) services. *Transportation Research Procedia*, *24*, 26–27. https://doi.org/10.1016/j.trpro.2017.05.095

Parragh, S. N., Cordeau, J. F., Doerner, K. F., & Hartl, R. F. (2012). Models and algorithms for the heterogeneous dial-a-ride problem with driver-related constraints. *OR Spectrum*, *34*(3), 593–633. https://doi.org/10.1007/S00291-010-0229-9/METRICS

Posada, M., Andersson, H., & Häll, C. H. (2017). The integrated dial-a-ride problem with timetabled fixed route service. *Public Transport*, *9*(1–2), 217–241. https://doi.org/10.1007/S12469-016-0128-9

Sörensen, L., Bossert, A., Jokinen, J. P., & Schlüter, J. (2021). How much flexibility does rural public transport need? – Implications from a fully flexible DRT system. *Transport Policy*, *100*, 5–20. https://doi.org/10.1016/J.TRANPOL.2020.09.005

Voudouris, C., & Tsang, E. (1999). Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, *113*(2), 469–499. https://doi.org/10.1016/S0377-2217(98)00099-X

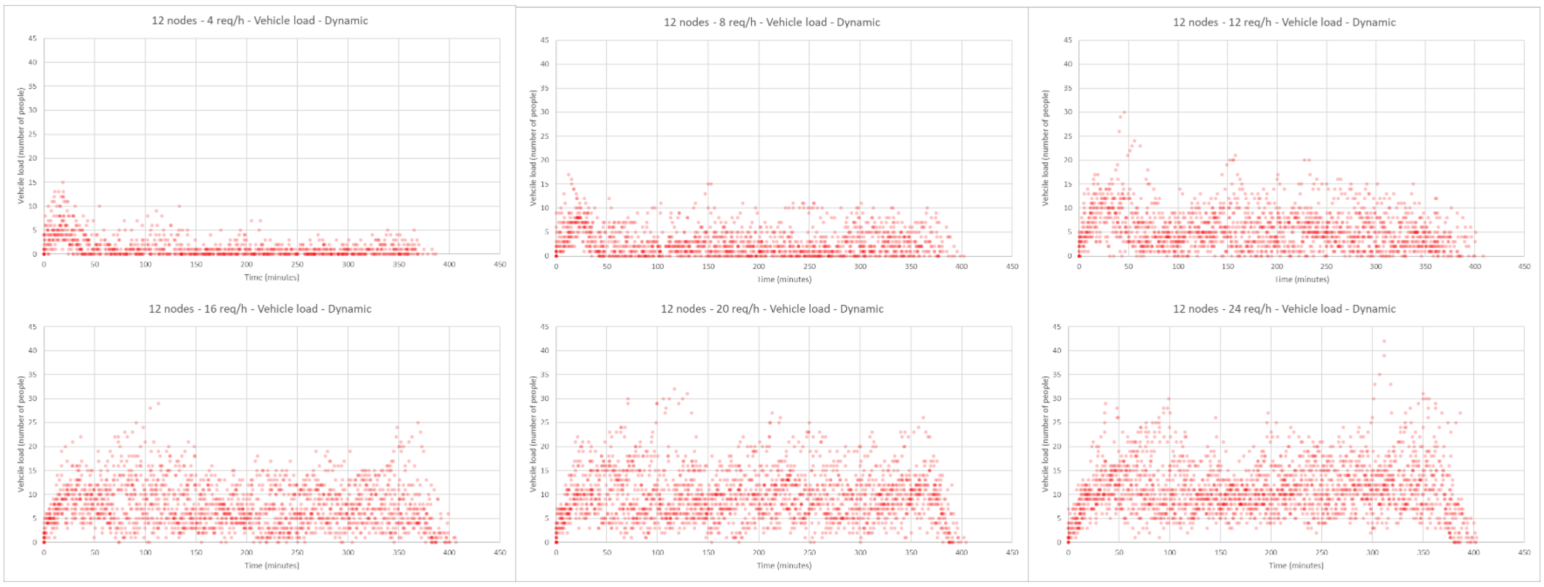
Wilson, N., Sussman, J., Goodman, L., & Higonnet, B. (1969). *Simulation of a computer aided routing system (CARS)*. https://repository.lib.ncsu.edu/bitstream/handle/1840.4/7558/1969_0016.pdf?sequence=1

Wong, K. I., & Bell, M. G. H. (2006). Solution of the Dial-a-Ride Problem with multi-dimensional capacity constraints. *International Transactions in Operational Research*, *13*(3), 195–208. https://doi.org/10.1111/J.1475-3995.2006.00544.X

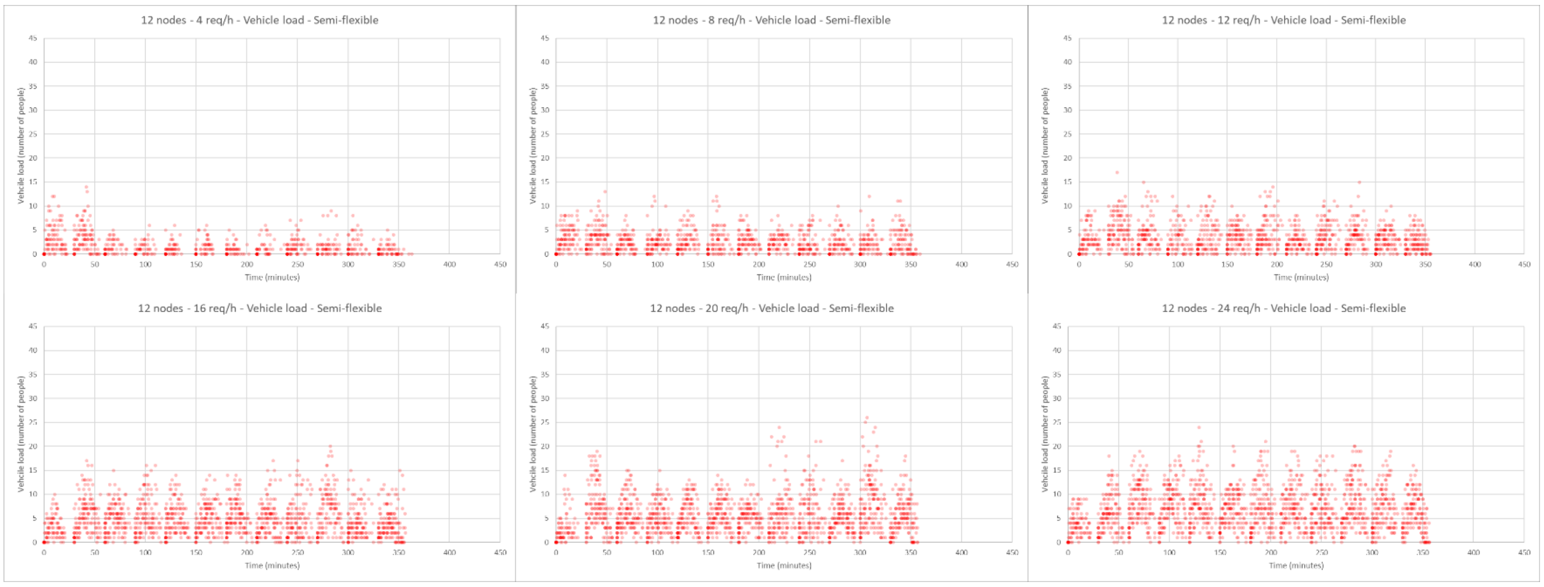
Xiang, Z., Chu, C., & Chen, H. (2006). A fast heuristic for solving a large-scale static dial-a-ride problem under complex constraints. *European Journal of Operational Research*, *174*(2), 1117–1139. https://doi.org/10.1016/J.EJOR.2004.09.060

Appendices

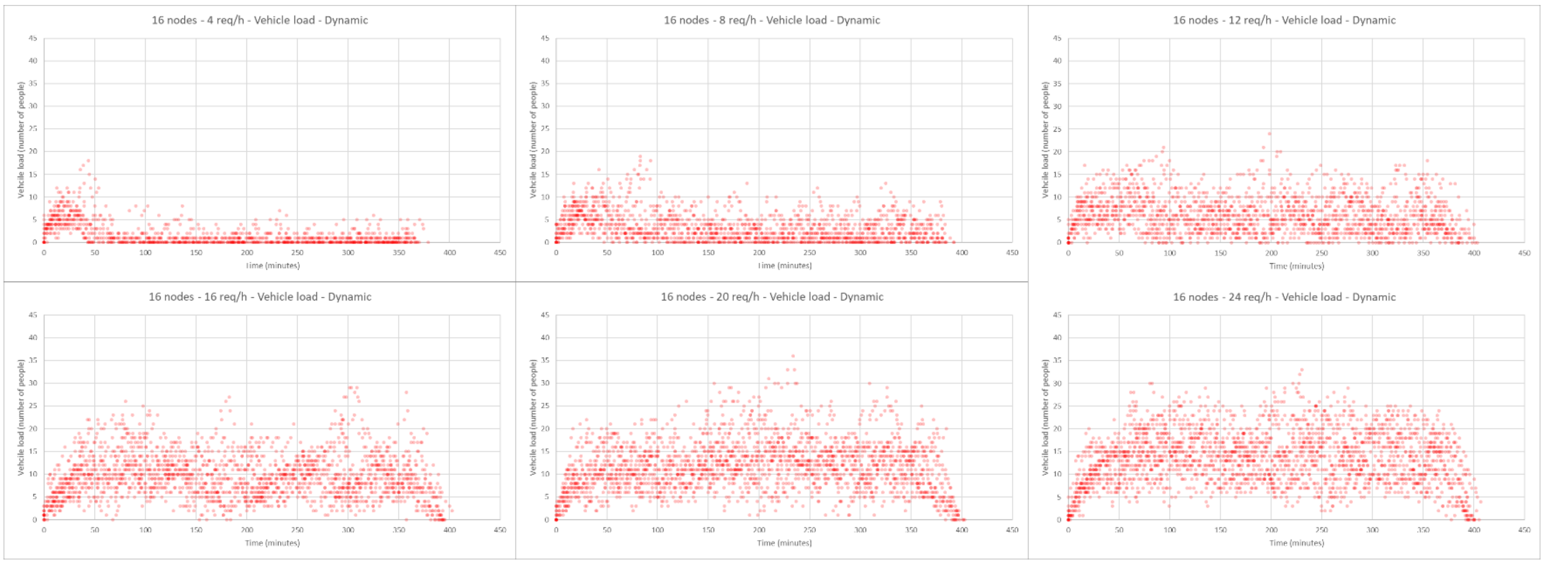
Appendix A: Vehicle loads over time



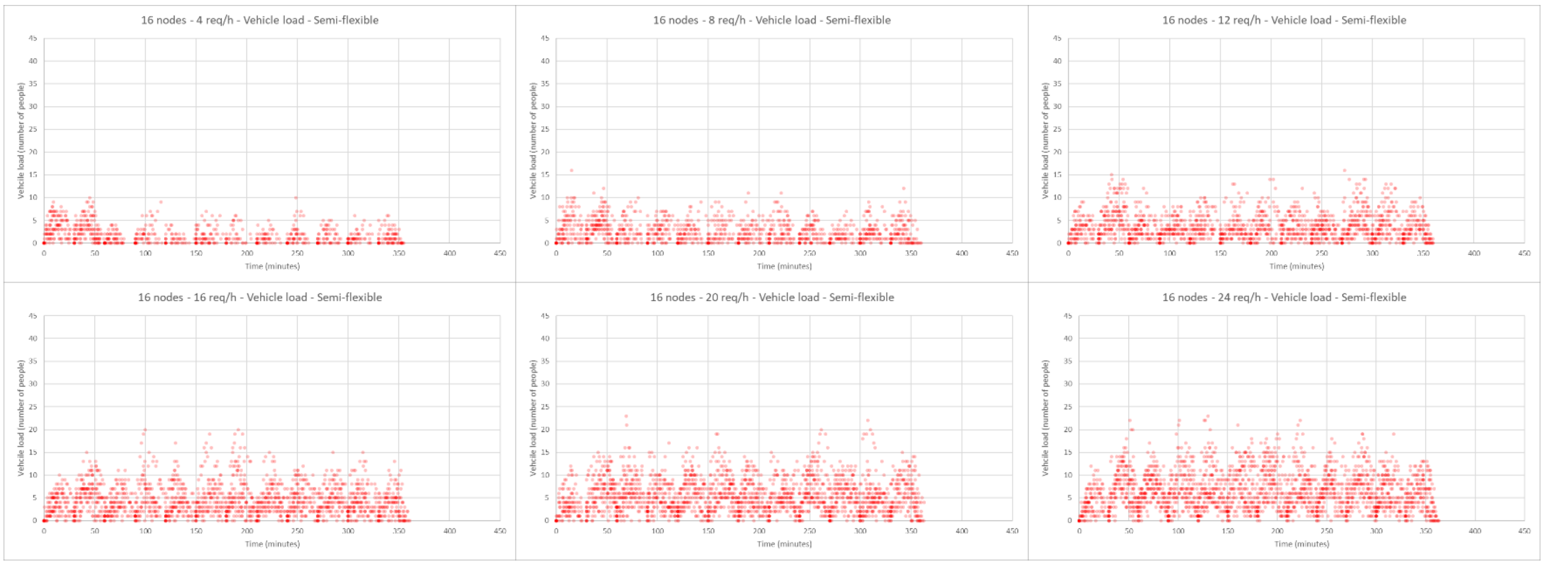
Appendix A.1: Vehicle loads over time, node density = 12 nodes, dynamic DRT



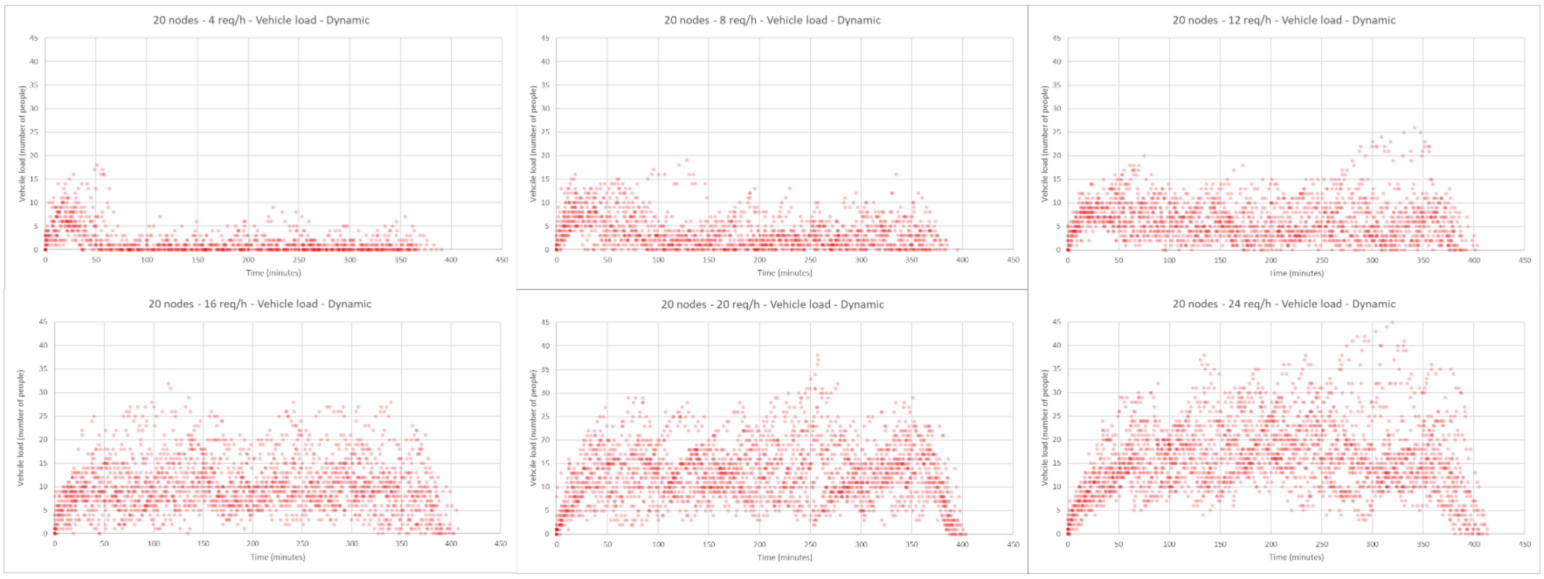
Appendix A.2: Vehicle loads over time, node density = 12 nodes, semi-flexible DRT



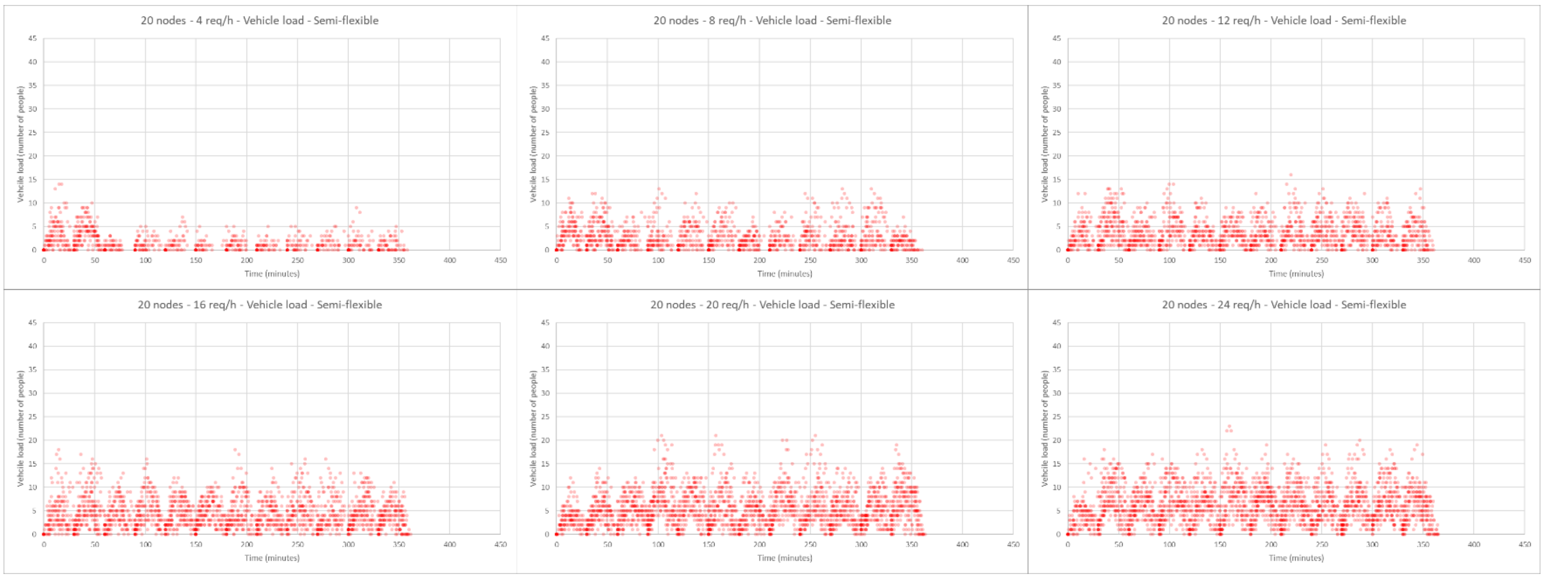
Appendix A.3: Vehicle loads over time, node density = 16 nodes, dynamic DRT



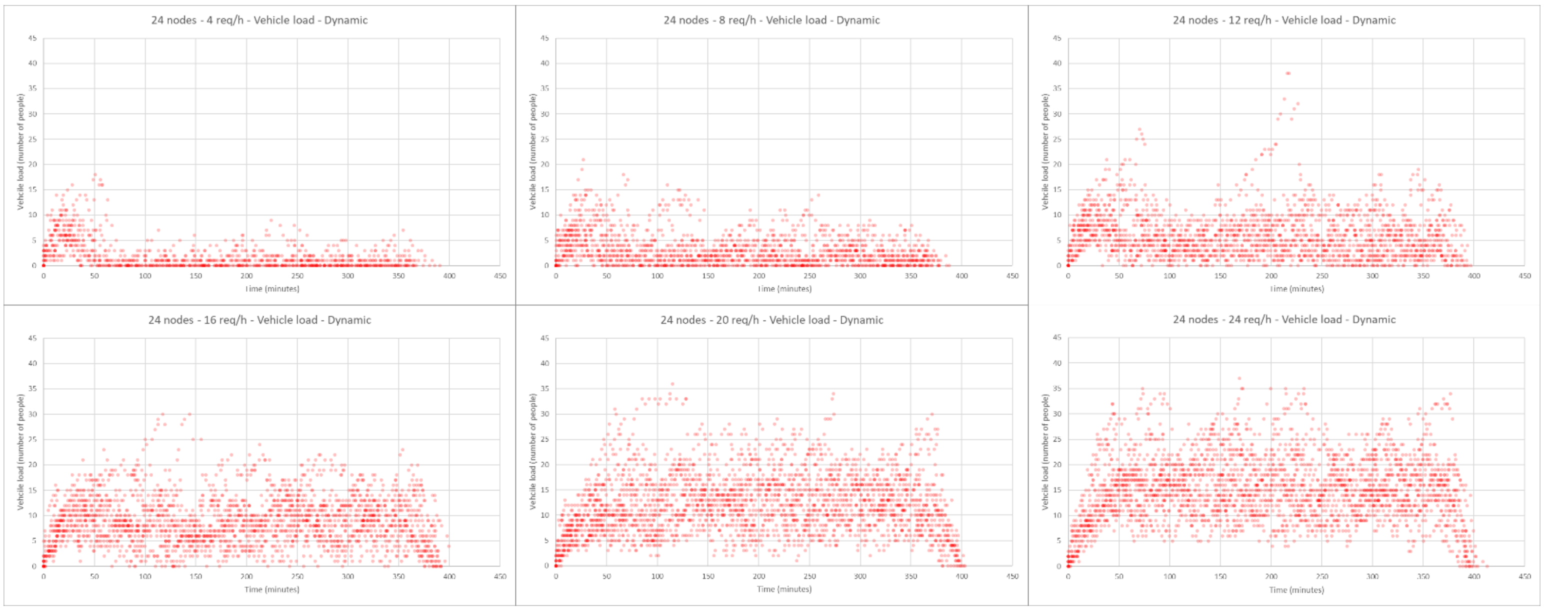
Appendix A.4: Vehicle loads over time, node density = 16 nodes, semi-flexible DRT



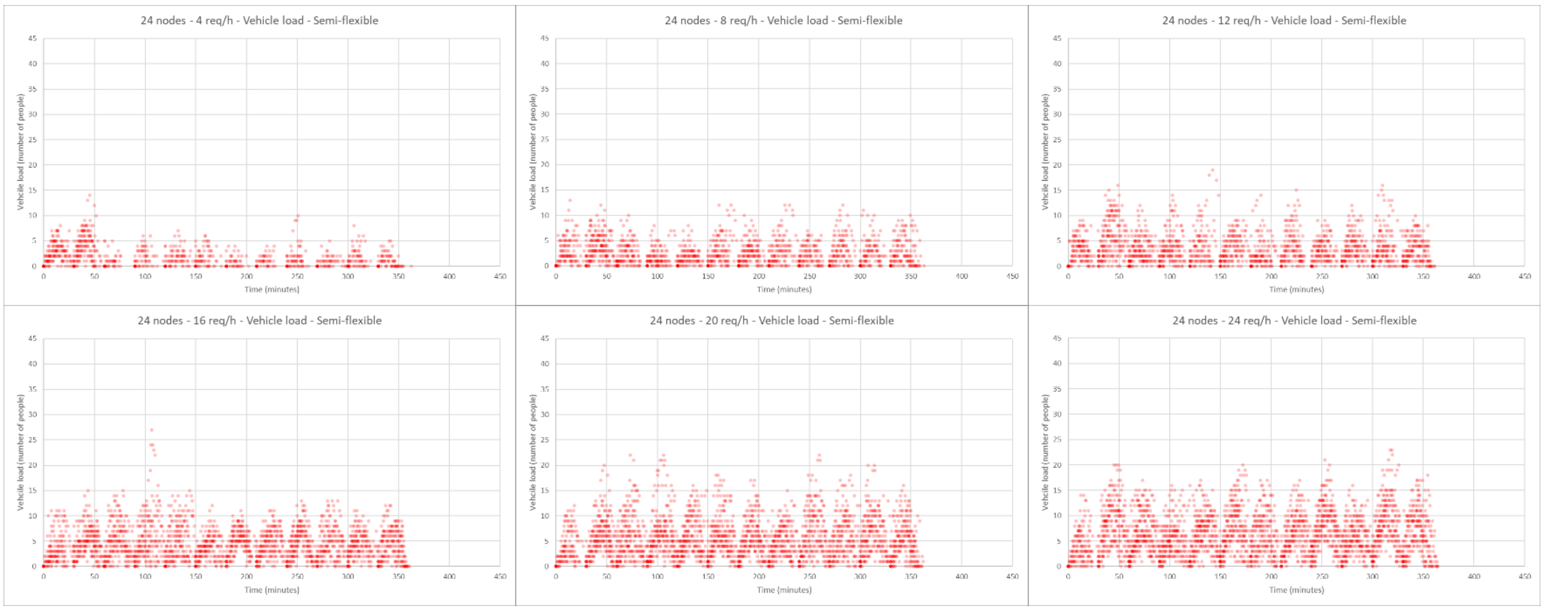
Appendix A.5: Vehicle loads over time, node density = 20 nodes, dynamic DRT



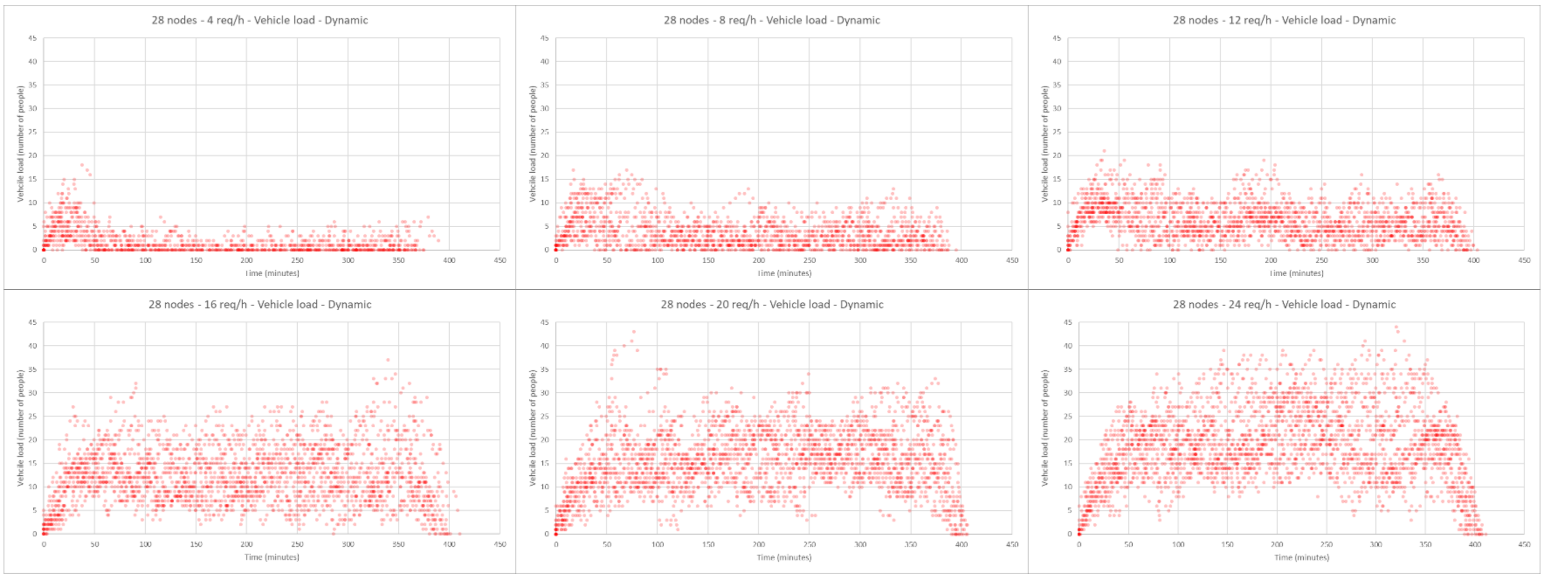
Appendix A.6: Vehicle loads over time, node density = 20 nodes, semi-flexible DRT



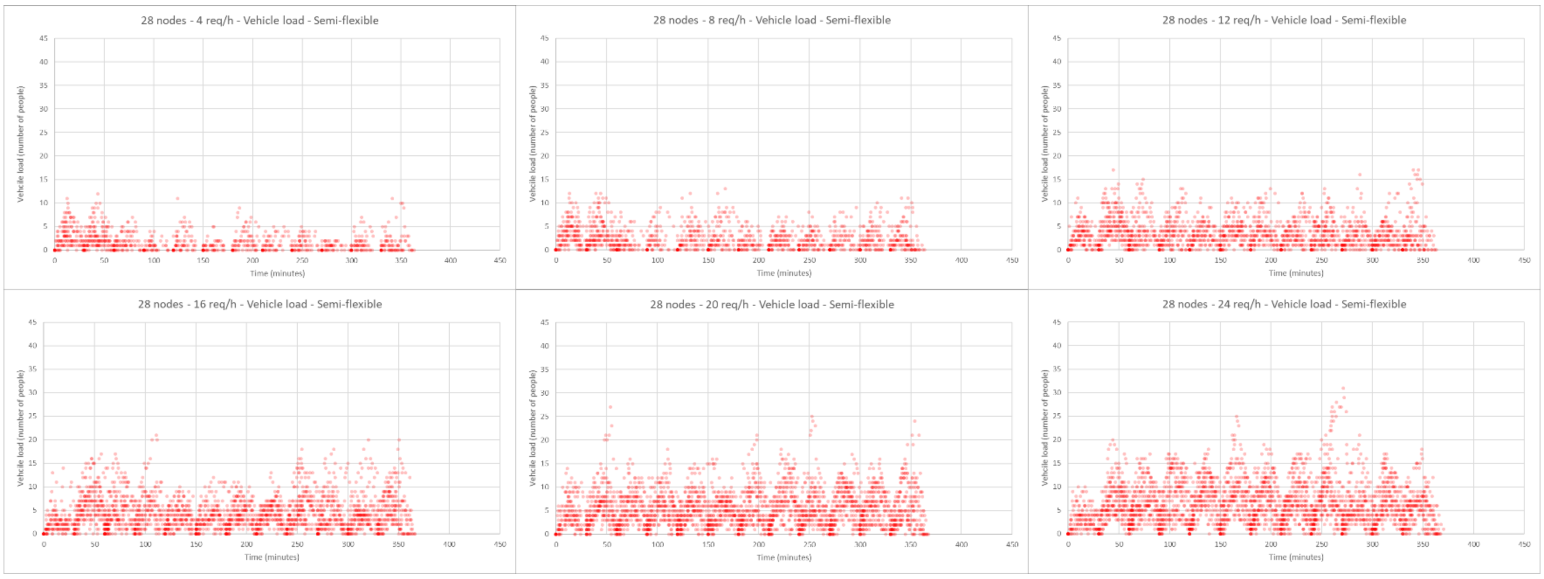
Appendix A.7: Vehicle loads over time, node density = 24 nodes, dynamic DRT



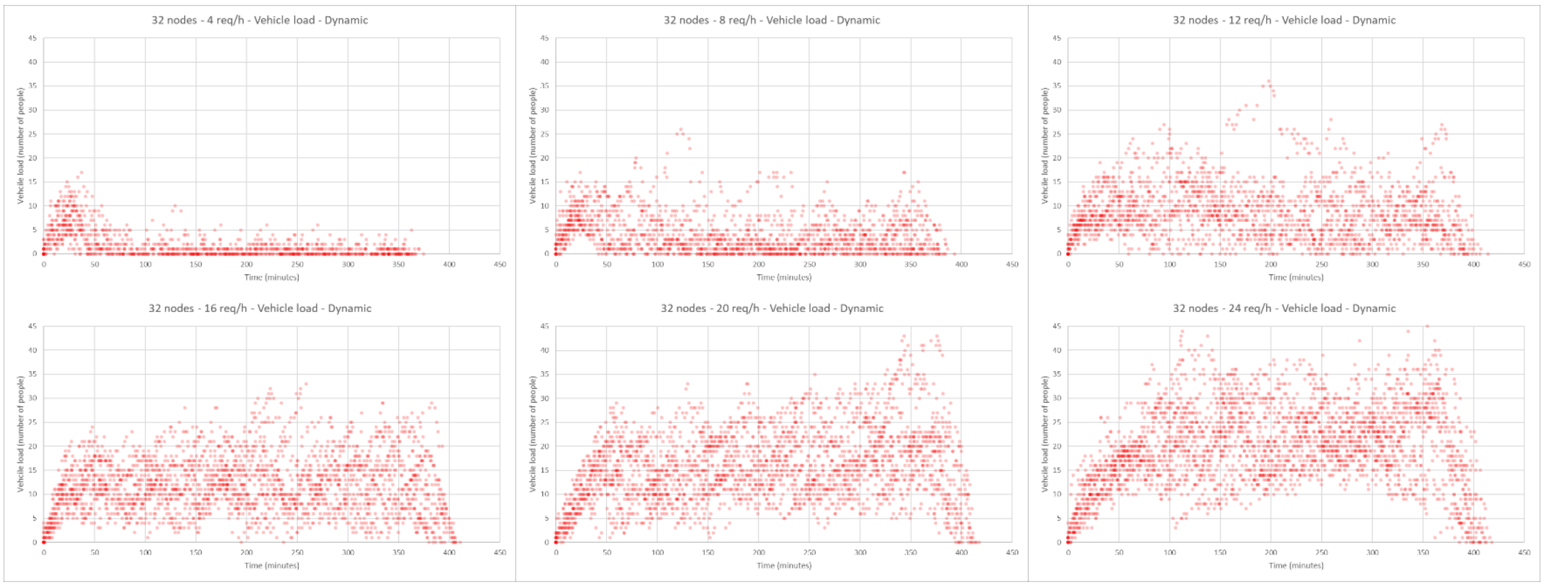
Appendix A.8: Vehicle loads over time, node density = 24 nodes, semi-flexible DRT



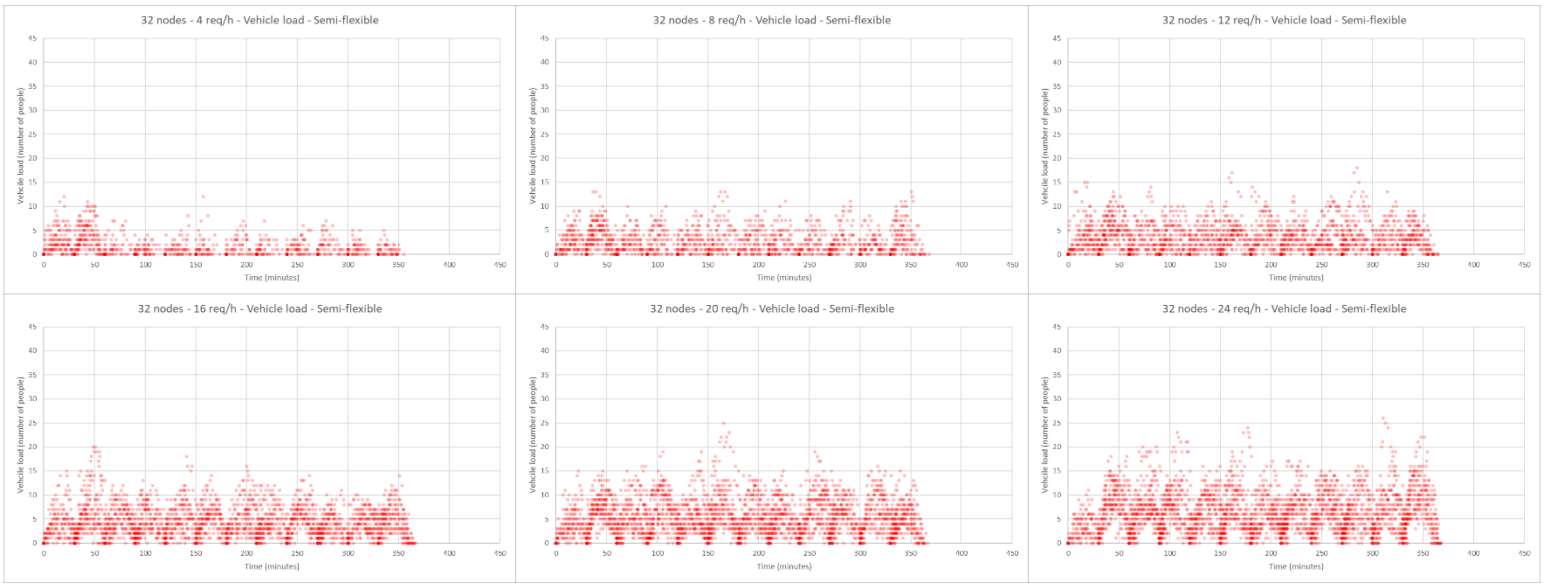
Appendix A.9: Vehicle loads over time, node density = 28 nodes, dynamic DRT



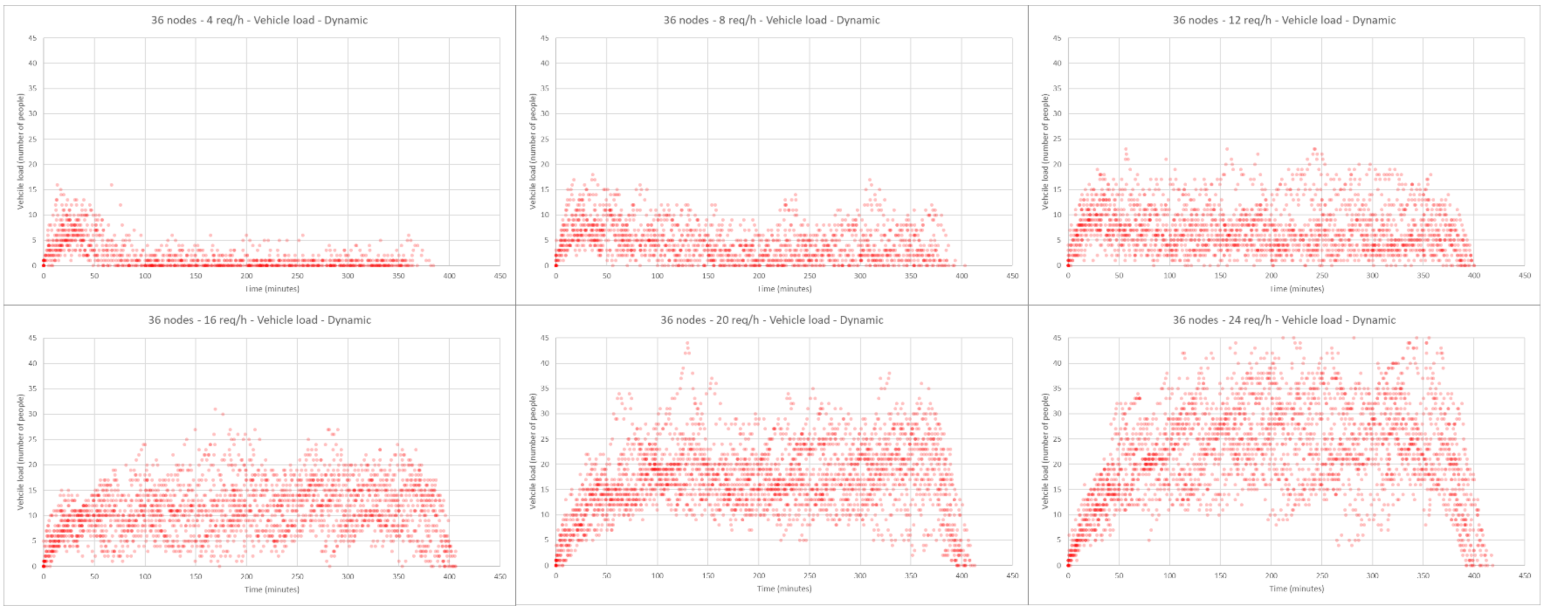
Appendix A.10: Vehicle loads over time, node density = 28 nodes, semi-flexible DRT



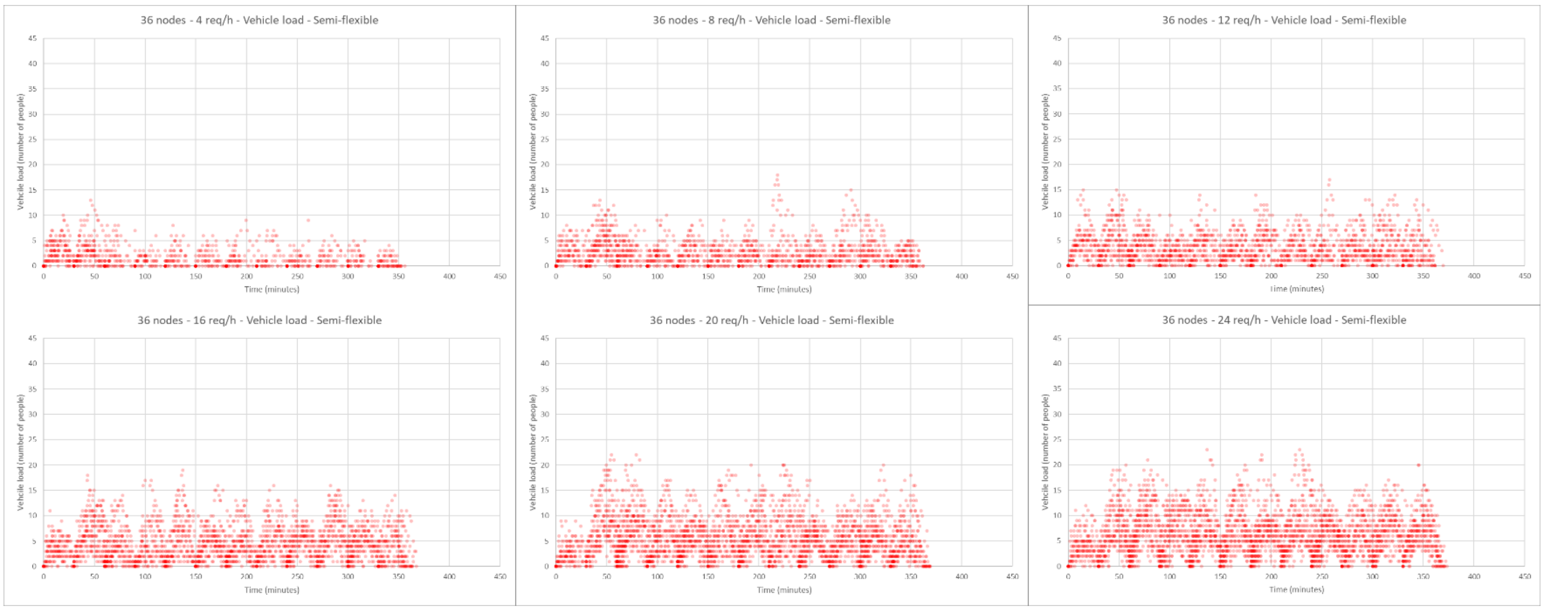
Appendix A.11: Vehicle loads over time, node density = 32 nodes, dynamic DRT



Appendix A.12: Vehicle loads over time, node density = 32 nodes, semi-flexible DRT

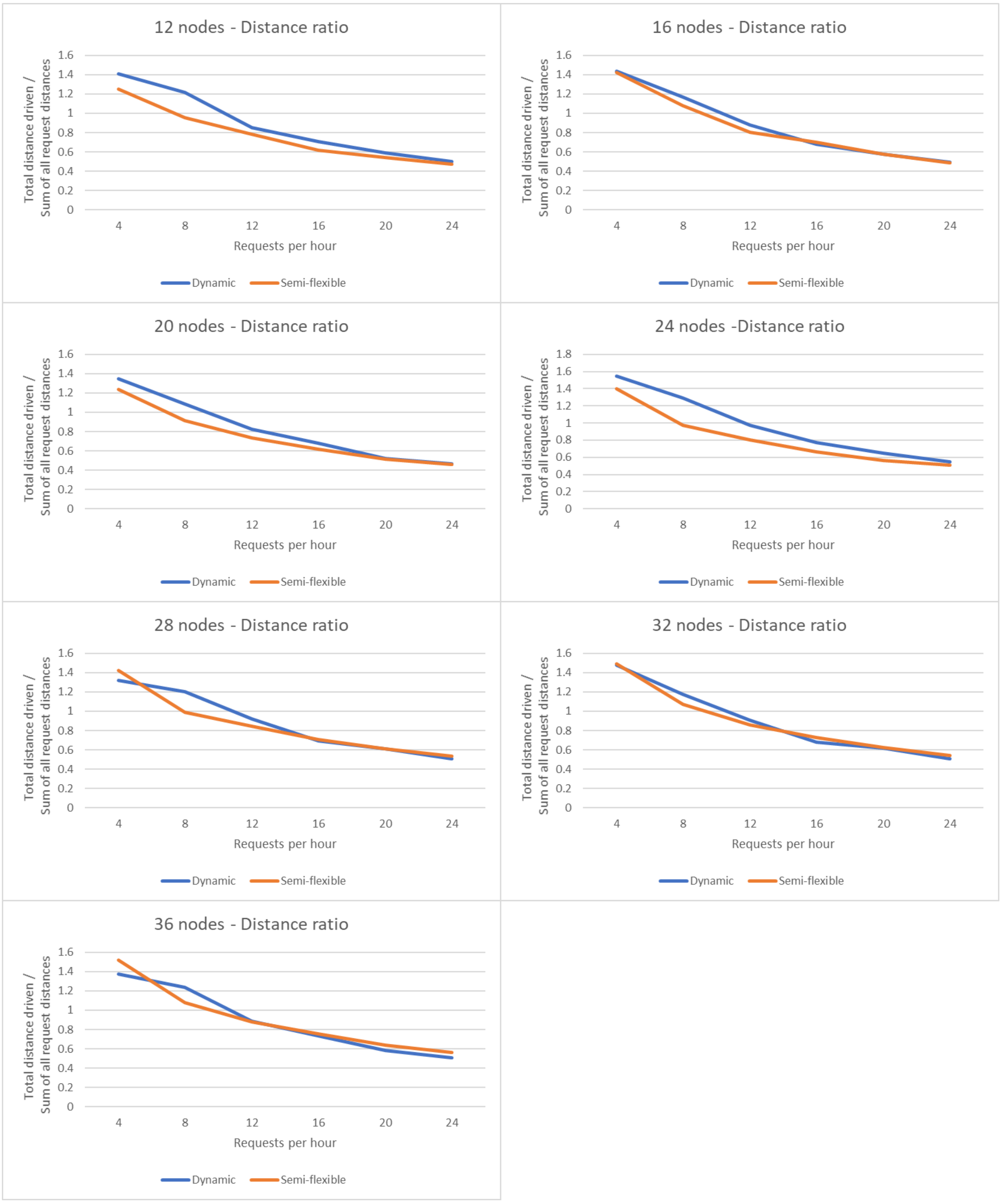


Appendix A.13: Vehicle loads over time, node density = 36 nodes, dynamic DRT



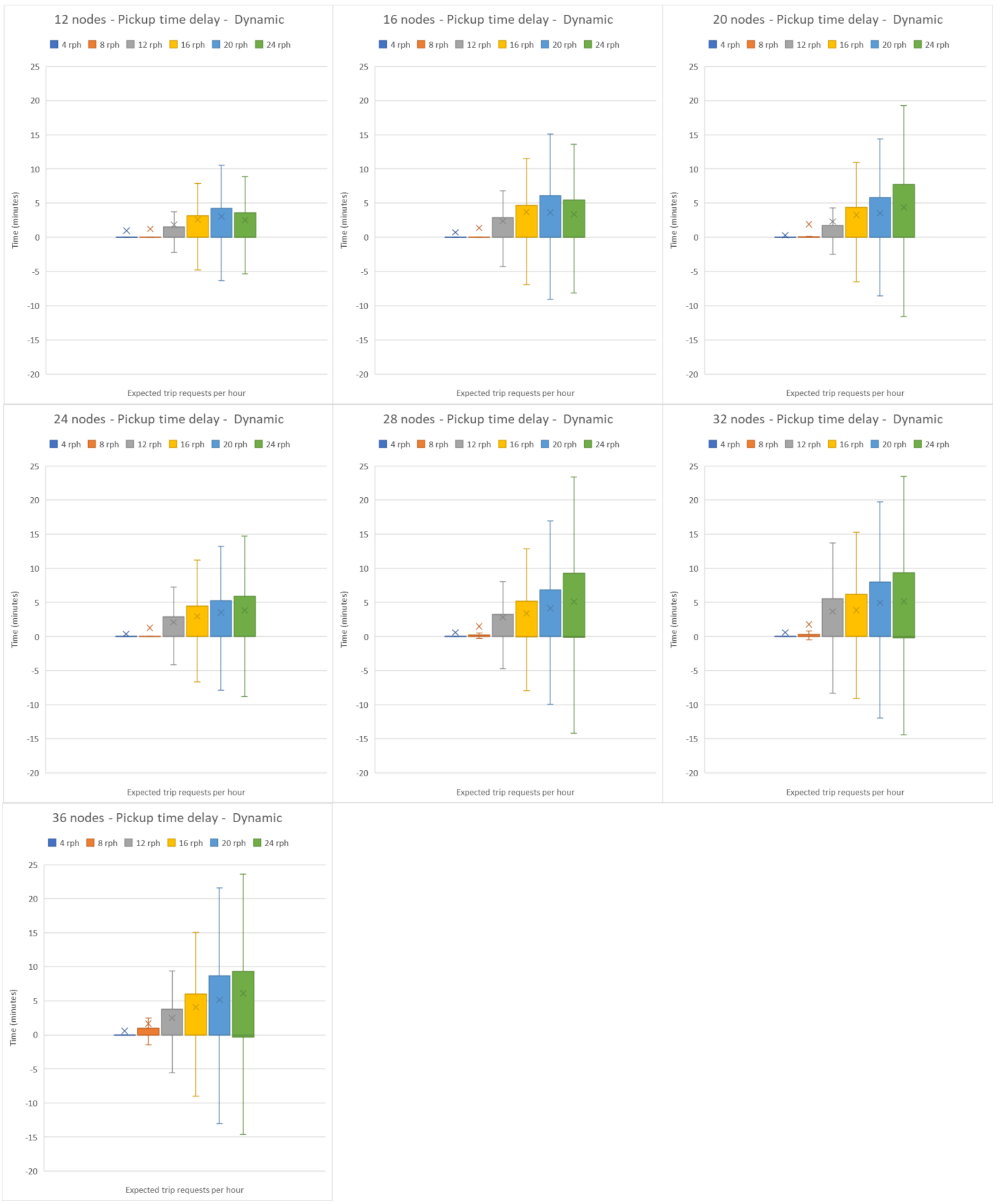
Appendix A.14: Vehicle loads over time, node density = 32 nodes, dynamic DRT

Appendix B: Distance ratios

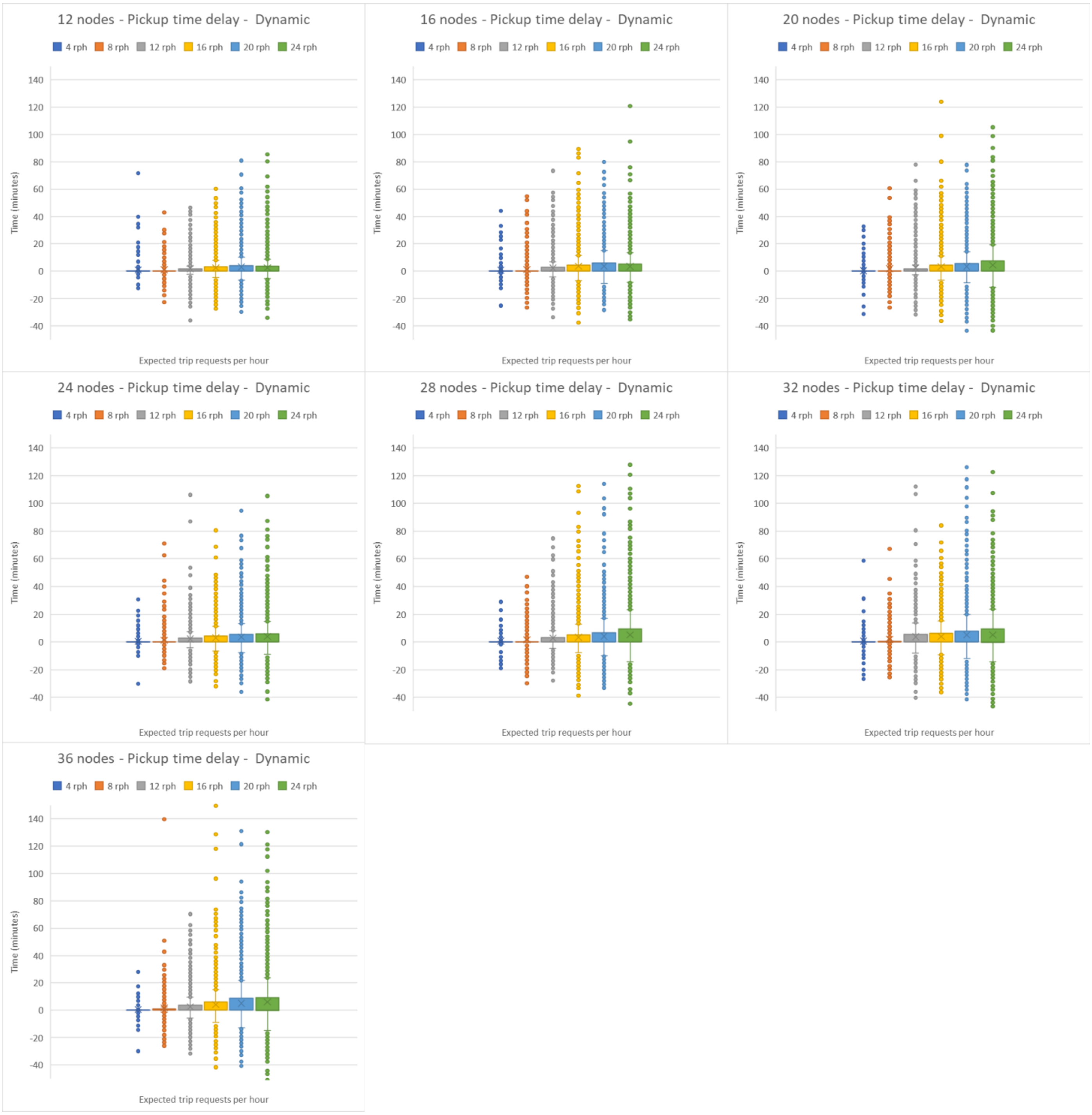


Appendix B.1: Average distance ratios (total distance driven by the vehicle divided by sum of all trip request distances) per node density.

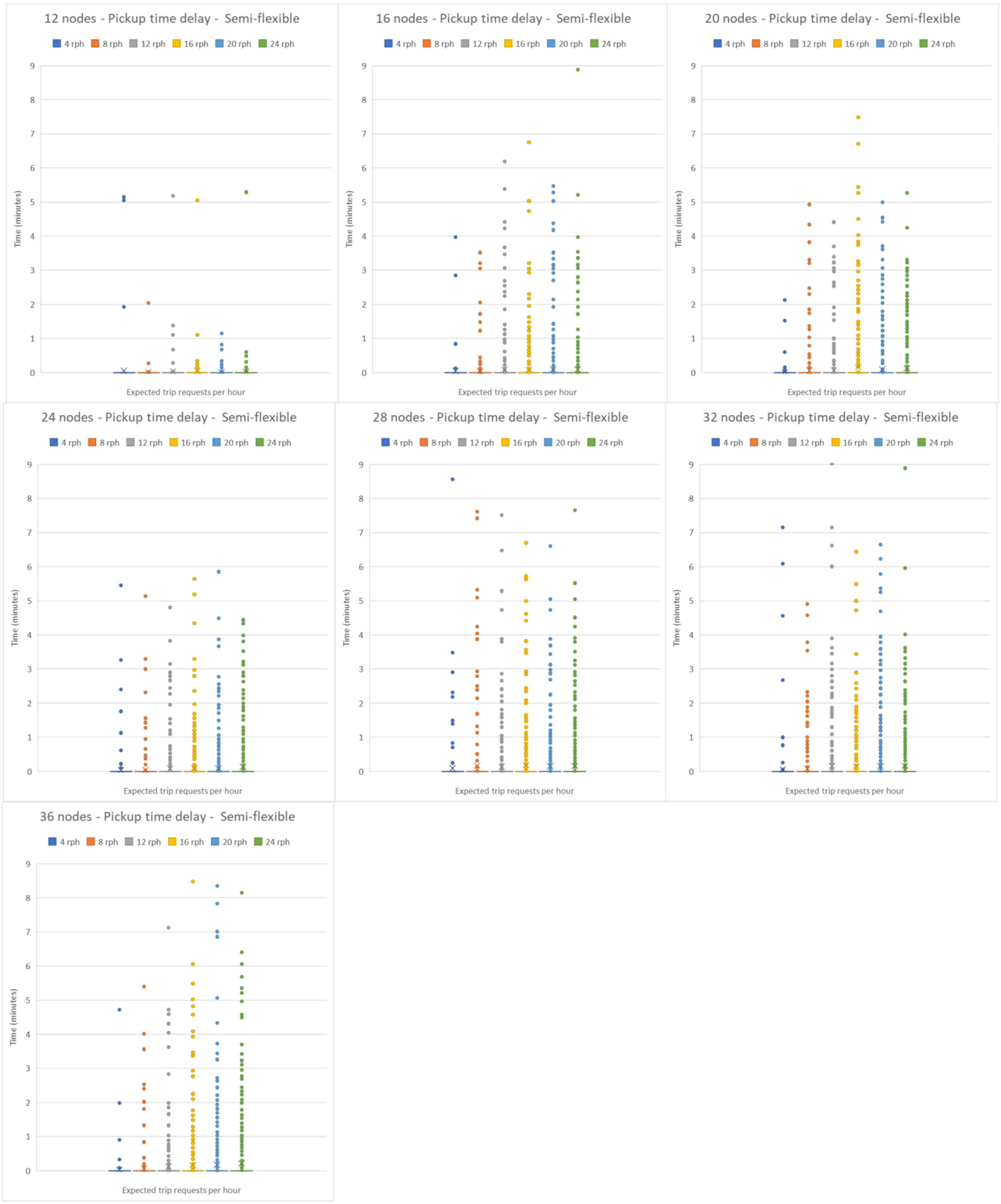
Appendix C: Pickup time delays



Appendix C.1: Pickup time delay distributions for each node density and demand level, dynamic DRT, without outliers.

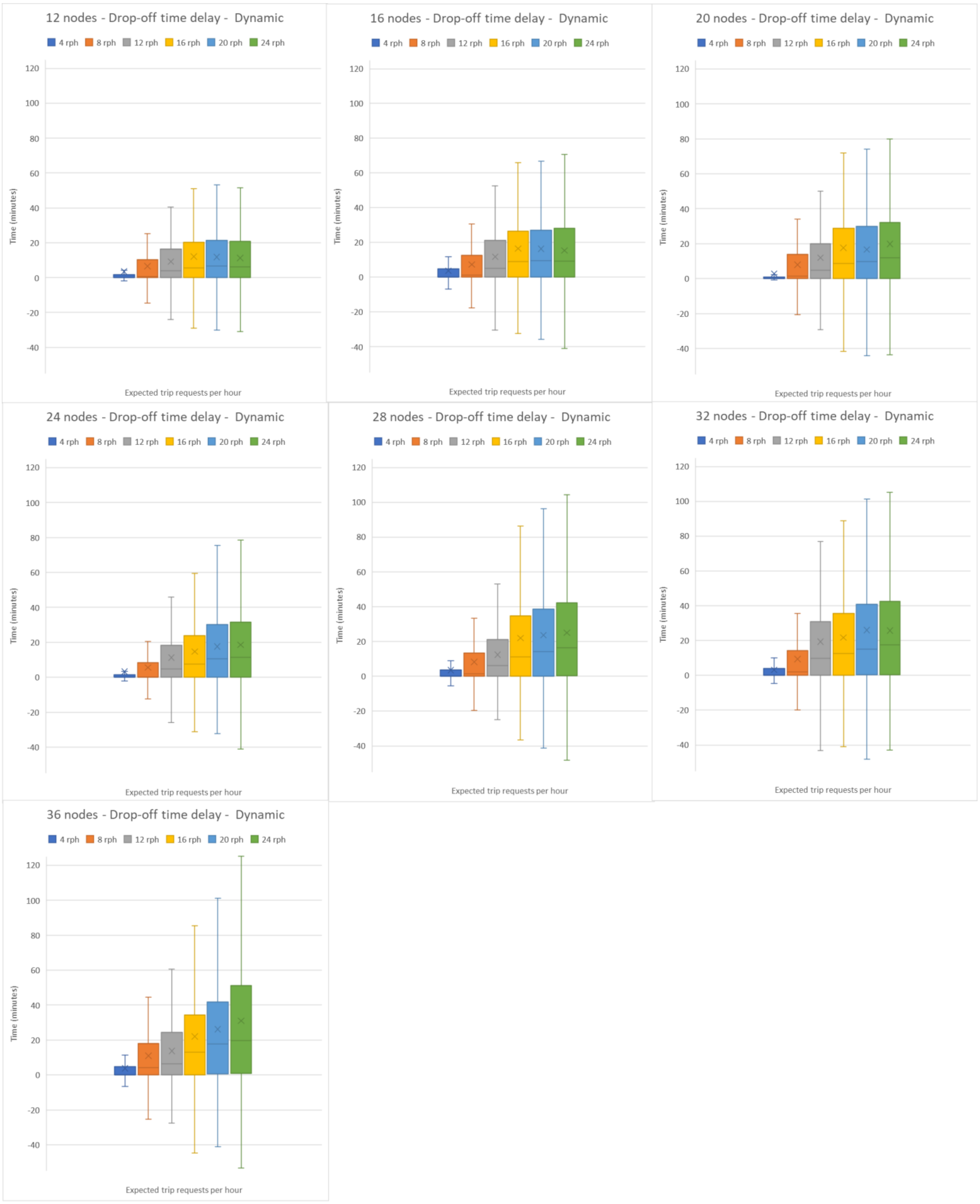


Appendix C.2: Pickup time delay distributions for each node density and demand level, dynamic DRT, with outliers.

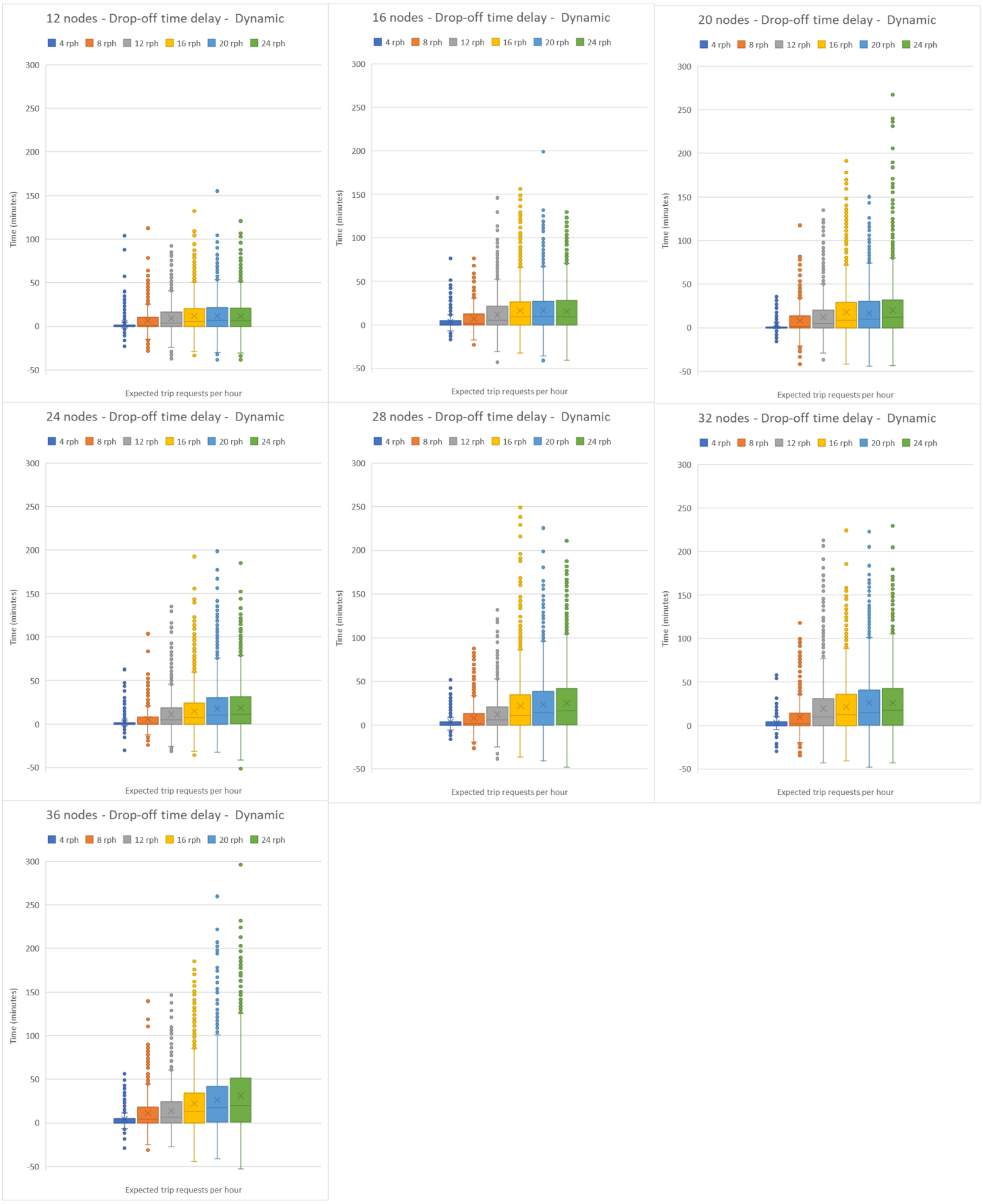


Appendix C.3: Pickup time delay distributions for each node density and demand level, semi-flexible DRT, with outliers.

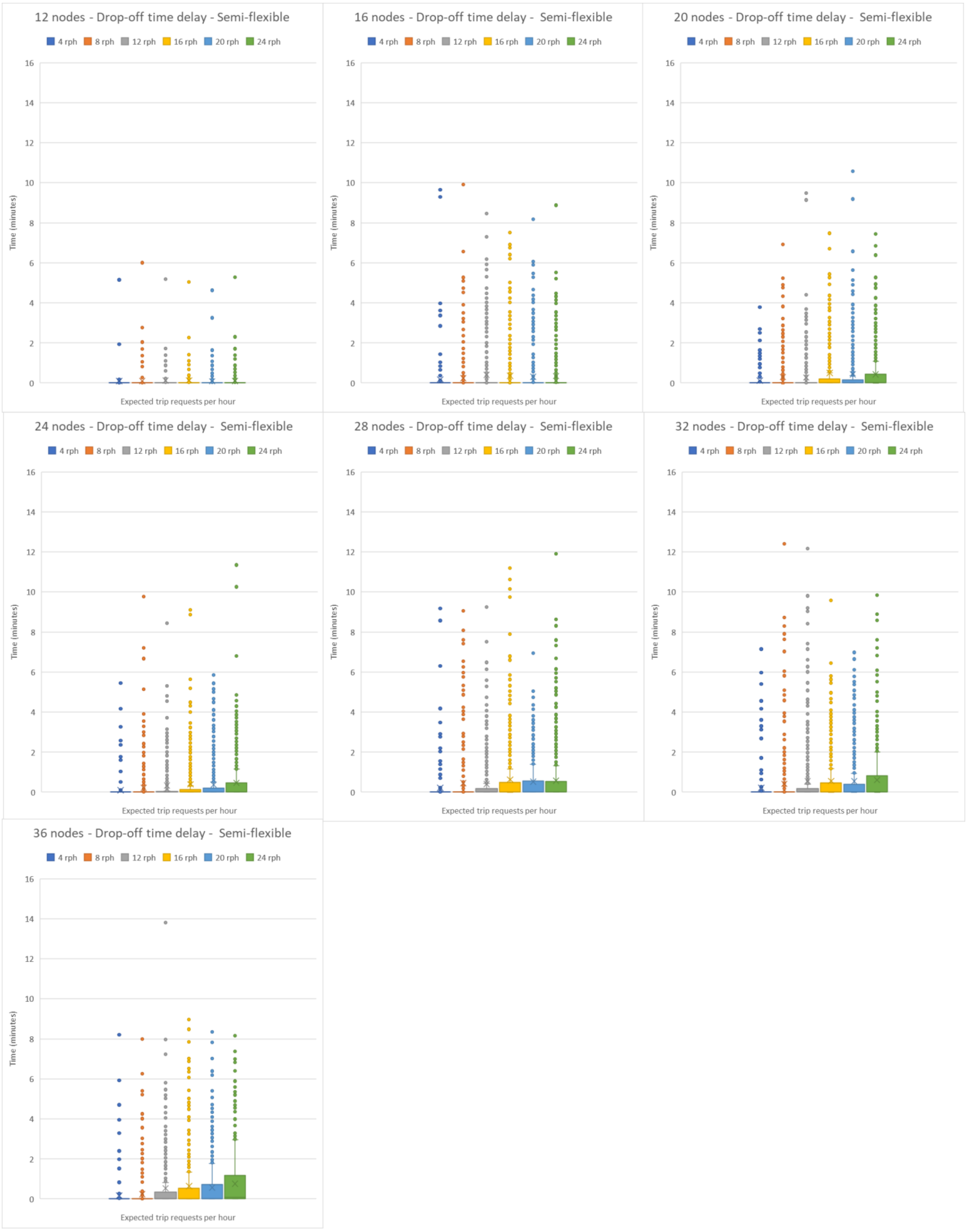
Appendix D: Drop-off time delays



Appendix D.1: Drop-off time delay distributions for each node density and demand level, dynamic DRT, without outliers.

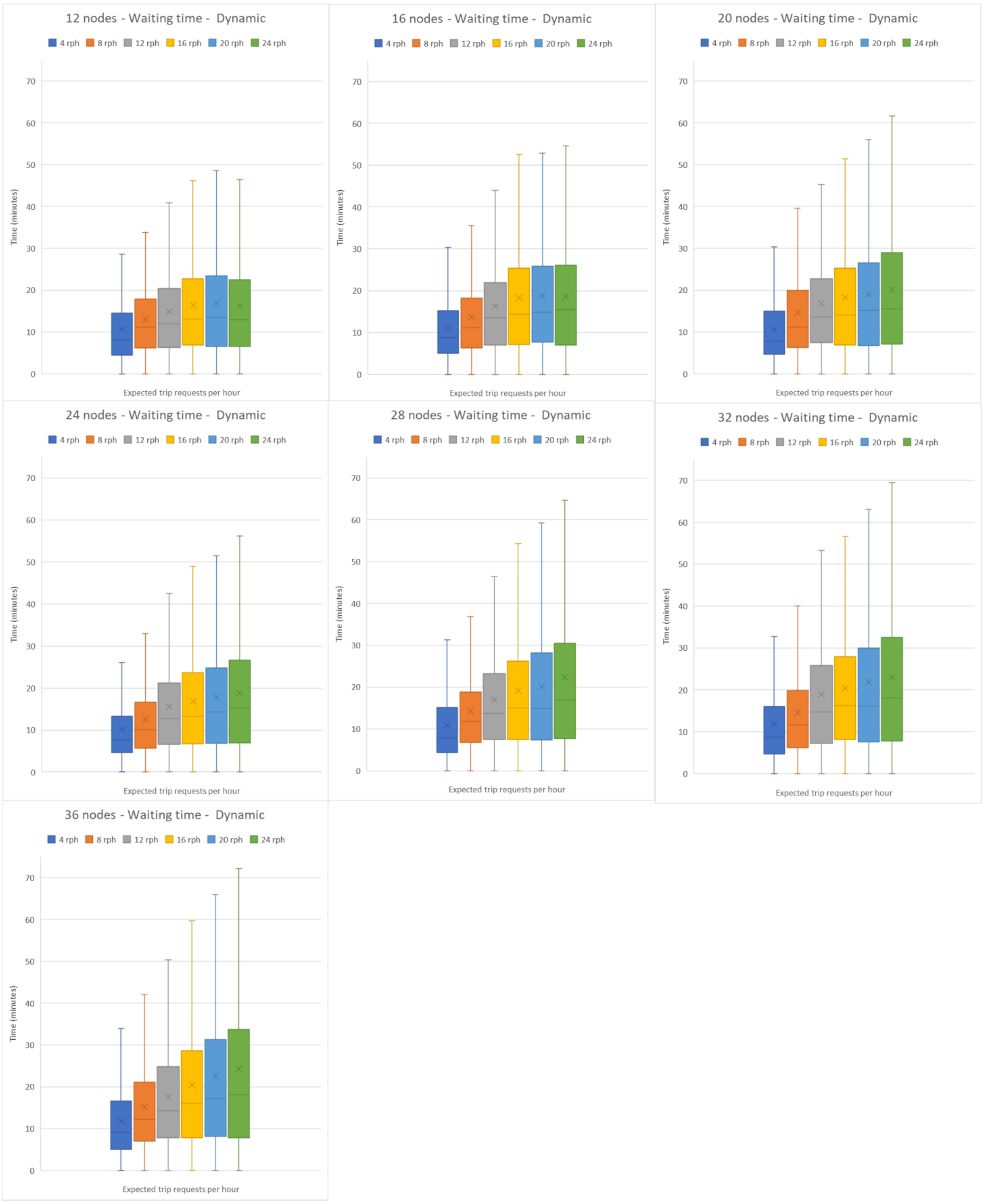


Appendix D.2: Drop-off time delay distributions for each node density and demand level, dynamic DRT, with outliers.

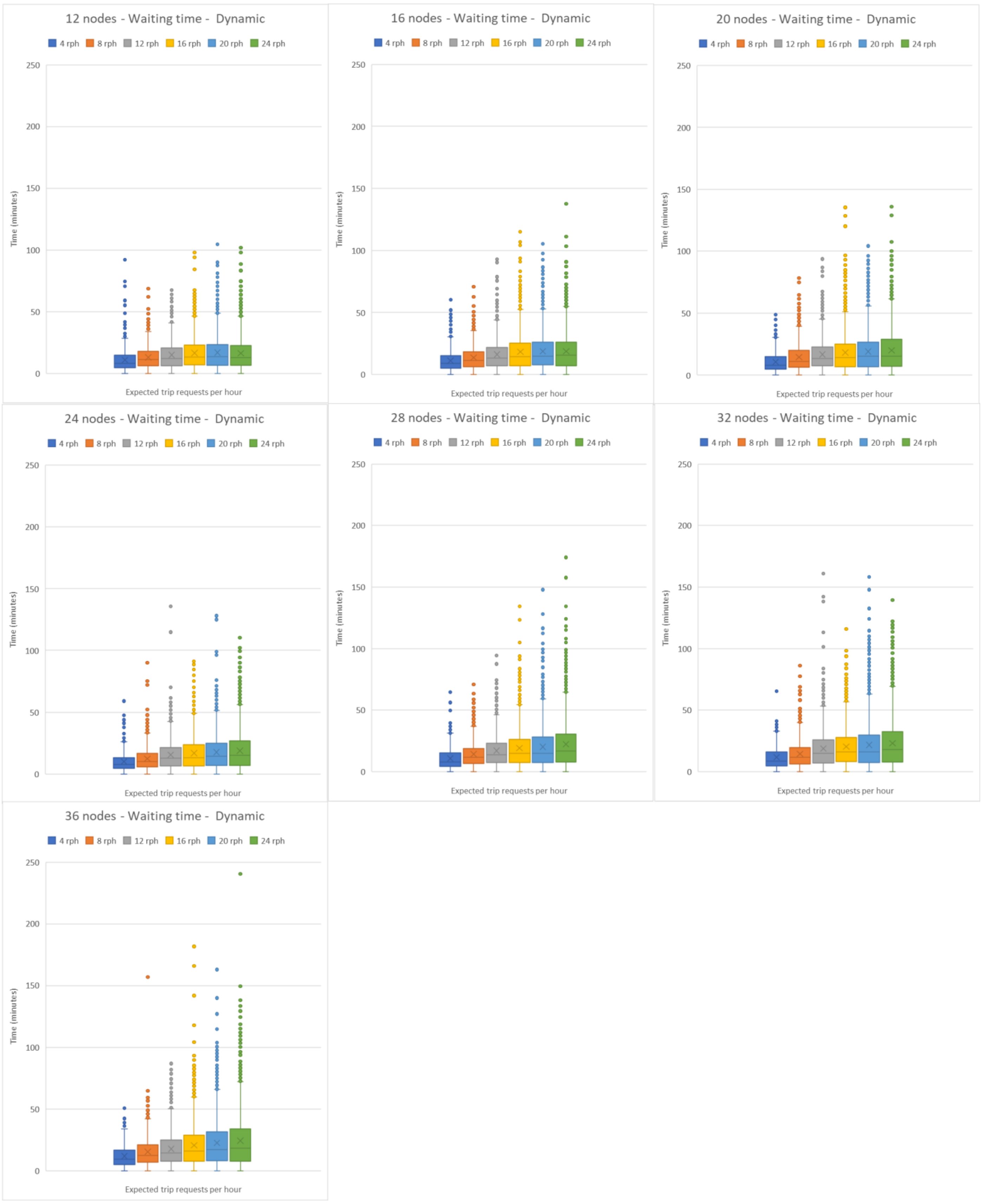


Appendix D.3: Drop-off time delay distributions for each node density and demand level, semi-flexible DRT, with outliers.

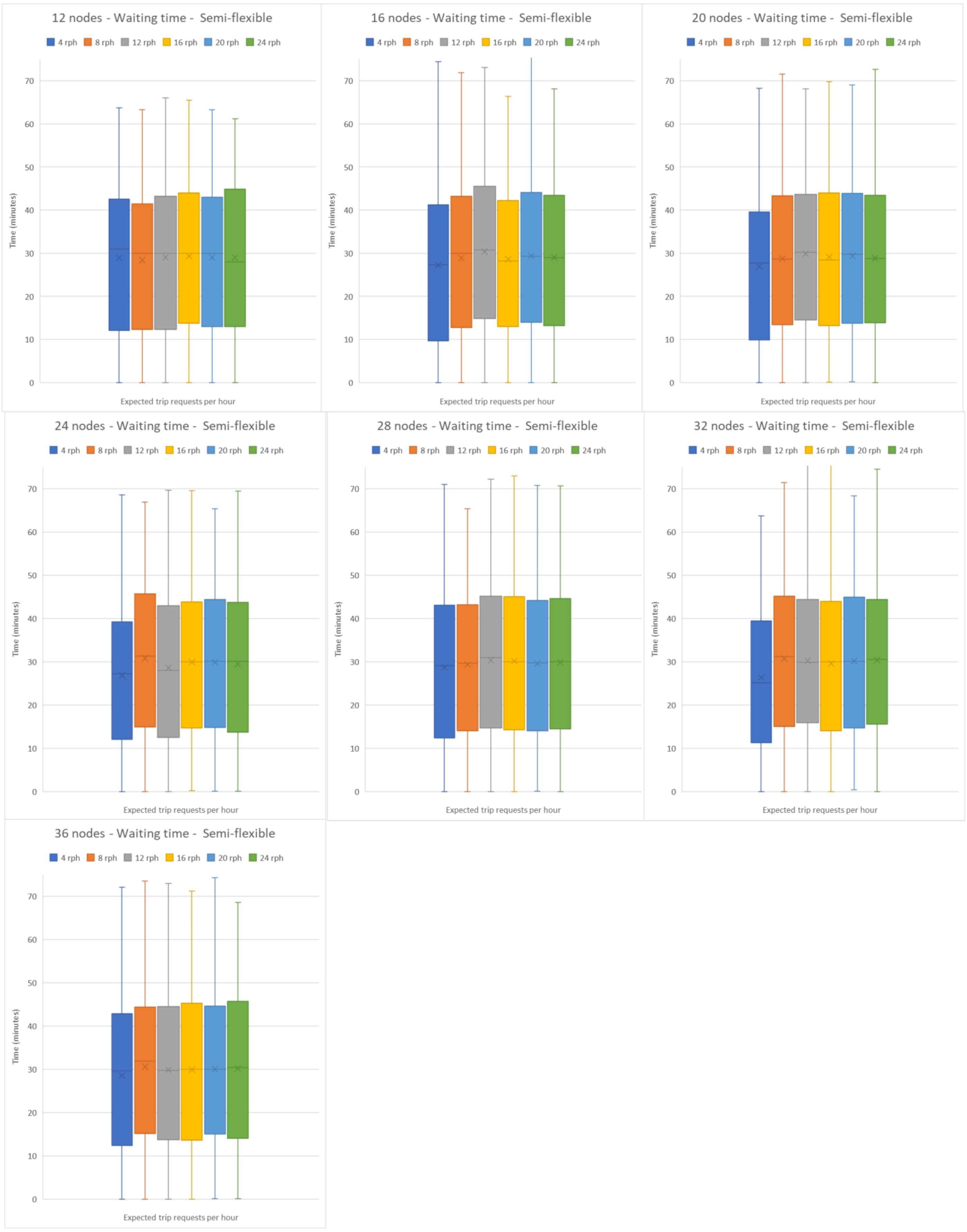
Appendix E: Waiting times



Appendix E.1: Waiting time distributions for each node density and demand level, dynamic DRT, without outliers.

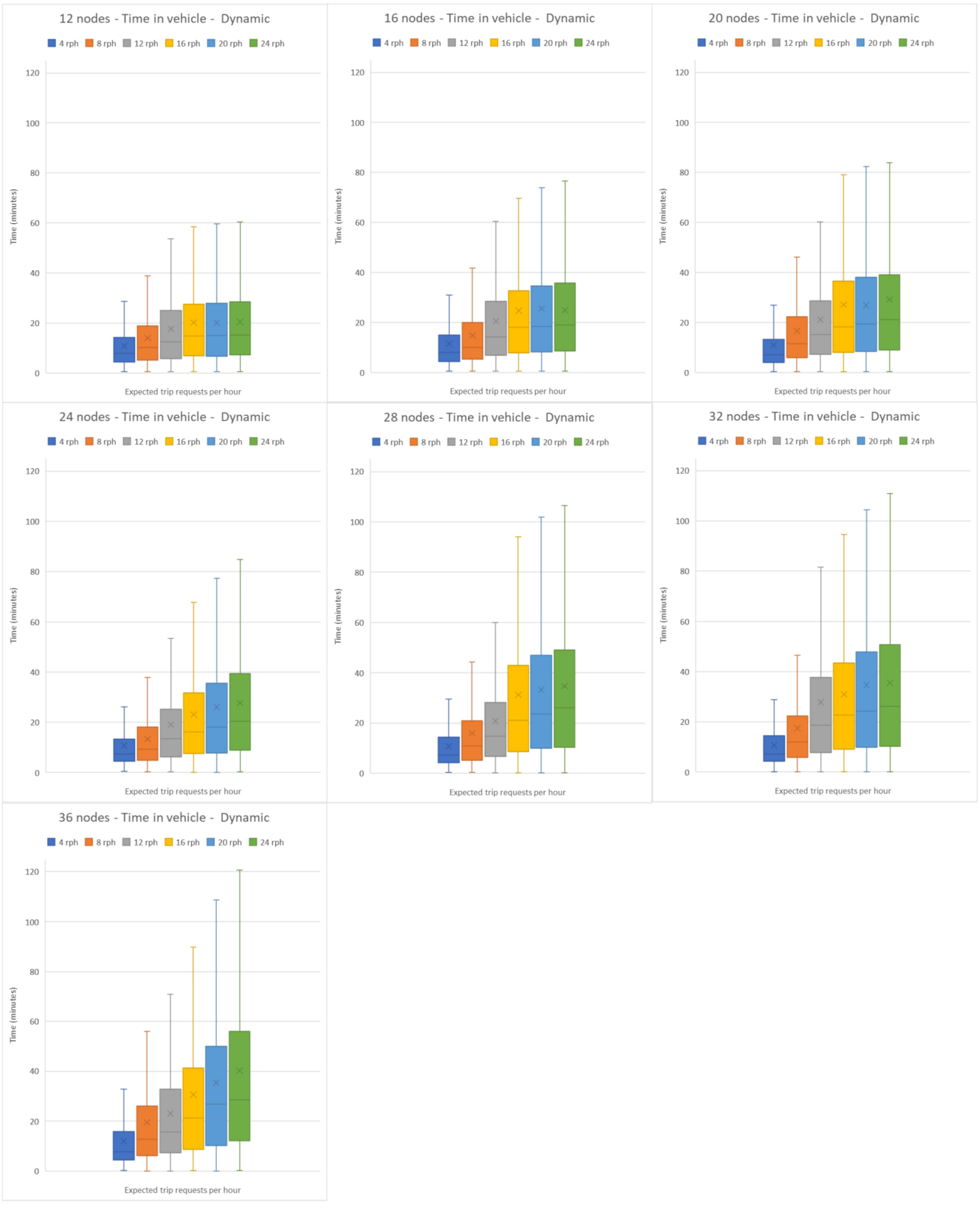


Appendix E.2: Waiting time distributions for each node density and demand level, dynamic DRT, with outliers.

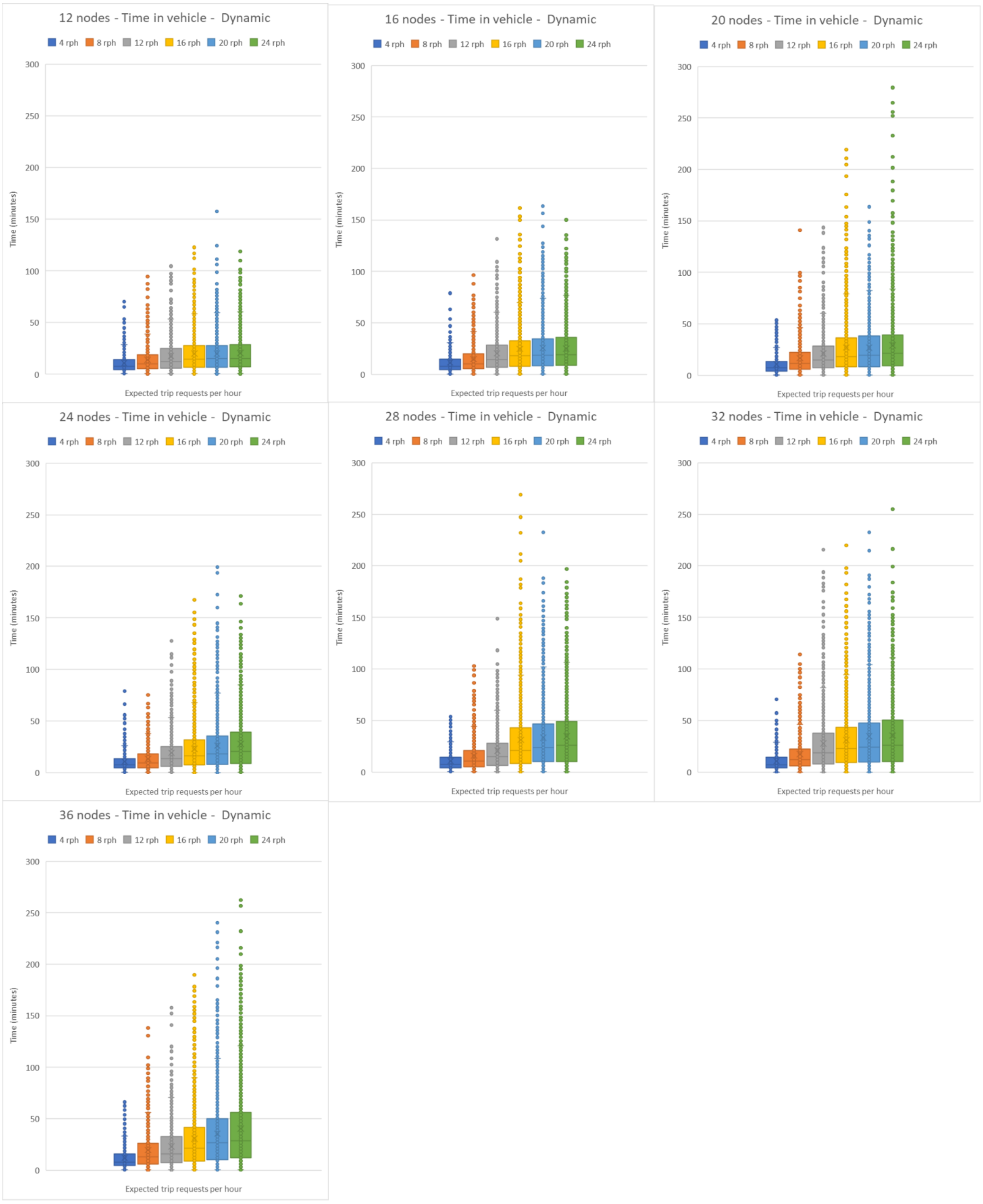


Appendix E.3: Waiting time distributions for each node density and demand level, semi-flexible DRT. Does not contain outliers.

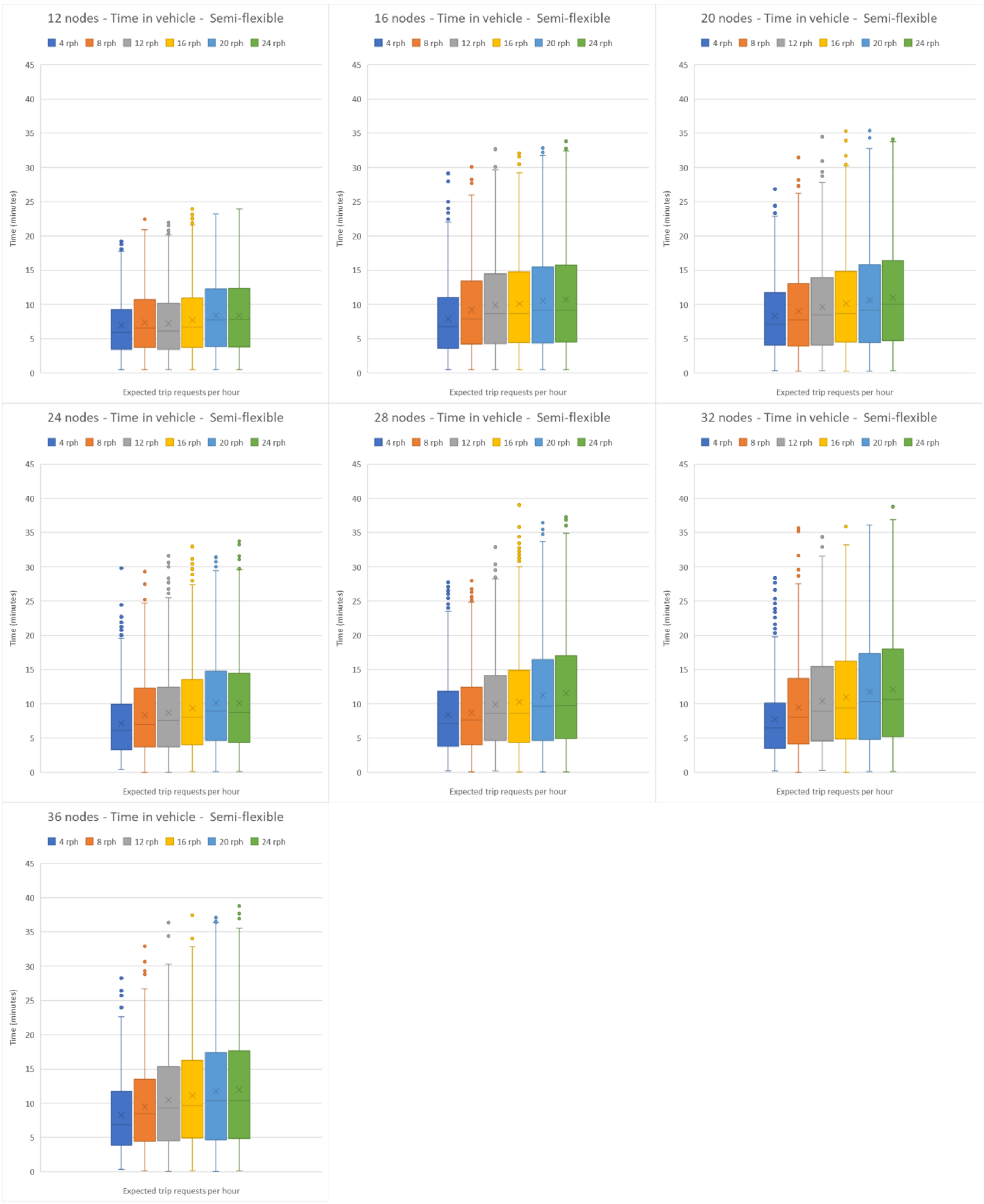
Appendix F: Trip durations



Appendix F.1: Trip duration distributions for each node density and demand level, dynamic DRT, without outliers.

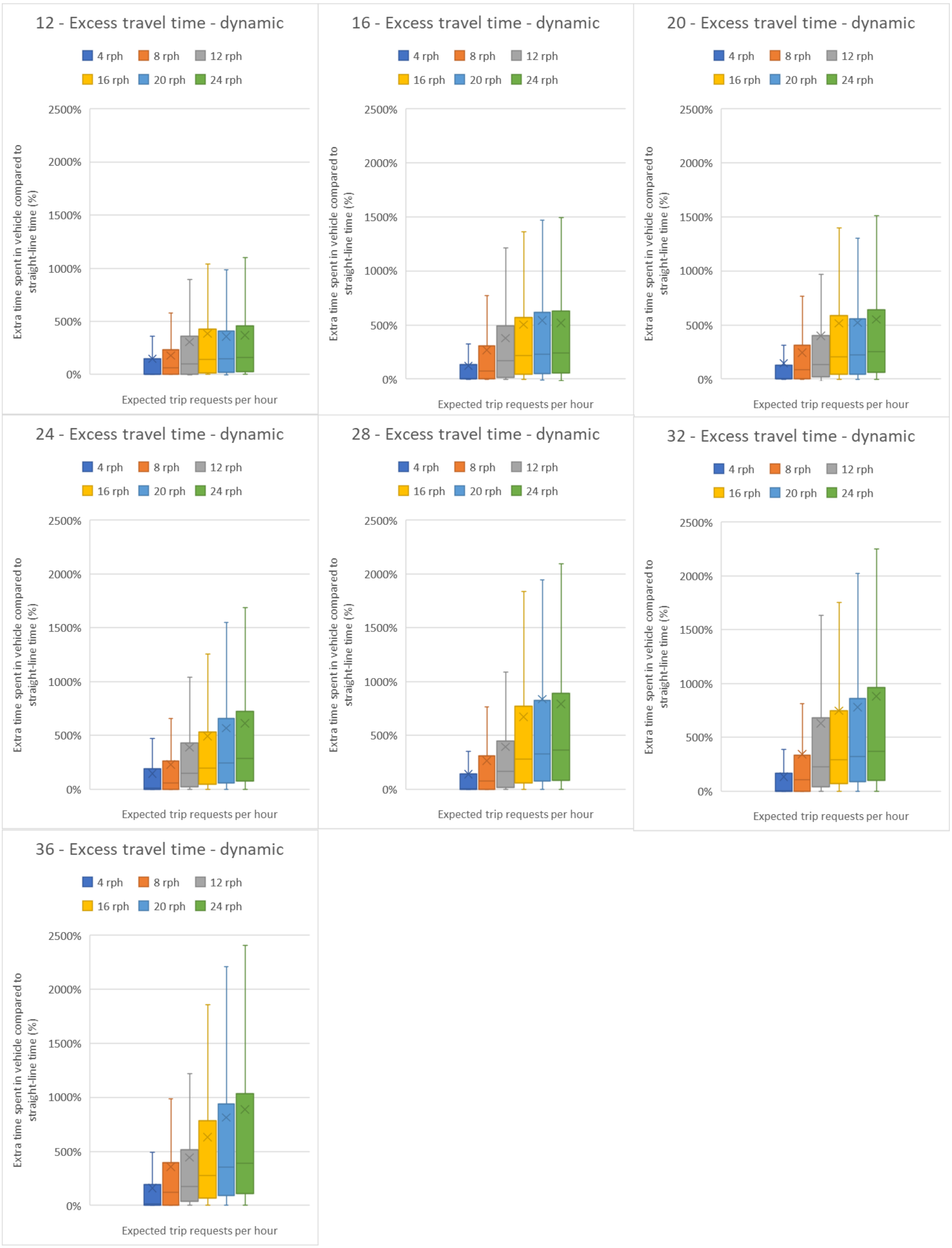


Appendix F.2: Trip duration distributions for each node density and demand level, dynamic DRT, with outliers.

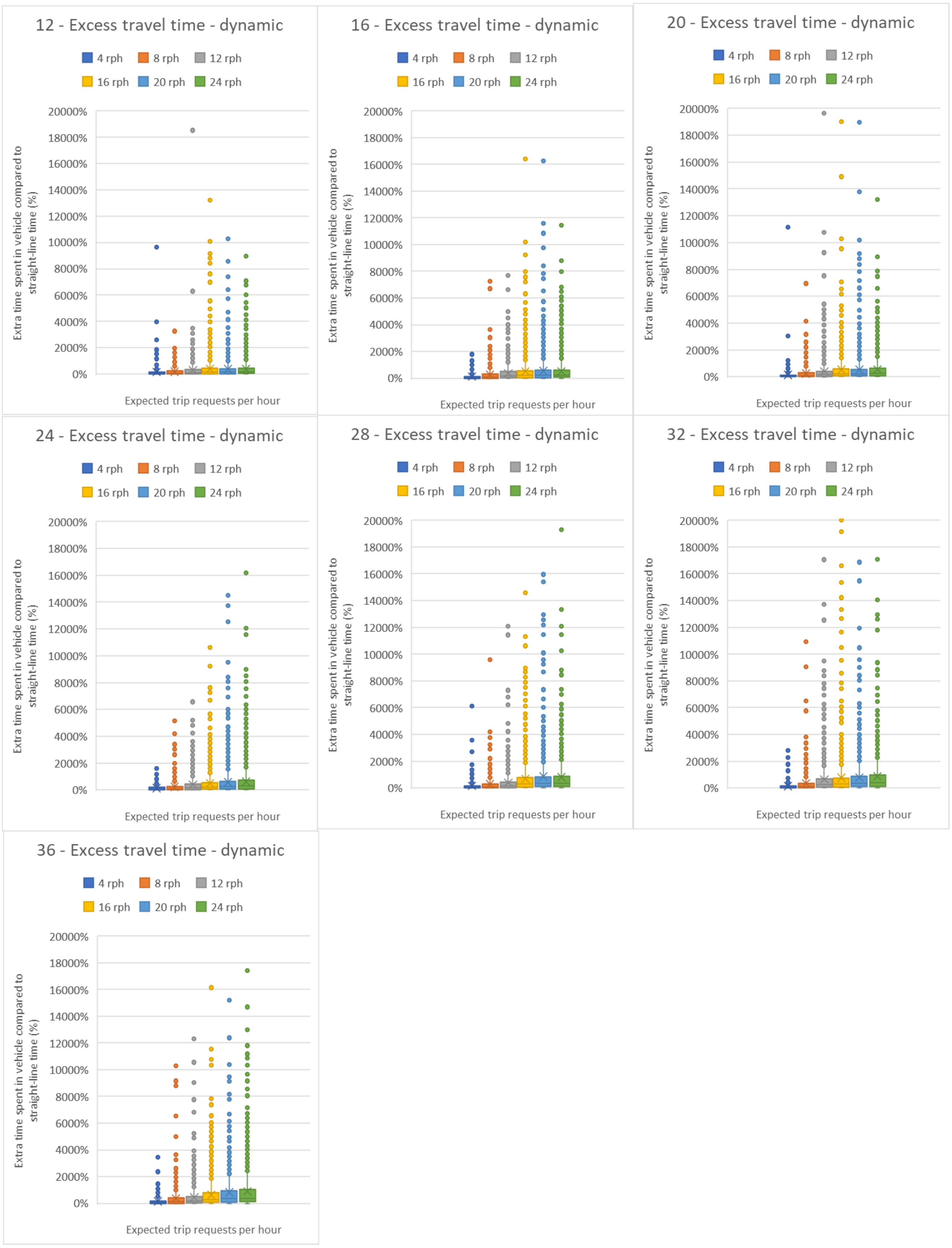


Appendix F.3: Trip duration distributions for each node density and demand level, semi-flexible DRT, with outliers.

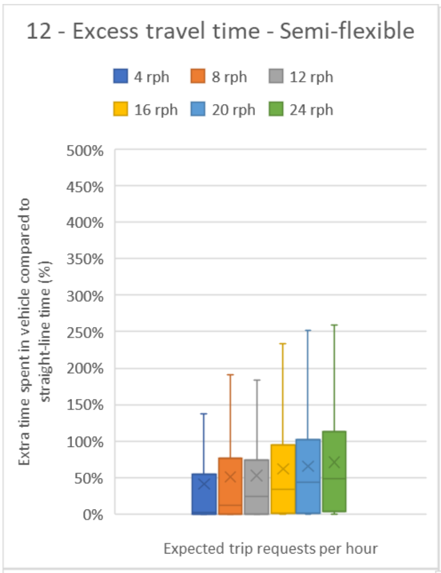
Appendix G: Excess travel times

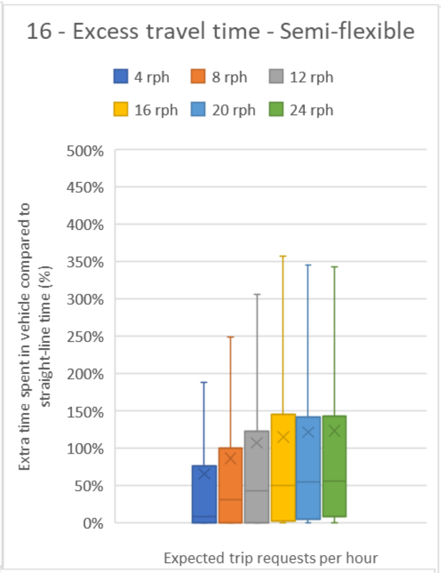


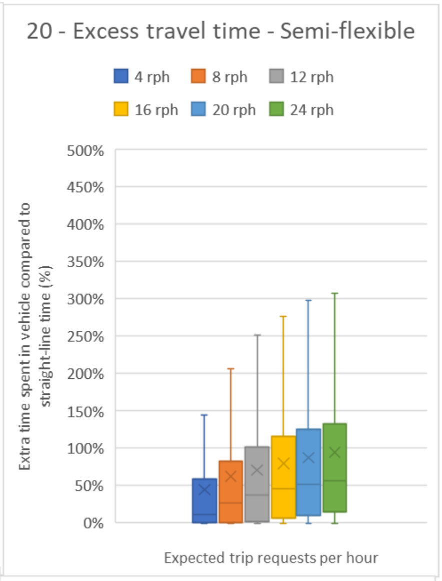
Appendix G.1: Excess travel time distributions for each node density and demand level, dynamic DRT, without outliers.

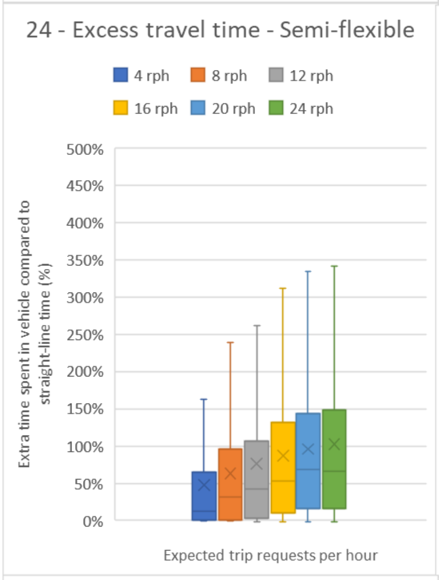


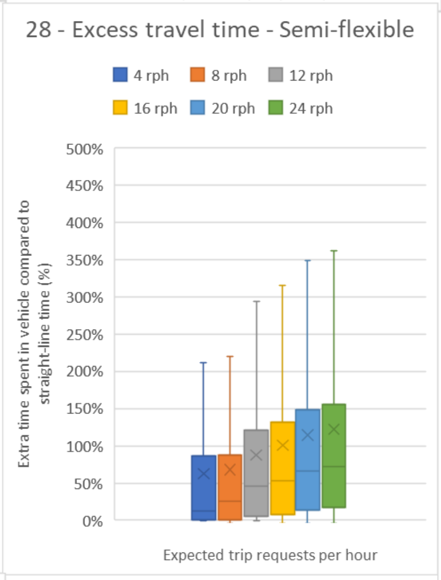
Appendix G.2: Excess travel time distributions for each node density and demand level, dynamic DRT, with outliers.

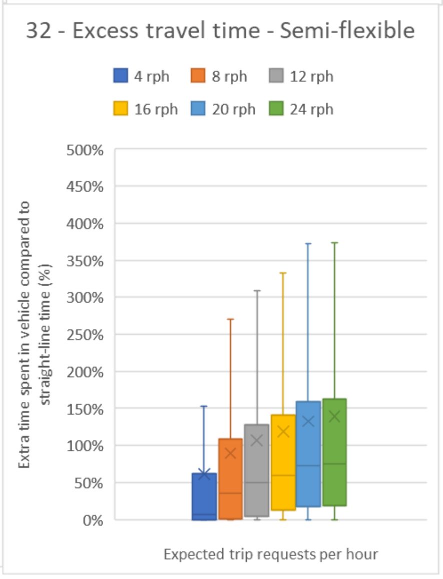


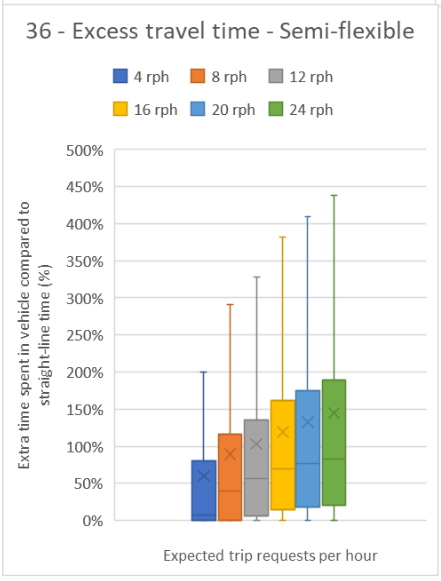




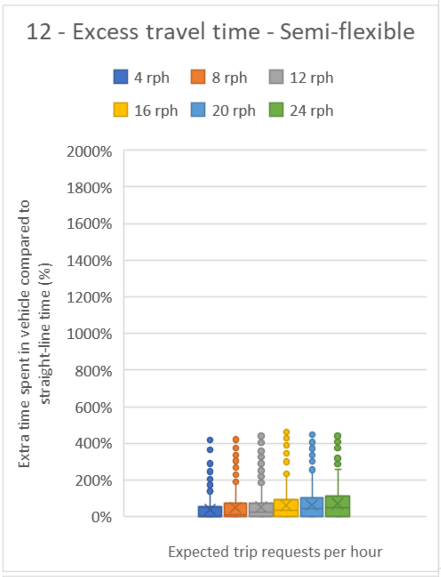


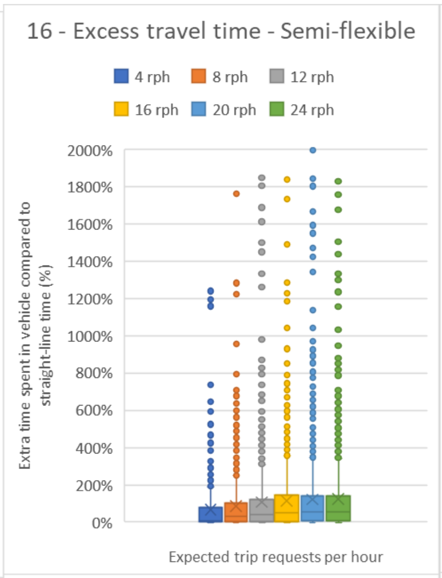


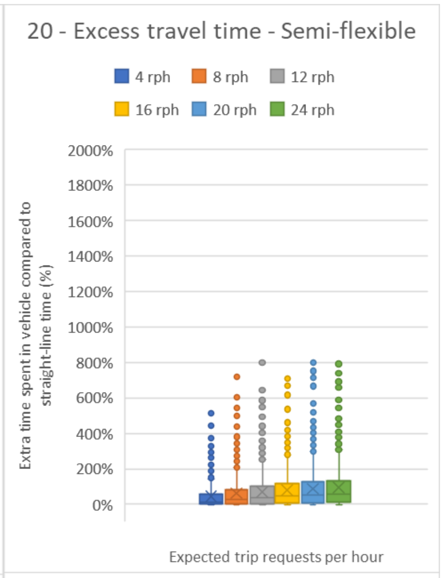


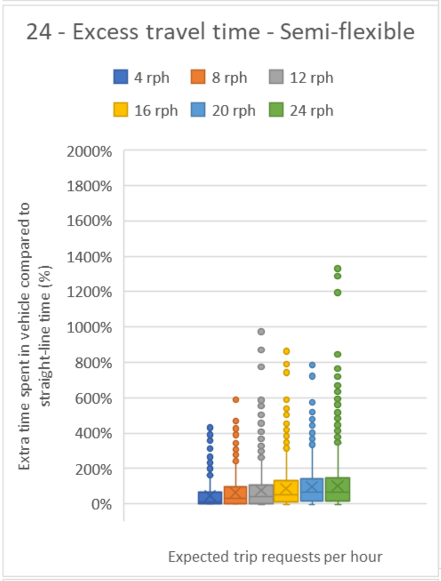


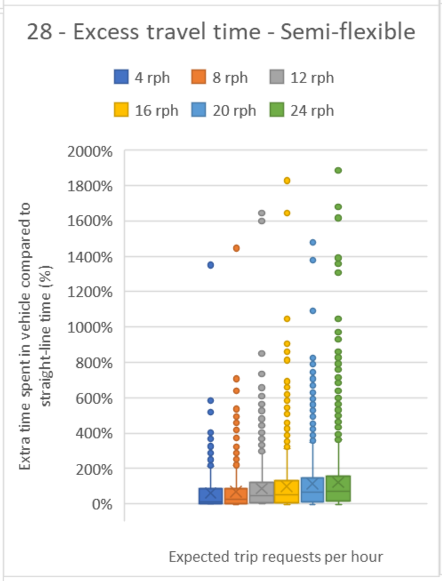
Appendix G.3: Excess travel time distributions for each node density and demand level, semi-flexible DRT, without outliers.

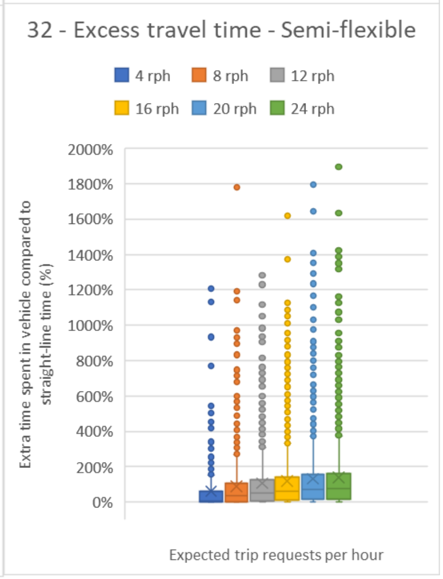


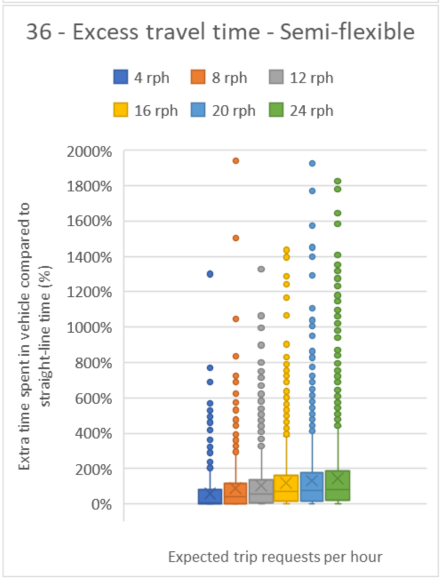






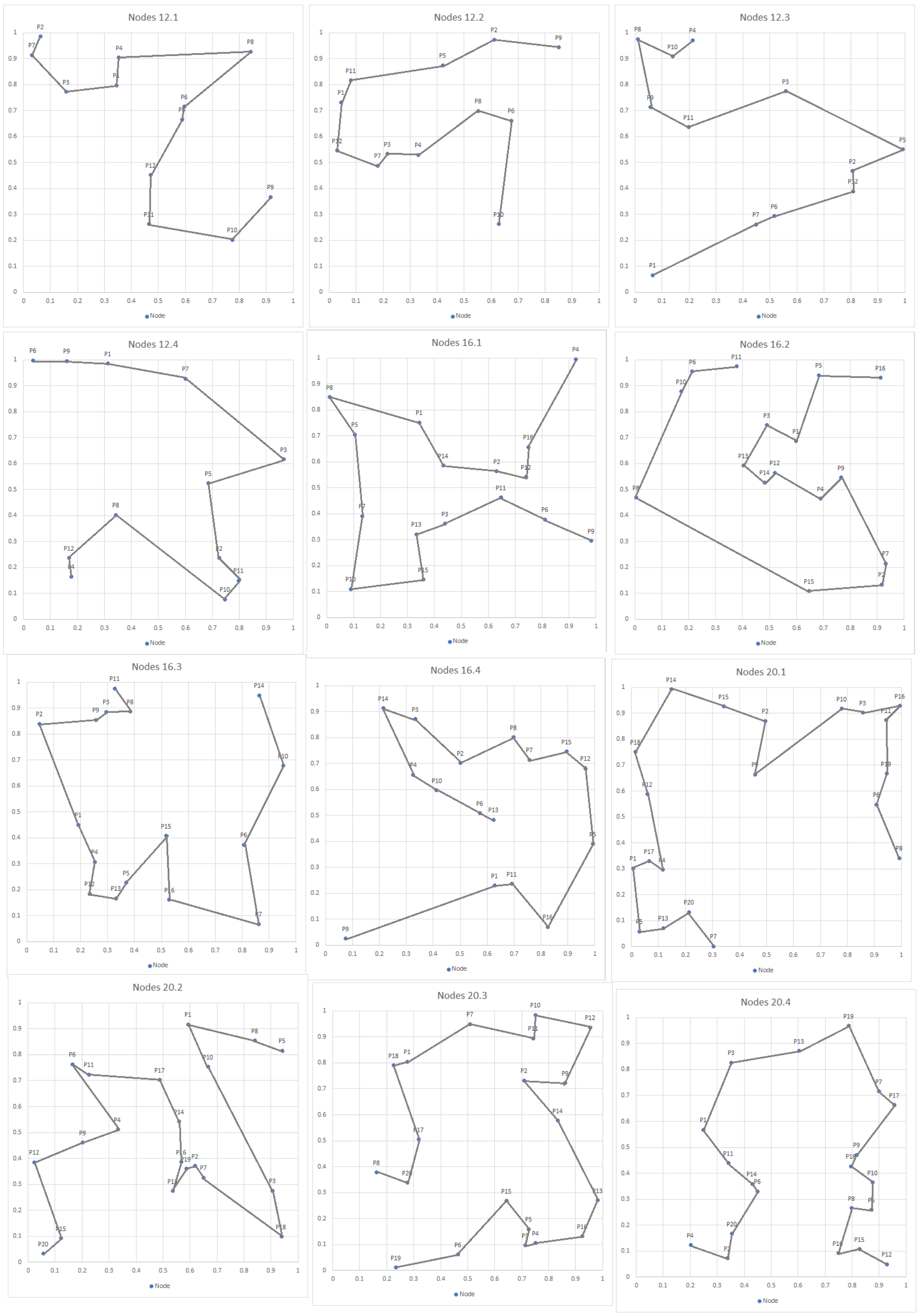




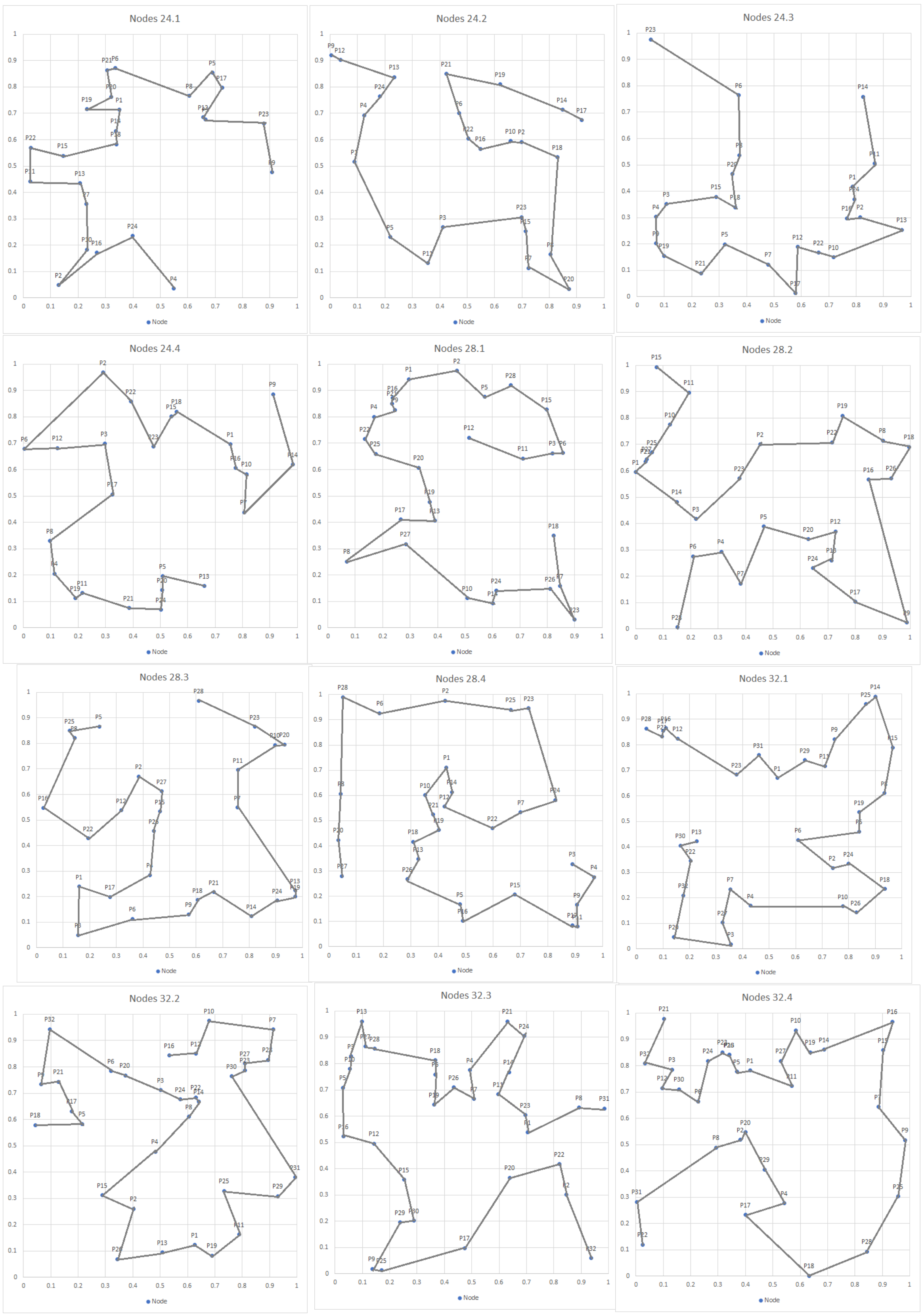


Appendix G.4: Excess travel time distributions for each node density and demand level, semi-flexible DRT, with outliers.

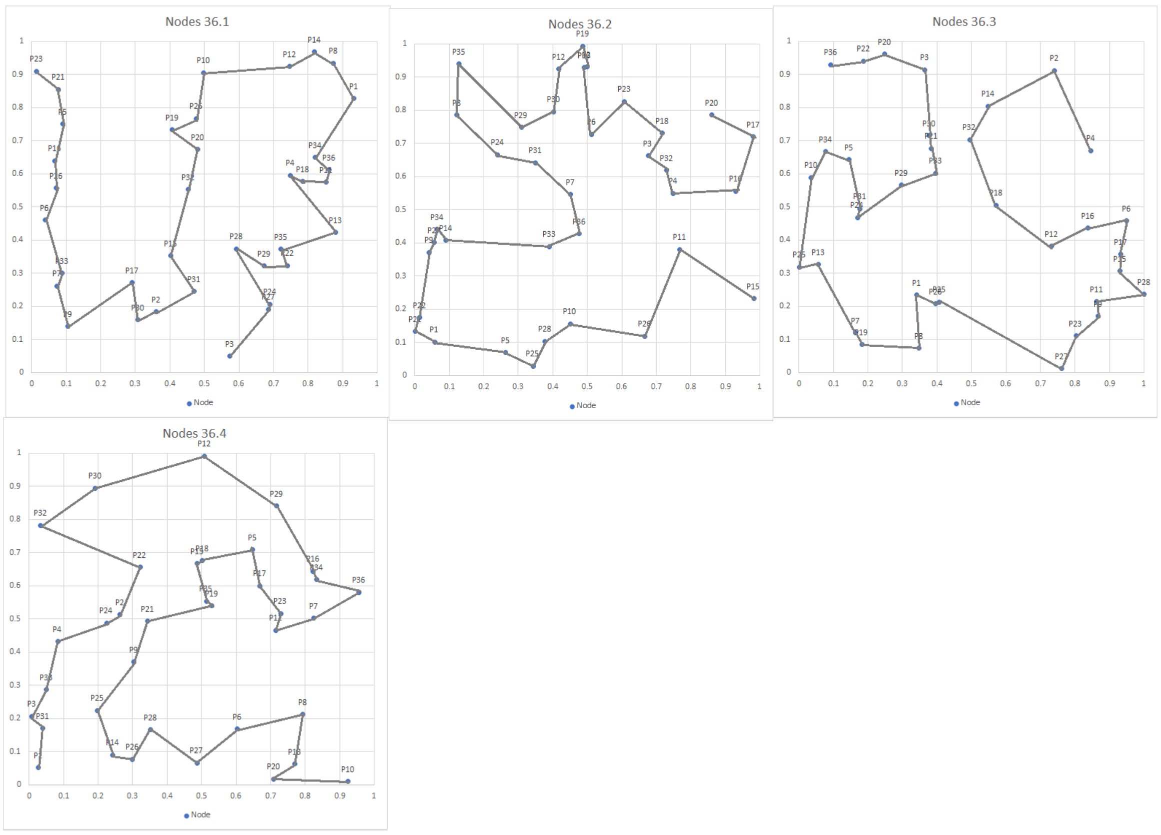
Appendix H: Node distributions and trunk sequences



Appendix H.1: node distributions and trunk sequences for node densities 12, 16 and 20 used in the simulations.



Appendix H.2: node distributions and trunk sequences for node densities 24, 28 and 32 used in the simulations.



Appendix H.3: node distributions and trunk sequences for node density 36 used in the simulations.

Appendix I: Code

Appendix I.1: Code used for dynamic DRT simulations.

```
"""

Script for simulating a dynamic Demand Responsive Transport system based on an area with nodes.
This version successfully creates routes based on an initial random set of requests, after which it
adds new requests based on processed requests after a certain timeframe (distance).
This version writes successive iterations into one Excel file with correct load information, for both
real-time and cumulative loads.
This version generates random requests over a certain time period, which are periodically inserted
into the main route.

"""

from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
from random import random, sample, choices
from random import *
from datetime import datetime
import openpyxl
import numpy as np
from numpy import *
import os
import csv
import time

startTime = time.time()

####################
##Input variables:##
####################

file = open(r"C:\Users\Path\DistanceMatrix.txt", "r")
distMatrix = file.read()
distMatrix = eval(distMatrix)


nodeDuplicationFactor = 8                        #The maximum times a request can be made from or to
the same node is half of this value. Value should therefore always be even. Not recommended to go
above 10.

numberOfRequests = 10                            #The number of initial requests

requestsPerHour = 4                             #The expected amount of requests per hour

amountOfRunningHours = 6                         #The amount of hours the simulation should run

maxPeoplePerRequest = 5                          #The maximum amount of people belonging to one
request

requestWeights = [75, 15, 5, 3, 2]                 #The chance distribution of the amount of people
for one request. [6, 3, 1, 1, 1] means that 50% of requests have 1 person, 25% have 2 people, etc.

path = r"C:\Users\Path\Output"    #Set the folder to save the results in.

#############################
##Non-changeable variables:##
#############################

amountOfNodes = len(distMatrix) #Amount of locations that can be visited: includes the dummy node 0.

workbook = openpyxl.Workbook() #Initiate

depot = 0

#######################
##Callable functions:##
#######################

def createRandomRequestList():
#Creates the list of random requests to be inserted over the runtime of the simulation
    newRequests = []
    for i in range(1, int(amountOfRunningHours * 60)):
        createRequest = uniform(0, 60) < requestsPerHour
        if createRequest:
            time = i * 100
            newRequest = sample(range(1,amountOfNodes),2)
            newRequests.append(newRequest + choices(range(1, maxPeoplePerRequest + 1), weights =
requestWeights) + [time, 0, 0, 0, 0])

    return newRequests

def createNodeDummies(distMatrix):
#Creates the multiplied distance matrix necessary for simultaneous pickup and dropoff
```

```
    matrix = []
    for x in distMatrix:
        matrix.append(nodeDuplicationFactor * x)
    matrix = nodeDuplicationFactor * matrix
    return matrix

def createRequests(originalRequests, amountOfRequests, maxPeoplePerRequest, requestWeights, 
amountOfNodes):
#Creates a list of random requests. Keeps into account original requests to be saved. Adheres to 
request constraints (does not assign more pickups and dropoffs per node than possible).
    requests = originalRequests
    while len(requests) < amountOfRequests:
        newRequest = sample(range(1,amountOfNodes + 1),2)
        checkPickUp = 0
        checkDropOff = 0
        for j in requests:
            if j[0] == newRequest[0]:
                checkPickUp += 1
            if j[1] == newRequest[1]:
                checkDropOff += 1
        if checkPickUp < nodeDuplicationFactor/2 and checkDropOff < nodeDuplicationFactor/2:
            requestTime = 0
            requests.append(newRequest + choices(range(1, maxPeoplePerRequest + 1), weights = 
requestWeights) + [requestTime, 0, 0, 0, 0])
    return requests

def assessDistancesAndReturnPrevNode(route, timeCutOff):
#Makes sure that route detours (nodes which need not be visited) are deleted and that distances are 
updated accordingly. Also returns the time cut off point, which signals that a new request has 
entered the system.
    nodeToReturn = []
    distance = route[0][1]

    for i in range(1, len(route)):
        distance += distMatrix[route[i][0]][route[i-1][0]]
        route[i][1] = distance
        if route[i-1][1] <= timeCutOff and route[i][1] > timeCutOff:
            nodeToReturn = route[i]
    if timeCutOff < route[0][1]:
        return route[0]
    elif timeCutOff >= route[-1][1]:
        return route[-1]
    return nodeToReturn

def setNodes(requests, pendingRequests):
#Sets the distance nodes to the extra values to prevent double visit errors
    for y in requests:
            y[1] += amountOfNodes
    for y in pendingRequests:
            y[1] += amountOfNodes
    for i in range(amountOfNodes):
        foundPickUp = 0
        foundDelivery = 0
        for x in requests:
            if x[0] == i:
                foundPickUp += 1
                x[0] += (foundPickUp - 1) * 2 * amountOfNodes
            if x[1] == i + amountOfNodes:
                foundDelivery += 1
                x[1] += (foundDelivery - 1) * 2 * amountOfNodes
        for x in pendingRequests:
            if x[0] == i:
                foundPickUp += 1
                x[0] += (foundPickUp - 1) * 2 * amountOfNodes
            if x[1] == i + amountOfNodes:
                foundDelivery += 1
                x[1] += (foundDelivery - 1) * 2 * amountOfNodes
    return [row for row in requests],  [row for row in pendingRequests]

def unSetNodes(requests, pendingRequests):
#Reverts the dummy values back to real node values for presentation
    for request in requests:
        request[0] = request[0] % amountOfNodes
        request[1] = request[1] % amountOfNodes
    for request in pendingRequests:
        request[0] = request[0] % amountOfNodes
        request[1] = request[1] % amountOfNodes
    return requests, pendingRequests

def create_data_model(distMatrix, requests, depot):
    data = {}
    #The distance matrix contains the distance from all pickup nodes (nodes 0 to n) to all delivery 
nodes (nodes n+1 to 2n)
    data['distance_matrix'] = createNodeDummies(distMatrix)
    #Trip requests: each location has one pickup node and one delivery node. Note that these cannot 
be the same: each location has two nodes: i and i+[amount of locations].
    #Example: location 8 has nodes 8 and 32 (when 24 nodes exist).
    data['pickups_deliveries'] = requests
```

```python
    #The depot is a virtual depot for system purposes: node 0 has distance 0 to all nodes.
    data['depot'] = depot
    return data

def printExcel(workbook, route, totalFulfilledRequests, amountOfNewRequests):
#Prints the final data to an Excel file
    workbook.worksheets[0].title = 'Complete route'
    workbook.worksheets[0].append(['Node', 'Distance', 'Load', 'Passengers'])
    servicedRequestsSheet = workbook.create_sheet('Serviced requests', 1)
    servicedRequestsSheet.append(['PickUp', 'DropOff', 'People', 'Requested Time', 'Initial Pickup
Time', 'True Pickup Time', 'Indicated dropoff', 'True dropoff'])

    for i in range(len(route)):
        if i == 0 or route[i-1] != route[i]:
            workbook.worksheets[0].append(route[i])

    for request in totalFulfilledRequests:
        servicedRequestsSheet.append(request)

    servicedRequestsSheet.append(['New requests: ', amountOfNewRequests])

    os.makedirs(path, exist_ok=True)

    currentTime = str(datetime.now().strftime('%m_%d_%H_%M_%S'))
    filePath = os.path.join(path, currentTime + ' ' + str(requestsPerHour) + ' RpH' + 'output.xlsx')
    workbook.save(filePath)
    print("Saved as ", filePath)

def processSolution(data, manager, requests, pendingRequests, totalLoad, initialDistance,
previousEndPoint, timeCutOff, routing, solution):
#Processes the solution given by the program. Calculates the correct loads and writes it, along with
the distances to the Excel file.
    masterRoute = []
    total_distance = 0
    newSheet = workbook.create_sheet('route ')
    writeHeader = ['Node', 'Distance', 'Load', 'Passengers']

    index = routing.Start(0)
    if requests[0][3] <= initialDistance:
        route_distance = initialDistance
    else:
        route_distance = requests[0][3]
    #Initialize variables.
    previous_route_distance = 0
    route_load = 0
    total_load = totalLoad
    previous_index = 0
    printString = []
    fulfilledRequests = []
    pendingToReturn = []
    fulfilledToReturn = []
    requestsToReturn = []
    totalLoadToReturn = totalLoad
    fulfilledRouteToReturn = []
    lastPointToReturn = 0
    cutoffPoint = True
    firstNode = True

    #Add pending requests loads to the current load.
    for request in pendingRequests:
        route_load += request[2]

    togo = requests.copy()
    passed = pendingRequests.copy()
    while not routing.IsEnd(index):
        node_index = manager.IndexToNode(index)
        if node_index%amountOfNodes > 0:
            if firstNode:
                route_distance += distMatrix[previousEndPoint%amountOfNodes][index%amountOfNodes]
                initialDistance += distMatrix[previousEndPoint%amountOfNodes][index%amountOfNodes]
                firstNode = False
            removeReqs = []
            for request in togo:
                #If the current node is the pickup node of a request, add the new passengers to the
load variables.
                if request[0]%amountOfNodes == index%amountOfNodes:
                    route_load += request[2]
                    total_load += request[2]
                    passed.append(request)
                    removeReqs.append(request)
            for request in removeReqs:
                togo.remove(request)
            removePends = []
            for pending in passed:
                #If the current node is the dropoff node of a pending request, remove the
disembarking passengers from the current load variable.
                if pending[1]%amountOfNodes == index%amountOfNodes:
                    route_load -= pending[2]
                removePends.append(pending)
            for pend in removePends:
                passed.remove(pend)
            printString = [index%amountOfNodes, route_distance, route_load, total_load]


        if node_index%amountOfNodes > 0:
            previous_index = index
        index = solution.Value(routing.NextVar(index))
        while previous_index%amountOfNodes == index%amountOfNodes:
            index = solution.Value(routing.NextVar(index))

        if ((previous_index % amountOfNodes) != 0 and (index % amountOfNodes) != 0) and abs(index %
amountOfNodes) != abs(previous_index % amountOfNodes):
            masterRoute.append(printString)

        route_distance += routing.GetArcCostForVehicle(
            previous_index, index, 0)

        previous_route_distance = route_distance

    masterRoute.append(printString)

    total_distance += route_distance

    newSheet.append(writeHeader)

    #Print the provisional route in a new sheet to Excel.
    routeToPrint = []
    for i in range(len(masterRoute)):
        if i == 0:
            routeToPrint.append(masterRoute[i])
        elif masterRoute[i][2] != masterRoute[i-1][2] or masterRoute[i][3] != masterRoute[i-1][3]:
            routeToPrint.append(masterRoute[i])

    pointToReturn = assessDistancesAndReturnPrevNode(routeToPrint, timeCutOff)
    visitedNodes = []
    #Update the pending requests and fulfilled requests, and update their values.
    for i in range(len(routeToPrint)):
        removeReqs = []
        for request in requests:
        #If the current node is the pickup node of a request, assign the request to the pending
requests lists, and update pickup times and loads.
            if request[0]%amountOfNodes == routeToPrint[i][0]:
                if request[4] == 0:
                    request[4] = routeToPrint[i][1]
                request[5] = routeToPrint[i][1]
                if i == 0:
                    pendingRequests.append(request)
                    removeReqs.append(request)
                elif routeToPrint[i-1][1] < timeCutOff:
                    pendingRequests.append(request)
                    removeReqs.append(request)
            if request[1]%amountOfNodes == routeToPrint[i][0] and request[0]%amountOfNodes in
visitedNodes:
                if request[6] == 0:
                    request[6] = routeToPrint[i][1]
                request[7] = routeToPrint[i][1]
        for request in removeReqs:
            requests.remove(request)
        removePends = []
        for pending in pendingRequests:
            #If the current node is the drop-off node of a pending request, assign the request to the
fulfilled requests lists, and update drop-off times and loads.
            if pending[1]%amountOfNodes == routeToPrint[i][0]:
                if pending[6] == 0:
                    pending[6] = routeToPrint[i][1]
                pending[7] = routeToPrint[i][1]
                if i == 0:
                    fulfilledRequests.append(pending)
                    removePends.append(pending)
                elif routeToPrint[i-1][1] < timeCutOff:
                    fulfilledRequests.append(pending)
                    removePends.append(pending)
        for pending in removePends:
            pendingRequests.remove(pending)
        visitedNodes.append(routeToPrint[i][0])

    if timeCutOff < 999999: #To indicate that it is the last node of the route
        totalLoadToReturn = pointToReturn[3]
        lastPointToReturn = pointToReturn[0]

    fulfilledRouteToReturn = []
    for i in range(len(routeToPrint)):
        if i == 0:
            fulfilledRouteToReturn.append(routeToPrint[i])
        elif routeToPrint[i-1][1] < timeCutOff:
            fulfilledRouteToReturn.append(routeToPrint[i])
    pendingToReturn = pendingRequests.copy()
    fulfilledToReturn = fulfilledRequests.copy()
    requestsToReturn = requests.copy()

    for node in routeToPrint:
        newSheet.append(node)

    return masterRoute, requestsToReturn, fulfilledToReturn, pendingToReturn, fulfilledRouteToReturn, 
lastPointToReturn, totalLoadToReturn

def runProgram(requests, pendingRequests, totalLoad, initialDistance, previousEndPoint, timeCutOff, 
depot):
#Runs the solver
    startTime = time.time()

    #Write the open requests to a new Excel sheet
    newSheet = workbook.create_sheet('req ')
    writeHeader = ['PickUp', 'DropOff', 'People', 'Requested Time']
    newSheet.append(writeHeader)
    for request in requests:
        newSheet.append([request[0]%amountOfNodes, request[1]%amountOfNodes, request[2], request[3]])

    # Instantiate the data problem.
    data = create_data_model(distMatrix, requests, depot)

    # Create the routing index manager.
    manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                           1, data['depot'])

    # Create Routing Model.
    routing = pywrapcp.RoutingModel(manager)

    # Define the cost of each arc between each node pair: this is the distance between them.
    def distance_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return data['distance_matrix'][from_node][to_node]

    transit_callback_index = routing.RegisterTransitCallback(distance_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # Add the distance dimension.
    dimension_name = 'Distance'
    routing.AddDimension(
        transit_callback_index,
        0,
        100000, #no maximum distance
        True,
        dimension_name)
    distance_dimension = routing.GetDimensionOrDie(dimension_name)
    distance_dimension.SetGlobalSpanCostCoefficient(100)

    # Set the constraints resulting from the requests
    for request in data['pickups_deliveries']:
        pickup_index = manager.NodeToIndex(request[0])
        delivery_index = manager.NodeToIndex(request[1])
        routing.AddPickupAndDelivery(pickup_index, delivery_index)
        routing.solver().Add(
            routing.VehicleVar(pickup_index) == routing.VehicleVar(
                delivery_index))
        #Pickup nodes of each request should always come in the route before the drop-off node in the 
same request.
        routing.solver().Add(
            distance_dimension.CumulVar(pickup_index) <=
            distance_dimension.CumulVar(delivery_index))


    # Set the initial solution heuristic and the metaheuristic for route searching.
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PARALLEL_CHEAPEST_INSERTION)
    search_parameters.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH)
    search_parameters.time_limit.seconds = 2

    # Solve the problem.
    solution = routing.SolveWithParameters(search_parameters)

    # Print solution on console.
    if solution:
        masterRouteLocal, requests, fulfilledRequests, pendingRequests, routeTraveled, 
lastPointToReturn, totalLoad = processSolution(data, manager, requests, pendingRequests, totalLoad, 
initialDistance, previousEndPoint, timeCutOff, routing, solution)

    return masterRouteLocal, requests, fulfilledRequests, pendingRequests, routeTraveled, 
lastPointToReturn, totalLoad
```

```
def main():
#The main call for the porgram. Generates the requests and iterates over them.

    #Initial variables: no requests, no route and no load.
    requests = []
    pendingRequests = []
    fulfilledRequests = []
    totalFulfilledRequests = []
    route = []
    totalLoad = 0
    initialDistance = 0
    previousEndPoint = 0

    #Create new random pickup requests.
    initialRequests = createRequests(requests, numberOfRequests, maxPeoplePerRequest, requestWeights,
amountOfNodes-1)
    newRequests = createRandomRequestList()
    newRequestsLength = len(newRequests)

    for i in range(newRequestsLength + 1):
        if i == 0:
            requests = initialRequests
        else:
            requests.append(newRequests[0])
            newRequests.remove(newRequests[0])

        #Give dummy nodes a distinct number.
        setNodes(requests, pendingRequests)
        #Error catch if the first node pickup is the same as the depot
        if requests[0][0]%amountOfNodes == previousEndPoint:
            requests[0][0] += amountOfNodes * 2

        if requests[0][1]%amountOfNodes == previousEndPoint:
            requests[0][1] += amountOfNodes * 2


        if i < newRequestsLength:
            nextTimeCutOff = newRequests[0][3]
        else:
            nextTimeCutOff = 9999999999999
        #Create a new route for the given requests and the pending requests.
        masterRoute, requests, fulfilledRequests, pendingRequests, routeTraveled, previousEndPoint,
totalLoad = runProgram(requests, pendingRequests, totalLoad, initialDistance, previousEndPoint,
nextTimeCutOff, previousEndPoint)

        #Give dummy nodes the same number.
        unSetNodes(requests, pendingRequests)

        #Add the fulfilled requests in this iteration to the total list of fulfilled requests.
        for request in fulfilledRequests:
            totalFulfilledRequests.append(request)

        for node in routeTraveled:
            route.append(node)

        initialDistance = route[-1][1]

        #Print the elapsed time for this iteration.
        endTime = time.time()
        elapsedTime = endTime - startTime
        print("Iteration ", i, ", Time:", elapsedTime, " seconds")

    #Give the fulfilled requests their real node value from their dummy value.
    unSetNodes(totalFulfilledRequests, [])

    uniqueRequests = []
    for request in totalFulfilledRequests:
        if uniqueRequests.count(request) == 0:
            uniqueRequests.append(request)

    #Print the total list of fulfilled requests to the Excel file and save.
    printExcel(workbook, route, uniqueRequests, newRequestsLength)

if __name__ == '__main__':
    main()
```

Appendix I.2: Code used for semi-flexible DRT simulations.

```
"""

Script for simulating a semi-flexible Demand Responsive Transport system based on an area with nodes.
This version successfully creates routes based on an initial random set of requests, after which it
adds new requests based on processed requests after a certain timeframe (distance).
This version writes successive iterations into one Excel file with correct load information, for both
real-time and cumulative loads.
This version keeps track of individual request times and true pickup times per request.

"""

from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
from random import random, sample, choices
from random import *
from datetime import datetime
import openpyxl
import numpy as np
from numpy import *
from pathlib import Path
import os
import csv
import time

####################
##Input variables:##
####################
file = open(r"C:\Users\Path\DistanceMatrix.txt", "r")
distMatrix = file.read()
distMatrix = eval(distMatrix)

maxRequestsperNode = 4

numberOfRequests = 10

maxPeoplePerRequest = 5

requestWeights = [75, 15, 5, 3, 2]

path = r"C:\Users\Path\OutputInterim" #Set the folder to save the interim results in.

finalPath = r"C:\Users\Path\OutputFinal" #Set the folder to save the final results in.

trunkFile = open(r"C:\Users\Path\Trunk.txt", "r") #The array with the trunk sequence

expeditionTime = 3000   #The time one expedition (itinerary) of the vehicle is alloted

requestsPerHour = 24     #Number of expected random requests per hour

expeditions = 12 #number of expeditions (half one way, half the other)

#############################
##Non-changeable variables:##
#############################

amountOfNodes = len(distMatrix) #Amount of locations that can be visited: includes the dummy node 0.

sequence = trunkFile.read()     #Read the trunk sequence from its file.
sequence = eval(sequence)

def createRandomRequestList(iteration):
#Creates  a new set of random requests for a specific itinerary.
    newRequests = []
    expectedAmountOfRequests = requestsPerHour * (expeditionTime/6000)
    for i in range(int(expeditionTime/100)):
        createRequest = uniform(0, 60) < requestsPerHour
        if createRequest:
            time = i * 100
            newRequest = sample(range(1,amountOfNodes),2)
            newRequests.append(newRequest + choices(range(1, maxPeoplePerRequest + 1), weights =
requestWeights) + [time + iteration * expeditionTime, 0, 0, 0, 0])

    return newRequests

def createRequests(currentSequence, amountOfRequests, maxPeoplePerRequest, requestWeights,
amountOfNodes):
#Creates a list of initial random requests, to be called in the first iteration.
    requests = []
    while len(requests) < amountOfRequests:
        newRequest = sample(range(1,amountOfNodes),2)
        checkPickUp = 0
        checkDropOff = 0
        for j in requests:
            if j[0] == newRequest[0]:
                checkPickUp += 1
            if j[1] == newRequest[1]:
                checkDropOff += 1
        if currentSequence.index(newRequest[0]) != 0 and currentSequence.index(newRequest[1]) != 0:
            if checkPickUp < maxRequestsperNode and checkDropOff < maxRequestsperNode:
                requestTime = 0
                requests.append(newRequest + choices(range(1, maxPeoplePerRequest + 1), weights =
requestWeights) + [requestTime, 0, 0, 0, 0])
    return requests

def printExcel(workbook, route, fulfilledRequests, storedRequests, i):
#Print the final route and served requests to an Excel file.
    workbook.worksheets[0].title = 'Complete route'
    workbook.worksheets[0].append(['Node', 'Distance', 'Load', 'Passengers'])
    servicedRequestsSheet = workbook.create_sheet('Serviced requests', 1)
    servicedRequestsSheet.append(['PickUp', 'DropOff', 'People', 'Requested Time', 'Initial Pickup
Time', 'True Pickup Time', 'Indicated dropoff', 'True dropoff'])
    unServicedRequestsSheet = workbook.create_sheet('Unserviced requests', 2)
    unServicedRequestsSheet.append(['PickUp', 'DropOff', 'People', 'Requested Time'])

    for node in route:
        workbook.worksheets[0].append(node)

    for request in fulfilledRequests:
        servicedRequestsSheet.append(request)

    for request in storedRequests:
        unServicedRequestsSheet.append(request)

    os.makedirs(path, exist_ok=True)

    currentTime = str(datetime.now().strftime('%m_%d_%H_%M_%S'))
    filePath = os.path.join(path, currentTime + ' ' + str(i) + 'output.xlsx')
    workbook.save(filePath)
    print("Saved as ", filePath)

def print_solution(solution, workbook, iteration):
#Print a single (provisional) route to an excel sheet
    newSheet = workbook.create_sheet('route ' + ', '.join(str(element) for element in iteration))
    writeHeader = ['Node', 'Distance', 'Load', 'Passengers']
    newSheet.append(writeHeader)

    for node in solution:
        newSheet.append(node)

def runProgram(currentSequence, requests, initialNode, workbook, iteration):
#Calculates a new route based on current requests
    startTime = time.time()
    activeNodesUnSorted = []
    pendingRequests = []
    requestsForLater = []
    newRequest = []
    if len(requests) > 0:
        newRequest = requests[-1].copy()
        requests.remove(newRequest)

    for request in requests:
        #if the request is a new request (i.e. its time of request is higher than the initial time),
don't process
        if currentSequence.index(request[0]) < currentSequence.index(request[1]):
            pendingRequests.append(request)
        else:
            requestsForLater.append(request)

    if newRequest != []:
        #If the request has its pickup node before the drop-off node in the sequence, accept the
request
        if currentSequence.index(initialNode[0]) <= currentSequence.index(newRequest[0]) <
currentSequence.index(newRequest[1]):
            pendingRequests.append(newRequest)
        else:
        #Else, save it for another itinerary.
            requestsForLater.append(newRequest)

    #Determine which nodes have demand, i.e. which nodes need to be visited.
    for request in pendingRequests:
        if currentSequence.index(initialNode[0]) < currentSequence.index(request[0]):
            if request[0] not in activeNodesUnSorted:
                activeNodesUnSorted.append(request[0])
        if currentSequence.index(initialNode[0]) < currentSequence.index(request[1]):
            if request[1] not in activeNodesUnSorted:
                activeNodesUnSorted.append(request[1])
    indices = []
    for node in activeNodesUnSorted:
        indices.append(currentSequence.index(node))

    #Sort the nodes based on the trunk sequence.
    activeNodesSorted = [x for y, x in sorted(zip(indices, activeNodesUnSorted))]
    #Make sure that the route ends at the last node of the trunk sequence.
    if activeNodesSorted != []:
        if activeNodesSorted[-1] != currentSequence[-1]:
            activeNodesSorted.append(currentSequence[-1])

    #Set the starting time of the itinerary.
    if iteration[1] == 0:
        initialNode[1] = iteration[0] * expeditionTime
    solution = []
    solution.append(initialNode)
    for node in activeNodesSorted:
        solution.append([node, 0, 0, 0])

    #Initialize the vehicle load.
    totalDistance = initialNode[1]
    load = initialNode[2]
    totalPassengers = initialNode[3]

    #Set the loads and distances according to the traveled route.
    for i in range(len(solution)):
        #Set the initial distance of the itinerary
        if i > 0:
            totalDistance += distMatrix[solution[i][0]][solution[i-1][0]]
        for request in pendingRequests:
            if request[0] == solution[i][0] and request[3] > totalDistance:
                totalDistance = newRequest[3]
        solution[i][1] = totalDistance
        for request in pendingRequests:
            if request[0] == solution[i][0]:
                #For the first element of the solution, the loads and passengers are already added in
the previous iteration, except for the new request.
                if i != 0 or iteration[1] == 0 or request == newRequest:
                    load += request[2]
                    totalPassengers += request[2]
                if request[4] == 0:
                    request[4] = totalDistance
                request[5] = totalDistance
            elif request[1] == solution[i][0]:
                if i != 0 or iteration[1] == 0 or request == newRequest:
                    load -= request[2]
                if request[6] == 0:
                    request[6] = totalDistance
                request[7] = totalDistance
            solution[i][2] = load
            solution[i][3] = totalPassengers

    newSheet = workbook.create_sheet('req ' + ', '.join(str(element) for element in iteration))

    writeHeader = ['PickUp', 'DropOff', 'People', 'Requested Time', 'Initial Pickup Time', 'True
Pickup Time', 'Initial Dropoff Time', 'True Dropoff Time']

    newSheet.append(writeHeader)

    for request in pendingRequests:
        newSheet.append(request)

    # Print solution to Excel.
    print_solution(solution, workbook, iteration)

    return solution, pendingRequests, requestsForLater, solution[-1]

def main():

    #Initial variables: no requests, no route and no load.
    currentSequence = sequence
    pendingRequests = []
    totalFulfilledRequests = []
    completeRoute = []
    totalRoute = []
    storedRequests = []

    #Calculate the route for each itinerary (expedition)
    for i in range(expeditions):
        workbook = openpyxl.Workbook()
        pendingRequests = []
        fulfilledRequests = []
        totalRoute = []

        if i == 0:
            initialNode = [currentSequence[0], i*expeditionTime, 0, 0]

        if i == 0:
            #Create new random pickup requests. Maximum amount of passengers per request is 5, with a
50% chance of it being only 1.
            pendingRequests = createRequests(currentSequence, numberOfRequests, maxPeoplePerRequest,
requestWeights, amountOfNodes-1)
        else:
            #If the itinerary is not the first one, use the non-served requests from the previous
itinerary.
            pendingRequests = storedRequests.copy()

        storedRequests = []

        #Create a list of random requests to be inserted during the itinerary.
        newRequests = createRandomRequestList(i)

        newRequestsLength = len(newRequests)

        #For each new request to be inserted, recalculate the route, distances and loads.
        for j in range(newRequestsLength + 1):

            iteration = [i, j]

            #Run the calculation for the insertion of the new request
            route, pendingRequests, requestsForLater, initialNode = runProgram(currentSequence,
pendingRequests, initialNode, workbook, iteration)

            definitiveRoute = []

            #Make sure that once requests are fulfilled, they are added to the final list of requests
            #Make sure that the route actually traveled by the vehicle is added to the final overall
route.
            if j < newRequestsLength:
                requestToInsert = newRequests[0]

                count = 0
                for node in route:
                    if node[1] <= requestToInsert[3]:
                        definitiveRoute.append(node)
                        count += 1
                if count < len(route):
                    definitiveRoute.append(route[count])

                toRemove = []
                for request in pendingRequests:
                    if request[7] <= definitiveRoute[-1][1]:
                        fulfilledRequests.append(request)
                        totalFulfilledRequests.append(request)
                        toRemove.append(request)
                for request in toRemove:
                    pendingRequests.remove(request)

                for node in definitiveRoute:
                    if totalRoute.count(node) == 0:
                        totalRoute.append(node)
                        completeRoute.append(node)

                pendingRequests.append(requestToInsert)
                newRequests.remove(requestToInsert)

            else:
                for node in route:
                    if totalRoute.count(node) == 0:
                        totalRoute.append(node)
                        completeRoute.append(node)
                toRemove = []
                for request in pendingRequests:
                    if request[7] <= totalRoute[-1][1]:
                        fulfilledRequests.append(request)
                        totalFulfilledRequests.append(request)
                        toRemove.append(request)
                for request in toRemove:
                    pendingRequests.remove(request)

            initialNode = totalRoute[-1]

            for request in requestsForLater:
                    storedRequests.append(request)

        initialNode = totalRoute[-1]
        if initialNode[0] != currentSequence[-1]:
            initialNode[0] = currentSequence[-1]

        currentSequence.reverse()

        #Print the total list of fulfilled requests to the Excel file and save.
        printExcel(workbook, totalRoute, fulfilledRequests, storedRequests, i)

    uniqueNodes = []
    for node in completeRoute:
        if uniqueNodes.count(node) == 0:
            uniqueNodes.append(node)

    #Print the final route to an Excel file
    file = Path(finalPath, str(requestsPerHour) + ' RpH.xlsx')
    if file.is_file():
        workbook = openpyxl.load_workbook(file)
    else:
        workbook = openpyxl.Workbook()
        workbook.worksheets[0].title = 'Complete route'
        workbook.worksheets[0].append(['Node', 'Distance', 'Time', 'Load', 'Passengers'])
        servicedRequestsSheet = workbook.create_sheet('Serviced requests', 1)
        servicedRequestsSheet.append(['PickUp', 'DropOff', 'People', 'Requested Time', 'Initial 
Pickup Time', 'True Pickup Time', 'Indicated dropoff', 'True dropoff'])

    #The below code makes sure that a calculated route is added to an existing Excel file and printed 
from the first empty column.
    insertColumn = 1;
    for column in range(1, 100):
        if workbook.worksheets[0].cell(2, column).value == None:
            insertColumn = column
            break
    print(insertColumn)
    i = 2
    for node in uniqueNodes:
        cell1 = workbook.worksheets[0].cell(row = i, column = column)
        cell1.value = node[0]
        cell2 = workbook.worksheets[0].cell(row = i, column = column + 1)
        cell2.value = node[1]
        cell3 = workbook.worksheets[0].cell(row = i, column = column + 2)
        cell3.value = node[1]/100
        cell4 = workbook.worksheets[0].cell(row = i, column = column + 3)
        cell4.value = node[2]
        cell4 = workbook.worksheets[0].cell(row = i, column = column + 4)
        cell4.value = node[3]
        i += 1

    for request in totalFulfilledRequests:
        workbook.worksheets[1].append(request)

    #Save the Excel file.
    os.makedirs(path, exist_ok=True)

    currentTime = str(datetime.now().strftime('%m_%d_%H_%M_%S'))
    filePath = os.path.join(finalPath, str(requestsPerHour) + ' RpH.xlsx')
    workbook.save(filePath)
    print("Saved as ", filePath)


if __name__ == '__main__':
    main()
```