CHAPTER 23

Dynamic Transport Services

Michal Maciejewski

Entry point to documentation:

`http://matsim.org/extensions` → dvrp

Invoking the module:

No predefined invocation. Starting point(s) under `http://matsim.org/javadoc` → dvrp → `RunOneTaxiExample` class.

Selected publications:

Maciejewski and Nagel (2013b,c,a); Maciejewski (2014)

23.1 Introduction

The recent technological advancements in ICT (Information and Communications Technology) provide novel, on-line fleet management tools, opening up a broad range of possibilities for more intelligent transport services: flexible, demand-responsive, safe and energy/cost efficient. Significant enhancements can aid in both traditional transport operations, like regular public transport or taxis and introduction of novel solutions, such as demand-responsive transport or personal rapid transport. However, the growing complexity of modern transport systems, despite all benefits, increases the risk of poor performance, or even failure, due to lack of precise design, implementation and testing.

One solution is to use simulation tools offering a wide spectrum of possibilities for validating transport service models. Such tools have to model, in detail, not only the dynamically changing demand and supply of the relevant service, but also traffic flow and other existing transport services, including mutual interactions/relations between all these components. Although several approaches have been proposed (e.g., Regan et al., 1998; Barcelo et al., 2007; Liao et al., 2008;

How to cite this book chapter:

Maciejewski, M. 2016. Dynamic Transport Services. In: Horni, A, Nagel, K and Axhausen, K W. (eds.) *The Multi-Agent Transport Simulation MATSim*, Pp. 145–152. London: Ubiquity Press. DOI: http://dx.doi.org/10.5334/baw.23.

Certicky et al., 2014), as far the author knows, no existing solutions provide large-scale microscopic simulation that include all the components above.

23.2 DVRP Contribution

To address the problem above, MATSim's DVRP (Dynamic Vehicle Routing Problem) contribution has been developed. The contribution is designed to be highly general and customizable to model and simulate a wide range of dynamic vehicle routing and scheduling processes. Currently, the domain model is capable of representing a wide range of one-to-many and many-to-many VRPs; one can easily extend the model even further to cover other specific cases (see Section 23.3). Since online optimization is the central focus, the DVRP contribution architecture allows plugging in of various algorithms. At present, there are several different algorithms available, among them an algorithm for the *Dynamic Multi-Depot Vehicle Routing Problem with Time Windows and Time-Dependent Travel Times and Costs*, analyzed in (Maciejewski and Nagel, 2012), and a family of algorithms for online taxi dispatching, studied in (Maciejewski and Nagel, 2013b,c,a; Maciejewski, 2014).

The DVRP contribution models both supply and demand, as well as optimizing fleet operations, whereas MATSim's core is used for simulating supply and demand, both embedded into a large-scale microscopic transport simulation. In particular, the contribution is responsible for:

- modeling the DVRP domain,
- listening to simulation events,
- monitoring the simulation state (e.g., movement of vehicles),
- finding least-cost paths,
- computing schedules for drivers/vehicles,
- binding drivers' behavior to their schedules, and
- coordinating interaction/cooperation between drivers, passengers and dispatchers.

Dynamic transport services are simulated in MATSim as one component of the overall transport system. The optimizer plugged into the DVRP contribution reacts to selected events generated during simulation, which could be: request submissions, vehicle departures or arrivals, etc. Additionally, it can monitor the movement of individual vehicles, as well as query other sources of online information, e.g., current traffic conditions. In response to changes in the system, the optimizer may update drivers' schedules, either by applying smaller modifications or re-optimizing them from scratch. Drivers are notified about changes in their schedules and adjust to them as soon as possible, including immediate diversion from their current destinations. For passenger transport, such as taxi or demand-responsive transport services, interactions between drivers, passengers and the dispatcher are simulated in detail, including calling a ride or picking up and dropping off passengers.

23.3 DVRP Model

The DVRP contribution can be used for simulating *Rich VRPs*. Compared to the classic *Capacitated VRP*, the major model enhancements are:

- one-to-many (many-to-one) and many-to-many topologies,
- multiple depots,
- dynamic requests,
- request and vehicle types,
- time windows for requests and vehicles,

- time-dependent stochastic travel times and costs, and
- network-based routing (including route planning, vehicle monitoring and diversion).

Except for the travel times and costs (discussed in Section 23.3.2), which are calculated on demand, all the VRP-related data are accessible via `VrpData`.[1] In the most basic setup, there are only two types of entities, namely `Vehicles` and `Requests`. This model, however, can be easily extended as required. For instance, for an electric vehicle fleet, specialized `ElectricVrpData` also stores information about `Chargers`. This, and other examples of extending the base VRP model, such as a model of the *VRP with Pickup and Delivery*, are available in the `org.matsim.contrib.dvrp.extensions` package.

23.3.1 Schedule

Each `Vehicle` has a `Schedule`, a sequence of different `Tasks`, such as driving from one location to another (`DriveTask`), or staying at a given location (e.g., serving a customer or waiting; `StayTask`).[2] A `Schedule` is where supply and demand are coupled. All schedules are calculated by an online optimization algorithm (see Section 23.6) representing the fleet's dispatcher. Each task is in one of the following states (defined in the `Task.TaskStatus` enum): `PLANNED`, `STARTED` or `PERFORMED`; each schedule's status is one of the following:

- `UNPLANNED`—no tasks assigned
- `PLANNED`—all tasks are `PLANNED` (none of them started)
- `STARTED`—one of the tasks is `STARTED` (this is the schedule's `currentTask`; the preceding tasks are `PERFORMED` and the succeeding ones are `PLANNED`)
- `COMPLETED`—all tasks are `PERFORMED`

In general, when modifying a `Schedule`, one can freely change and rearrange the planned tasks; those performed are considered to be read-only. For the current task, one can, for instance, change its end time, although the start time must remain unchanged. Proceeding from the current task to the next one is carried out by invoking the `Schedule.nextTask()`) method.

The execution of the current task may be monitored with a `TaskTracker`.[3] In the most basic version, trackers predict only the end time of the current task. More complex trackers also provide detailed information on the current state of task execution. `OnlineDriveTaskTracker`, for example, offers functionality similar to GPS navigation, such as monitoring the movement of a vehicle, predicting its arrival time and even diverting its path.

`ScheduleImpl`, along with `DriveTaskImpl` and `StayTaskImpl`, is the default implementation of `Schedule` and offers several additional features, such as data validation or automated task handling. It also serves as the starting point when implementing domain-specific schedules or tasks (e.g., `ChargeTask` in the electric VRP model mentioned above).

23.3.2 Least-Cost Paths

MATSim's network model consists of nodes connected by one-way links. Because of the queue-based traffic flow simulation (Section 1.3), a link is the smallest traversable element (i.e., a vehicle cannot stop in the middle of a link). Besides links, the DVRP contribution also operates on a higher level of abstraction: paths. Each path is a sequence of links to be traversed to get from one location

[1] Package `org.matsim.contrib.dvrp.data`.
[2] Package `org.matsim.contrib.dvrp.schedule`.
[3] Package `org.matsim.contrib.dvrp.tracker`.

to another in the network, or more precisely, from the end of one link end to the end of another link.

The functionality of finding least-cost paths is available in the `org.matsim.contrib.dvrp.router` package. `VrpPathCalculator` calculates `VrpPaths` by means of the least-cost path search algorithms available in MATSim's core (Jacob et al., 1999; Lefebvre and Balmer, 2007).[4] Because of changing traffic conditions, paths are calculated for a given departure time. Since MATSim calculates average link travel time statistics for every 15 minutes time period by default, the 15 minutes time bin is also used for computing shortest paths.

`VrpPaths` are used by `DriveTasks` to specify the link sequence to be traversed by a vehicle between two locations. It is possible to divert a vehicle from its destination by replacing the currently followed `VrpPath` with a `DivertedVrpPath`.

To reduce computational burden, the already calculated paths can be cached for future reuse (see `VrpPathCalculatorWithCache`). However, when calculating least-cost paths from one location to many potential destinations, a significant speed-up can be achieved by means of least-cost tree search (see `org.matsim.utils.LeastCostPathTree`).

23.4 DynAgent

Contrary to the standard day-to-day learning in MATSim (but see also Section 97.3.5), in the DVRP contribution, each driver behaves dynamically and follows orders coming continuously from the dispatcher. The `DynAgent` class, along with the `org.matsim.contrib.dynagent` package, provides the foundation for simulating dynamically behaving agents. Although created for DVRP contribution needs, `DynAgent` is not limited to this context and can be used in a wide range of different simulation scenarios where agent dynamism is required.

`DynAgent`'s main concept assumes an agent can actively decide what to do at each simulation step instead of using a pre-computed (and occasionally re-computed; see 30.4.2) plan. It is up to the agent whether decisions are made spontaneously or (re-)planned in advance. In some applications, a `DynAgent` may represent a fully autonomous agent acting according to his/her desires, beliefs and intentions, whereas in other cases, it may be a non-autonomous agent following orders systematically issued from the outside (e.g., a driver receiving tasks from a centralized vehicle dispatching system).

23.4.1 Main Interfaces and Classes

The `DynAgent` class is a dynamic implementation of `MobsimDriverPassengerAgent`. Instead of executing pre-planned `Activitys` and `Legs`, a `DynAgent` performs `DynActivitys` and `DynLegs`. The following assumptions underlie the agent's behavior:

- The `DynAgent` is the physical representation of the agent, responsible for the interaction with the real world (i.e., traffic simulation).
- The agent's high-level behavior is controlled by a `DynAgentLogic` that can be seen as the agent's brain; the `DynAgentLogic` is responsible for deciding on the agent's next action (leg or activity), once the current one has ended.
- Dynamic legs and activities fully define the agent's low-level behavior, down to the level of a single simulation step.

At the higher level, the `DynAgent` dynamism results from the fact that dynamic activities and legs are usually created on the fly by the agent's `DynAgentLogic`; thus, the agent does not have to plan

[4] Package `org.matsim.core.router`.

future actions in advance. When the agent has a roughly detailed legs and activities plan, he/she does not have to adhere to it and may modify his/her plan at any time (e.g., change the mode or destination of a future leg, or include or omit a future activity).

Low-level dynamism is provided by the execution of dynamic activities and legs. As for the currently executed activity, the agent can shorten or lengthen its duration at any time. Additionally, at each time step, the agent may decide what to do right now (e.g., communicate with other agents, replan the next activity or leg, and so on). When driving a car (`DriverDynLeg`), the agent can change the route, destination or even decide about picking up or dropping off somebody on the way. When using public transport (`PTPassengerDynLeg`), the agent chooses which bus to get on and at which stop to exit.

Incidentally, the behavior of MATSim's default plan-based agent, `PersonDriverAgentImpl`, can be simulated by `DynAgent`, combined with the `PlanToDynAgentLogicAdapter` logic. This adapter class creates a series of dynamic activities and legs that mimics a given `Plan` of static `Activity` and `Leg` instances.

23.4.2 Configuring and Running a Dynamic Simulation

`DynAgent` has been written for and validated against `QSim`. Dynamic leg simulation requires no additional code. However, to take advantage of dynamic activities, `DynActivityEngine` should be used, instead of `ActivityEngine`. The `doSimStep(double time)` method of `DynActivityEngine` ensures that dynamic activities are actively executed by agents and that their end times can be changed.

The easiest way to run a single iteration of `QSim` is as follows:

1. Create and initialize a `Scenario`,
2. call `DynAgentLauncherUtils`' `initQSim(Scenario scenario)` method to create and initialize a `QSim`; this includes creating a series of objects, such as an `EventsManager`, `DynActivityEngine`, or `TeleportationEngine`,
3. add `AgentSources` of `DynAgents` and other agents to the `QSim`,
4. run the `QSim` simulation, and
5. finalize processing events by the `EventsManager`.

Depending on needs, the procedure above can be extended with additional steps, such as adding non-default engines or departure handlers to the `QSim`.

23.4.3 RandomDynAgent Example

The `org.matsim.contrib.dynagent.examples.random` package contains a basic illustration of how to create and run a scenario with `DynAgents`. To highlight differences with plan-based agents, in this example 100 dynamic agents travel randomly (`RandomDynLeg`) and perform random duration activities (`RandomDynActivity`).

High-level random behavior is controlled by `RandomDynAgentLogic`, that operates according to the following rules:

1. Each agent starts with a `RandomDynActivity`; see the `computeInitialActivity(DynAgent agent)` method.
2. Whenever the currently performed activity or leg ends, a random choice on what to do next is made between the following options: (a) stop being simulated by starting a deterministic `StaticDynActivity` with infinite end time, (b) start a `RandomDynActivity`, or (c) start a `RandomDynLeg`; see the `computeNextAction(DynAction oldAction, double now)` method.

The lower level stochasticity results from random decisions being made at each consecutive decision point. In the case of `RandomDynLeg`, each time an agent enters a new link, he or she decides whether to stop at this link or to continue driving; in the latter case, the subsequent link is chosen randomly; see the `RandomDynLeg(Id<Link> fromLinkId, Network network)` constructor and the `movedOverNode(Id<Link> newLinkId)` method. As for `RandomDynActivity`, at each time step the `doSimStep(double now)` method is called and a random decision is made on the activity end time.

Following the rules specified in Section 23.4.2, setting up and running this example scenario is straightforward. `RandomDynAgentLauncher` reads a network, initializes a `QSim`, then adds a `RandomDynAgnetSource` to the `QSim`, and finally, launches visualization and starts simulation. The `RandomDynAgentSource` is responsible for instantiating 100 `DynAgents` that are randomly distributed over the network. The simulation ends when the last active agent becomes inactive.

23.5 Agents in DVRP

Realistic simulation of dynamic transport services requires a proper model of interactions and possible collaborations between the main actors: drivers, customers (often passengers) and the dispatcher. By default, drivers and passengers are simulated as agents, while the dispatcher's decisions are calculated by the optimization algorithm (see Section 23.6). This, however, is not the only possible configuration. One may simulate, for example, a decentralized system with a middleman as dispatcher rather than the fleet's manager.

23.5.1 Drivers

A driver is modeled as a `DynAgent`, whose behavior is controlled by a `VrpAgentLogic` that makes the agent follow the dynamically changing `Schedule`.[5] As a result, all changes made to the schedule are visible to and obeyed by the driver. Whenever a new task is started, the driver logic (using a `DynActionCreator`) translates it into the corresponding dynamic action. Specifically, a `DriveTask` is executed as a `VrpLeg`, whereas a `StayTask` is simulated as a `VrpActivity`. Both `VrpLeg` and `VrpActivity` are implemented so that any change to the referenced task is automatically visible to them. At the same time, any progress made while carrying them out is instantly reported to the task tracker.

23.5.2 Passengers

To simulate passenger trips microscopically, passengers are modeled as `MobsimPassengerAgent` instances. As part of the simulation, they can board, ride and, finally, exit vehicles. In contrast to the drivers, they may be modeled as the standard MATSim agents, each having a fixed daily plan consisting of legs and activities.

Interactions between drivers, passengers and the dispatcher, such as submitting `Passenger Requests` or picking up and dropping off passengers, are coordinated by a `PassengerEngine`[6]. Requests may be immediate (*as soon as possible*) or made in advance (*at the appointed time*). In the former case, a passenger starts waiting just after placing the order;in the latter case, the dispatched vehicle my arrive at the pickup location before or after the designated time, which means that either the driver or the customer, respectively, will wait for the other to come. To ensure proper coordination between these two agents, the pickup activity performed by the driver must implement the `PassengerPickupActivity` interface.

---

[5] Package `org.matsim.contrib.dvrp.vrpagent`.

[6] Package `org.matsim.contrib.dvrp.passenger`.

23.6 Optimizer

Since demand and supply are inherently stochastic, the general approach to dealing with dynamic transport services consists of updating vehicles' schedules in response to observed changes (i.e., events). This can be done by means of re-optimization procedures that consider all requests (within a given time horizon) or fast heuristics focused on small updates of the existing solution, rather than constructing a new one from scratch. Usually, re-optimization procedures give higher quality solutions compared to ocal update heuristics; however, when it comes to real-world applications, where high (often real-time) responsiveness is crucial, broad re-optimization may be prohibitively time-consuming.

In the most basic case, an optimizer implements the `VrpOptimizer` interface[7], that is, implements the following two methods:

- `requestSubmitted(Request request)`—called on submitting `request`; in response, the optimizer either adapts vehicles' schedules so that `request` can be served, or rejects it.
- `nextTask(Schedule<? extends Task> schedule)`—called whenever `schedule`'s current task has been completed and the driver switches to the next planned task; this is the last moment to make or revise the decision on what to do next.

This basic functionality can be freely extended. Besides request submission, one may, for example, consider modifying or even canceling already submitted requests. Another option is monitoring vehicles as they travel along designated routes and reacting when they are ahead of/behind their schedules. Such functionality is available by implementing `VrpOptimizerWithOnlineTracking`'s `nextLinkEntered(DriveTask driveTask)` method, which is called whenever a vehicle moves from the current link to the next one on its path.

Last but not least, there are two ways of responding to the incoming events. They can be handled either *immediately* (*synchronously*) or *between time steps* (*asynchronously*). In the former case, schedules are re-calculated (updated or re-optimized) directly, in response to the calling of the optimizer's methods. This simplifies accepting/rejecting new requests, since the answer is immediately passed back to the caller. In the latter case, all events observed within a simulation step are recorded and then processed in batch mode just before the next simulation step begins.[8] By doing that, one can not only speed up computations significantly, but also avoid situations when, due to vehicles' inertia (e.g., an idle driver can stop waiting and depart only at the beginning of the simulation step), two or more mutually conflicting decisions could be made by the optimizer at distinct moments during a single simulation step, causing the latter to overwrite the former (not always intentional).

23.7 Configuring and Running a DVRP Simulation

Like in within-day replanning (see Chapter 30), dynamic transport services are typically run with the DVRP contribution as a single-iteration simulation. Setting up and running such a simulation requires carrying out the following steps:

1. Create a `Scenario` (MATSim's domain data) and `VrpData` (VRP's domain data),
2. create a `VrpOptimizer`; this includes instantiation of a least-cost path/tree calculator, e.g., `VrpPathCalculator`, and

[7] Package `org.matsim.contrib.dvrp.optimizer`.

[8] This can be achieved by using an optimizer implementing the interface `org.matsim.core.mobsim.framework.listeners.MobsimBeforeSimStepListener`.

3. call DynAgentLauncherUtils' initQSim(Scenario scenario) method to create and initialize a QSim; this includes creating a series of objects, such as an EventsManager, DynActivityEngine, or TeleportationEngine.
4. When simulating passenger services, add a PassengerEngine to the QSim; this includes instantiation of a PassengerRequestCreator that converts calls/orders into PassengerRequests; otherwise (i.e., non-passenger services), add an appropriate source of requests to the QSim, either as a MobsimEngine or MobsimListener.
5. Then, add AgentSources to the QSim; for the DynAgent-based drivers, one may use a specialized VrpAgentSource and provide a DynActionCreator.[9]
6. run the QSim simulation, and
7. finalize processing events by the EventsManager.

The org.matsim.contrib.dvrp.run package contains VrpLauncherUtils and other utility classes that simplify certain steps of the above scheme. To facilitate access to the data representing the current state of the simulated dynamic transport service, MatsimVrpContext provides the Scenario and VrpData objects and the current time (based on the timer of QSim).

The VrpOptimizer's performance may be assessed either by analyzing the resulting schedules, or by processing events collected during the simulation.

23.8 OneTaxi Example

The org.matsim.contrib.dvrp.examples.onetaxi package contains a simple example of how to simulate on-line taxi dispatching with the DVRP contribution. In this scenario, there are ten taxi customers and one taxi driver, who serves all requests in the FIFO order. Each customer dials a taxi at a given time to get from work to home. The example is made up of six classes:

- OneTaxiRequest—represents a taxi request.
- OneTaxiRequestCreator—converts taxi calls into requests prior to submitting them to the optimizer.
- OneTaxiOptimizer—creates and updates the driver's schedule.
- OneTaxiServeTask—represents StayTasks related to picking up and dropping off customers.
- OneTaxiActionCreator—translates tasks into dynamic activities and legs.
- OneTaxiLauncher—sets up and runs the scenario.

All data necessary to run the OneTaxi example is located in the /contrib/dvrp/src/main/resources/one_taxi directory.

23.9 Research with DVRP

Currently, the DVRP contribution is used in several research projects. Two of them focus on on-line dispatching of electric taxis in Berlin and Poznan (Maciejewski and Nagel, 2013b,c,a; Maciejewski, 2014). Another project deals with design of demand-responsive transport, where DVRP has been applied to the case of twin towns, Yarrawonga and Mulwala, described in Chapter 95 (Ronald et al., 2015, 2014). In a recently launched project, the DVRP contribution will be used for simulation of DRT services in three cities: Stockholm, Tel Aviv and Leuven.

The current code development focuses on increasing performance and flexibility of the implemented shortest paths search (see Section 23.3.2). An interesting future research topic, related specifically to DRT planning, is multi-modal path search, where on-demand vehicles may be combined with fixed-route buses within a single trip. Another potential research direction is adding a benchmarking functionality and standardized interfaces so that the DVRP contribution could serve as a testbed for the *Rich VRP* optimization algorithms.

[9] Package org.matsim.contrib.dvrp.vrpagent,