MAC-POSTS Users' Manual

Prepared by

Qiling Zou and Sean Qian

Department of Civil and Environmental Engineering

Carnegie Mellon University

April 20, 2023

Contents

1 An Overview of MAC-POSTS

MAC-POSTS , the abbreviation of *Mobility data Analytics Center - Prediction, Optimization, and Simulation toolkit for Transportation Systems*, is a toolkit for dynamic network modeling developed by Mobility data Analytics Center at Carnegie Mellon University. It is designed to deal with dynamic network modeling tasks for large-scale transportation networks and can help transportation planners and researchers estimate dynamic origin-destination (OD) demand, assess traffic congestion, fuel consumption, and emissions, evaluate the effectiveness of a variety of traffic management measures, and devise cost-effective traffic management plans.

As an open-source dynamic traffic analysis toolkit, MAC-POSTS is robust, user-friendly, and relatively easy to calibrate. It incorporates multiple state-of-the-art dynamic models and demand estimation algorithms. Its core traffic simulation module is coded in C++ for performance purposes while it also provides convenient Python application programming interfaces (APIs) that promote ease of use.

MAC-POSTS can be used in many dynamic network modeling tasks, such as dynamic network loading and dynamic traffic assignment. Particularly, it has the following highlighted features:

- High spatial and temporal granularity
  As a mesoscopic model, MAC-POSTS takes the advantage of both microscopic and macroscopic models. It models the vehicles individually while tracking their movements by macroscopic models and can produce second-by-second vehicle traces throughout the entire trip.

- Large-scale simulation
  MAC-POSTS can handle large-scale dynamic network modeling tasks (e.g., regional networks with tens of thousands of links and OD pairs and millions of vehicles).

- High fidelity
  MAC-POSTS adopts a cutting edge computational-graph-based algorithm to estimate dynamic OD demand to minimize discrepancies between simulations and real-world observations. It enables effective and efficient model calibration for large-scale networks.

- Multi-source data
  MAC-POSTS can utilize different traffic data from multiple sources, such as traffic counts, travel speeds, vehicle registration information, and bus transit records, which can better calibrate the dynamic model against realistic traffic conditions.

- Multi-class vehicles
  MAC-POSTS can simulate multi-class vehicles (e.g., cars and trucks) using the state-of-the-art dynamic link and node models while the existing dynamic models often can only handle single-class vehicles. Thus, it can capture more detailed and realistic traffic dynamics and provides a more accurate estimation of fuel consumption and emissions.

- Hybrid route choice model
  MAC-POSTS adopts a hybrid route choice model: habitual travelers will take the prescribed routes according to their day-to-day experience or anticipated travel time, while other travelers, referred to as adaptive travelers, will respond to dynamic traffic conditions and choose their routes as they go. Users have the flexibility to choose the portions of habitual travelers and adaptive travelers in the network so as to produce realistic traffic conditions and network performance. Not only does the route choice model have the ability to capture travel behavior accurately, but also the simulation is computationally efficient

- Various traffic and emission metrics
  Based on the high-granularity simulation, MAC-POSTS can provide many useful metrics at both the aggregate and disaggregate levels, such as dynamic changes in travel speed, congestion condition, queuing and density on every link segment, network-wide traffic measures (Vehicle Miles Traveled (VMT), Vehicle Hours Traveled (VHT), total travel delay, average travel speed, average travel time, average travel delay, etc.), and fuel consumption and emission (hydrocarbon (HC), carbon dioxide (CO2), carbon monoxide (CO), and nitric oxide and nitrogen dioxide (NOX)).

- Flexibility and extensibility
  MAC-POSTS is designed in an object-oriented fashion. Different components of dynamic models are coded as classes in C++ with interfaces exposed to users. Based on their needs, users can simply select and combine different components, without knowing the actual implementation details of each component, to establish dynamic network models, just like “building blocks”. Such design also enables reusability and pluggability of code which allows for fast iteration and agile development.

2 Getting Started

MAC-POSTS works as a Python package. And the code is open-sourced on a GitHub repository: `https://github.com/maccmu/macposts`.

2.1 System requirements

MAC-POSTS currently supports Linux and macOS operating systems. Although there is no minimum requirement for the computer hardware, users are recommended to perform large-scale dynamic network modeling tasks with large RAM.

2.2 Installation

1. Create a Python 3 environment
   Configure a **Python 3** environment on the computer. Users can also create and work with Python virtual environments. And also make sure the latest version of **pip** (the Python package manager) is installed in this environment.

2. C++ dependencies
   To install from the repository, ensure that a working **C++ toolchain** and **CMake** $\geq$ **3.10** are installed on the computer.

3. Clone MAC-POSTS repository
   Run the following command in the terminal to initialize and clone all submodules of MAC-POSTS:

   ```
   git clone --recurse-submodules https://github.com/maccmu/
      macposts.git
   ```

4. Install MAC-POSTS
   Once the MAC-POSTS repository is ready, switch to the created Python environment in Step 1, and run the following command at the project root to install MAC-POSTS:

   ```
   pip install .
   ```

   For development, run

   ```
   pip install -e .[dev]
   ```

   instead to also install the development tools and enable editable mode. If users also need debug information for the C++ library (and also enable other settings for debugging), set the environment variable `DEBUG=1` before installation.

2.3 Other useful resources

The repository described above contains the core of MAC-POSTS. Another repository `https://github.com/maccmu/macposts-contrib` provides some useful Python scripts using MAC-POSTS to carry out some dynamic network modeling tasks (e.g., dynamic OD demand estimation). Note that if users would like to carry out dynamic OD demand estimation with MAC-POSTS, **PyTorch** is also required to be installed.

3 Work Flow in MAC-POSTS

Figure 1 illustrates the workflow of using MAC-POSTS to perform a complete dynamic network modeling task.

3.1 Data collection

The task starts with collecting both network and traffic data. The raw network data usually comes as GIS files. Users need to extract the following necessary information from GIS files as text files to be read by MAC-POSTS.

- Link attribute (number of lanes, free-flow speed, length, traffic capacity, and jam density)
- Topology (a list of links and their associated two endpoint nodes)
- Origins, destinations, and OD connectors

The traffic data is used to calibrate the dynamic network model. MAC-POSTS is able to take various types of data such as historical demand, traffic count, travel time/speed, and vehicle registration information.

Please note that the data described here is generally required by most dynamic network modeling tools. MAC-POSTS does NOT provide any specific functions to prepare such data. It relies on users to properly collect and process these data as input. More information about the input is discussed in Section 6.

3.2 Dynamic OD demand destination (DODE)

Once the network and traffic data are ready, users can carry out DODE to calibrate the model, which is to estimate time-varying OD demand so that the dynamic model can produce results (e.g., traffic flow) matching the real-world observations. The dynamic OD demand plays a key role in transportation planning and management to understand spatio-temporal vehicular flow and its travel behavior.

Particularly, if multi-class vehicle data is available, MAC-POSTS can provide the multi-class dynamic OD demand (MCDOD), which represents the number of vehicles in each of general vehicle classes (e.g. personal cars, trucks, ride-sourcing vehicles, connected vehicles, etc.) departing from an origin and heading to a destination in a particular time interval. The definition of "classes" is very general, by vehicle sizes, specifications, and nature of trips. The MCDOD reveals the fine-grained traffic demand information for different vehicle classes and the overall spatio-temporal mobility patterns can be inferred from the OD demand and its resultant path/link flows. Policymakers can understand the departure/arrival patterns of multi-class vehicles through

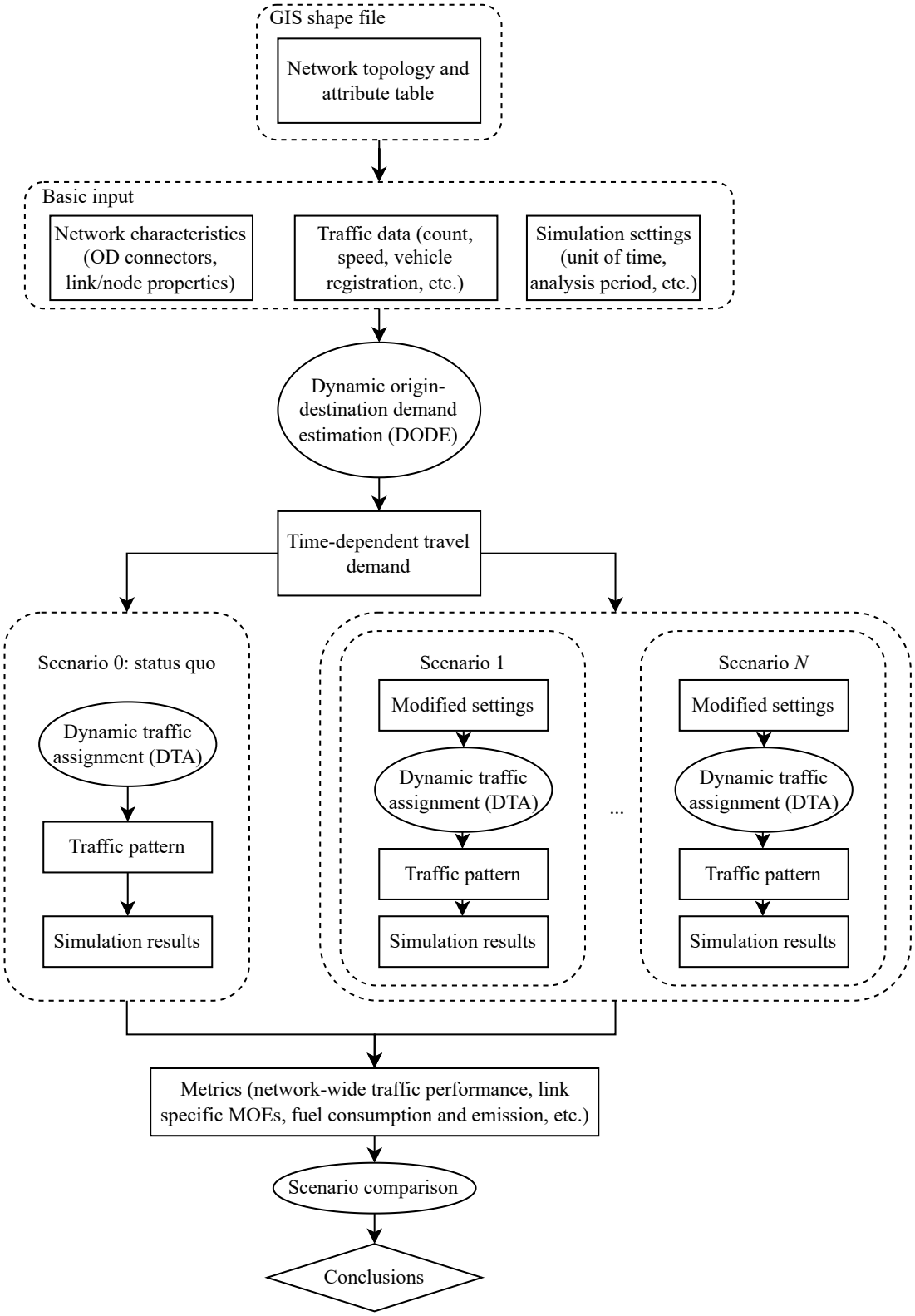

Figure 1: Workflow in MAC-POSTS

the MCDOD. The MCDOD also helps the policymakers understand the impact of each vehicle class on the roads, and hence each class of vehicles can be managed separately. In addition, most Advanced Traveler Information Systems/Advanced Traffic Management Systems (ATIS/ATMS) would require accurate MCDOD as the model input.

MAC-POSTS adopts a state-of-the-art computational-graph-based DODE algorithm which is suitable for large-scale networks with large-scale traffic data. This algorithm is a reformulation of the standard iterative heuristic methods for solving bi-level optimizations, while it provides a novel perspective to view the DODE problem as a machine learning task. Based on the constructed computational graph, the gradient of the loss with respect to the demand is derived. Users can choose from multiple off-the-shelf optimizers in PyTorch (e.g., `Adam`, `NAdam`, and `SGD`) to solve the DODE problem (Paszke et al., 2019). More details about the DODE algorithm can be referred to Ma, Pi, and Qian (2020).

3.3 Dynamic traffic assignment (DTA)

Given the dynamic OD demand, DTA is to assign this demand to routes based on the route choice model. The output of DTA is time-dependent path flows.

Depending on the route choice model, MAC-POSTS has three types of DTA models:

- Dynamic user equilibrium (DUE)
  DUE aims to find a time-dependent flow pattern in a network such that no user can improve his/her experienced travel time by unilaterally switching routes for a given departure time, namely, the temporal extension of the Wardropian UE condition (Peeta & Mahmassani, 1995). MAC-POSTS uses a path-based formulation of DUE and adopts two solution methods: the traditional method of successive averages (MSA) and the closed-form gradient projection (GP) method (Pi, Ma, & Qian, 2019). Note that the GP is usually solved by a quadratic programming solver, which can be computationally expensive. In comparison, the method proposed by (Pi et al., 2019) decomposes the whole problem into independent smaller problems at the OD level, which can be solved in parallel, and solves the problem analytically using the KKT conditions. Therefore, this GP method significantly reduces the computational cost, especially for large-scale networks.

- Dynamic system optimal (DSO)
  DSO aims to determine a time-dependent flow pattern in a network such that the total network cost is minimized. It is well known that under a path-based formulation, the DSO can be transformed into a standard DUE by replacing the path

marginal cost (PMC) with the path cost and then the solution methods for DUE can be applied. The PMC is notoriously difficult to obtain. In MAC-POSTS, the approximation of PMC proposed by (Z. S. Qian, Shen, & Zhang, 2012) is adopted, in which approximate PMCs are obtained by tracing flow perturbations across regular links, merges, and diverges in a network.

- Hybrid routing
  Hybrid routing models two types of travelers. A portion of the travelers, habitual travelers, will take the prescribed route according to their day-to-day experience or anticipated travel time, regardless of real-time traffic conditions on the network, while other travelers, referred to as adaptive travelers, will respond to dynamic traffic conditions and choose their routes as they go. Users have the flexibility to choose the portions of habitual travelers and adaptive travelers in the network so as to produce realistic traffic conditions and network performance. Unlike the DUE and DSO, which usually require many iterations in order to reach convergence, this route choice model is iteration free and thus is more computationally efficient (Z. S. Qian & Zhang, 2013).

3.4 Dynamic network loading (DNL)

The DNL propagates time-dependent path flows through the route links and determines the dynamic traffic pattern as a result of dynamic demand and supply interaction. MAC-POSTS adopts a mesoscopic model that models the vehicles individually while tracking their movements by macroscopic models and thus balances the level of modeling details and the computational cost. The link model and the node model are the most important components of the DNL. The link model characterizes the evolution of vehicle flows on road segments while the node model delineates the merging and diverging behavior of traffic flows on road junctions. In MAC-POSTS, users can choose from different link and node models to establish a DNL model based on their needs.

The DODE, DTA, and DNL are intertwined with each other, and their relationship is depicted in Figure 2. It can be seen that the DNL is responsible for simulating traffic dynamics and is usually the most computationally expensive. DNL is the foundation for DTA and DODE. The DTA solution methods usually iterate between the route choice and the network loading until convergence is achieved (i.e., for DUE and DSO). On top of DTA, DODE aims to adjust the dynamic OD demand so that the simulated traffic pattern can match the real-world observations.

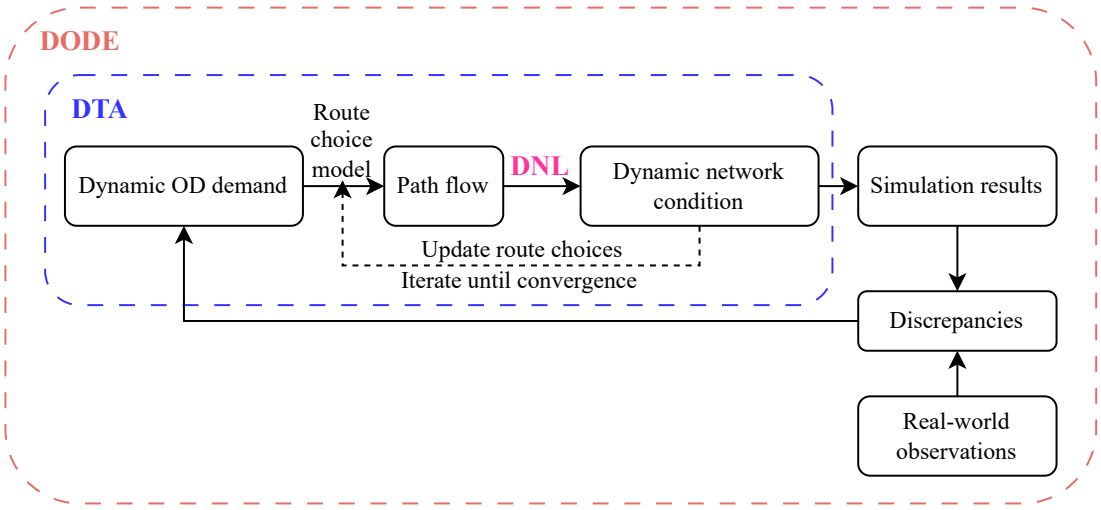

Figure 2: Relationship among DODE, DTA, and DNL

3.5 Scenario analysis and comparison

Once the dynamic model is calibrated with DODE, this model reflects the status quo of the network. Based on this, users can change settings, create different possible scenarios, re-run the DTA/DNL model, and carry out the comparative analysis. This is particularly useful to answer those "what-if" questions. For example, possible scenarios may include but are not limited to, setting tolls on some links, setting up a work zone, increasing/decreasing demand for some OD pairs, and implementing different traffic management strategies. By comparing the corresponding metrics of interest (e.g., congestion and emission) from different scenarios, users can gain a better understanding of the impacts of introducing such changes quantitatively at both disaggregate or aggregate levels and may then draw some conclusions or make decisions.

Please note that this workflow in Figure 1 is for a complete dynamic network modeling task. Users can decide to use only some parts of the workflow based on their needs or data availability. For example, if users already have time-dependent path flows at hand and do not need to carry out DODE or DTA, they can just perform the DNL only.

4 Design framework and implementation

4.1 Design concept

The main goal of MAC-POSTS design is to devise reusable components for fast iterating development. We hope to reduce the amount of codes developers need to read before actually implementing their own models. We achieve this by defining the hierarchy of components in the models, enforcing interfaces for polymorphic components, and managing memory through unified factory classes. The first two strategies help to enhance reusability and pluggability and the third strategy helps to prevent memory leaking during collaborative development.

4.1.1 Hierarchy

MAC-POSTS mainly consists of three parts. The first part is called *component*, which contains separated models such as the dynamic node/link model and routing choice model. Each component has clear interfaces with other components, so developers can implement each component with different actual models. The second part is *algorithm*, which takes relevant components as input and aims to solve specific problems in dynamic network modeling. By combining different components and algorithms, we can establish complex dynamic network models, the third part *model*. The types of models may vary case by case based on different needs.

Component and *algorithm* also have their own hierarchies, which means that one component may consist of several lower-level components and one algorithm may call other algorithms as well. The hierarchy design of the implementation enables the reuse of codes for different purposes. The overview of the hierarchy structure of MAC-POSTS is presented in Figure 3.

As can be seen from Figure 3, *component* has the hierarchy itself, the lower-level components include node, link, vehicle, path, etc., while the higher-level components include factory, path table, path set, etc. Note that Figure 3 only illustrates some core components of MAC-POSTS for the sake of brevity.

4.1.2 Interfaces

For each component and algorithm, it is possible to have different implementations. For example, multiple models can be used to implement the behavior of dynamic node for the component `Node`. Similarly, the algorithm `Shortest_Path` can be implemented by various label-correcting algorithms, such as Dijkstra and First-In-First-Out (FIFO).

Based on inheritance and polymorphism, two important concepts in object-oriented programming, MAC-POSTS has a pluggable design, in which the interface is the key. By clearly defining the interface, the components can be plugged into other

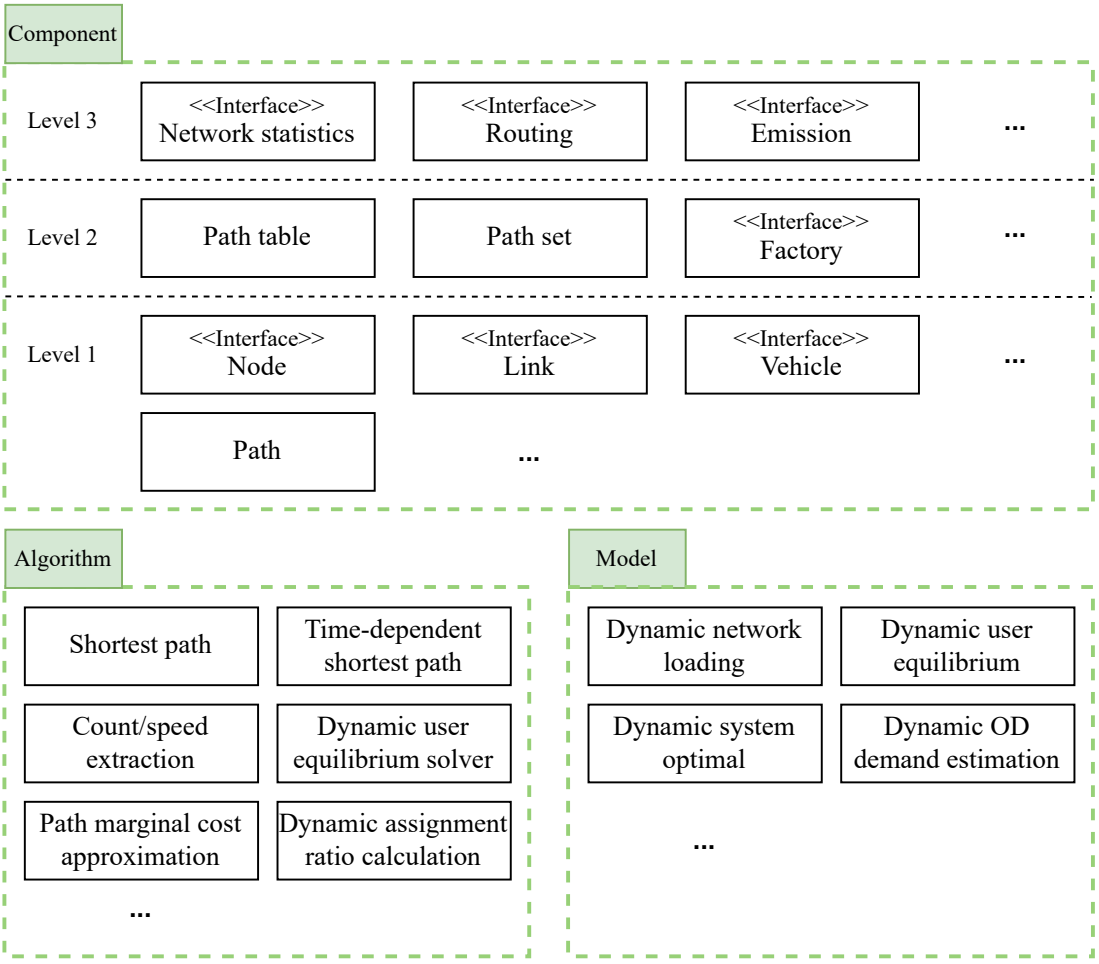

Figure 3: Overview of MAC-POSTS framework

components and algorithms without further modification and adaptation. A typical interface-based design for one component is shown in Figure 4.

As shown in Figure 4, a typical interface consists of two levels, the upper level defines the interfaces and the lower level actually implements the behavior of the components. One upper level may have different lower level implementations. For example, for a dynamic link model (upper level), MAC-POSTS has four different implementations: point queue, link queue, cell transmission model, and link transmission model (lower level). But they all share a unified interface, thus each implementation can be directly plugged into the upper-level model very easily based on the needs.

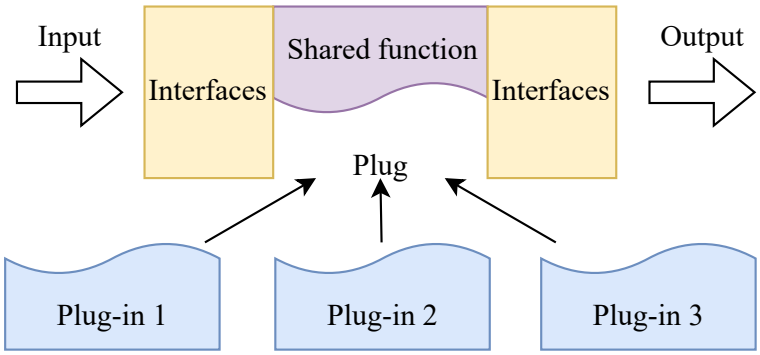

Figure 4: Illustration of a typical interface

4.1.3 Memory management

Memory management is an essential part of collaborative development and fast iteration. MAC-POSTS use a class `Factory` to manage all the memory used. In MAC-POSTS, different components usually have their corresponding `Factory` classes. For example, the component `Link` has a `Link_Factory` and the component `Node` has a `Node_Factory`. All the memory used by MAC-POSTS is allocated and released by those `Factory` objects, and all the components and algorithms are not allowed to allocate memory for the whole model. In this way, the memory can be managed in a centralized manner.

4.2 Components

This section demonstrates some major components implemented in MAC-POSTS, and discuss how the reusability and pluggability are achieved in the implementations.

4.2.1 Node

Node models describe the behavior of road junctions with respect to vehicle inflow and outflow. Two actual models are implemented here: virtual demand and supply based model (Pi et al., 2019) and critical demand level based model (Jin, 2017),

as shown in Figure 5.

In the interface, it has `in_link_array` and `out_link_array` to represent the upstream and downstream links, respectively. It also has a function called `evolve()` to simulate the traffic flow evolution at the node. Here `evolve()` is just an interface and is actually realized in the lower-level implementations. *Virtual SD Junction* model calculates the `virtual_supply` and `virtual_demand` in the class itself and then implements the `evolve()` function. Similarly, *General Road Junction* model also calculates what it needs, such as `critial_demand_level` and then implements the `evolve()` function.

To use the node components, users only need to know the interface of the upper level implementations, they do not need to understand how the actual node flow is calculated in the lower level models. Also, users can easily switch among different node models to compare the performance of different models.

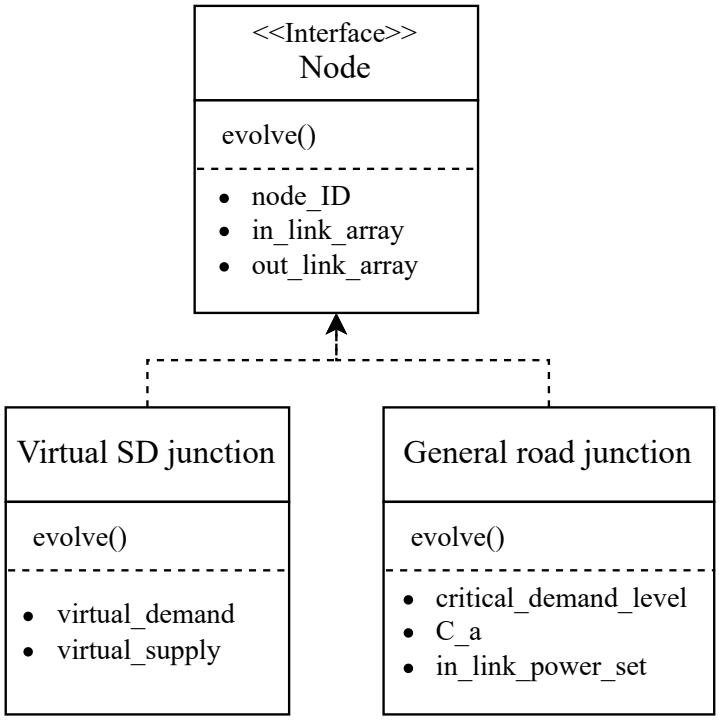

Figure 5: Overview of node component

4.2.2 Link

Four dynamic link models are implemented for the link component, but similar to the node component, they all have a unified interface, as presented in Figure 6.

In the link interface, `evolve()` function simulates the evolution of traffic flows on the link, `get_demand()` and `get_supply()` obtain the current traffic conditions on the links. These functions are the interfaces of the link components. For example, when node component calculates the virtual demand, it will call the `get_demand()` function on each upstream link and `get_supply()` function on

each downstream link to get model inputs, then conduct the rest of the calculations. The link interface also has some shared variables such as capacity that are commonly used in different models. Note that another component *cumulative curve* is also implemented and added to the link model, which is often used to record the cumulative flow for the link.

For different lower level implementations, *Point Queue* model implements the bottleneck behavior on the link, so it uses a `trace_tracker` to keep track of the bottleneck configuration and queue condition. *Cell Transmission Model* further segments the link to be a sequence of cells, which is implemented by *CTM Cell*. Each cell contains a vehicle queue and a `evolve()` function, the `evolve()` function in CTM will just call the `evolve()` function of each cell in the `cell_list`. *Link Transmission Model* does not discretize the link into small cells and instead relies on cumulative curves to calculate the in/out flows. *Link Queue* model only relies on the density of the link, so its `evolve()` is implemented directly from `vehicle_queue`.

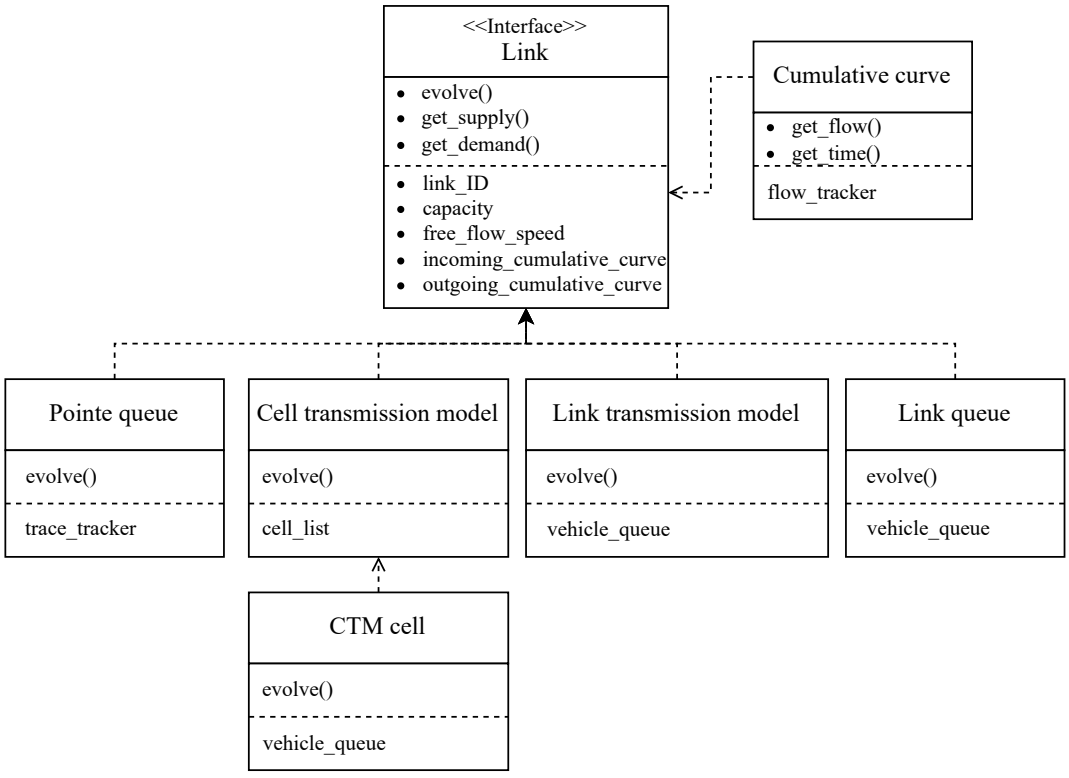

Figure 6: Overview of link component

Moreover, for modeling the heterogeneous vehicle flow on links, we adopt a multi-class traffic flow model proposed in S. Qian, Li, Li, Zhang, and Wang (2017), which can model the flow dynamics consisting of multiple classes of vehicles with distinct flow characteristics. It pragmatically generalizes the cell transmission model to multi-class heterogeneous vehicle flow. It includes the concept "physical space split" for each class, which is the fraction of physical space that each vehicle class

occupies and uses to progress. Then the "perceived equivalent density" of each class is calculated, representing the equivalent density perceived by some vehicle class, if converting all other class vehicles to this class based on the space they occupied. At each loading time interval, vehicles move through cells following the relations between upstream demand and downstream supply computed using the "physical space split" and "perceived equivalent density", as well as the fundamental diagram of each class. The main feature of this multi-class flow model is that it encapsulates three mixed flow regimes: one class can overtake the other class under free flow, overtaking occurs restrictively under semi-congestion, and no overtaking can occur under congestion. More details can be found in S. Qian et al. (2017).

4.2.3 Routing

The routing component also has a unified interface. Function `init_routing()` initializes the routing before the simulation and function `update_routing` updates the routing policy for each vehicle when necessary in the simulation. `graph` is another component used to represent the connectivity of links and nodes. `od_factory` is a unified interface used to access the origins and destinations of the network, `node_factory` is a unified interface used to access the nodes of the network, `link_factory` is a unified interface used to access the link conditions of different links, `vehicle_factory` is a unified interface used to access each vehicle on the network. The overview of the routing component is presented in Figure 7.

The routing component has three implementations. *Fixed Routing* simply routes the vehicles to the prescribed path. So it will have a `path_table` to track all the paths, which relies on the *Path Set* component. *Path Set* component is further dependent on the *Path* component. *Adaptive Routing* uses the real-time information to route vehicles to the shortest path at the current time step, so it takes the *Network Statistics* as its subcomponent, and uses `shortest_path` algorithm to generate the routes. As for *Hybrid Routing*, it is just the combination of *Fixed Routing* and *Adaptive Routing* and has an attribute `adaptive_ratio` to decide the possibility of a vehicle using adaptive routing.

Recall that Although we mentioned that MAC-POSTS has three routing choice models in Section 3: DUE, DSO, and hybrid routing, DUE and DSO are all iteration-based algorithms and in each iteration, they just use *Fixed Routing*.

4.2.4 Vehicle

The overview of vehicle component is depicted in Figure 8. Since MAC-POSTS uses a mesoscopic DNL model, each vehicle is treated as an agent. It has corresponding attributes to record origin, destination, departing time, etc., which are useful to reconstruct vehicle trajectories if necessary. Particularly, each vehicle has an attribute `class` to indicate the class of this vehicle (e.g., car or truck), based on which,

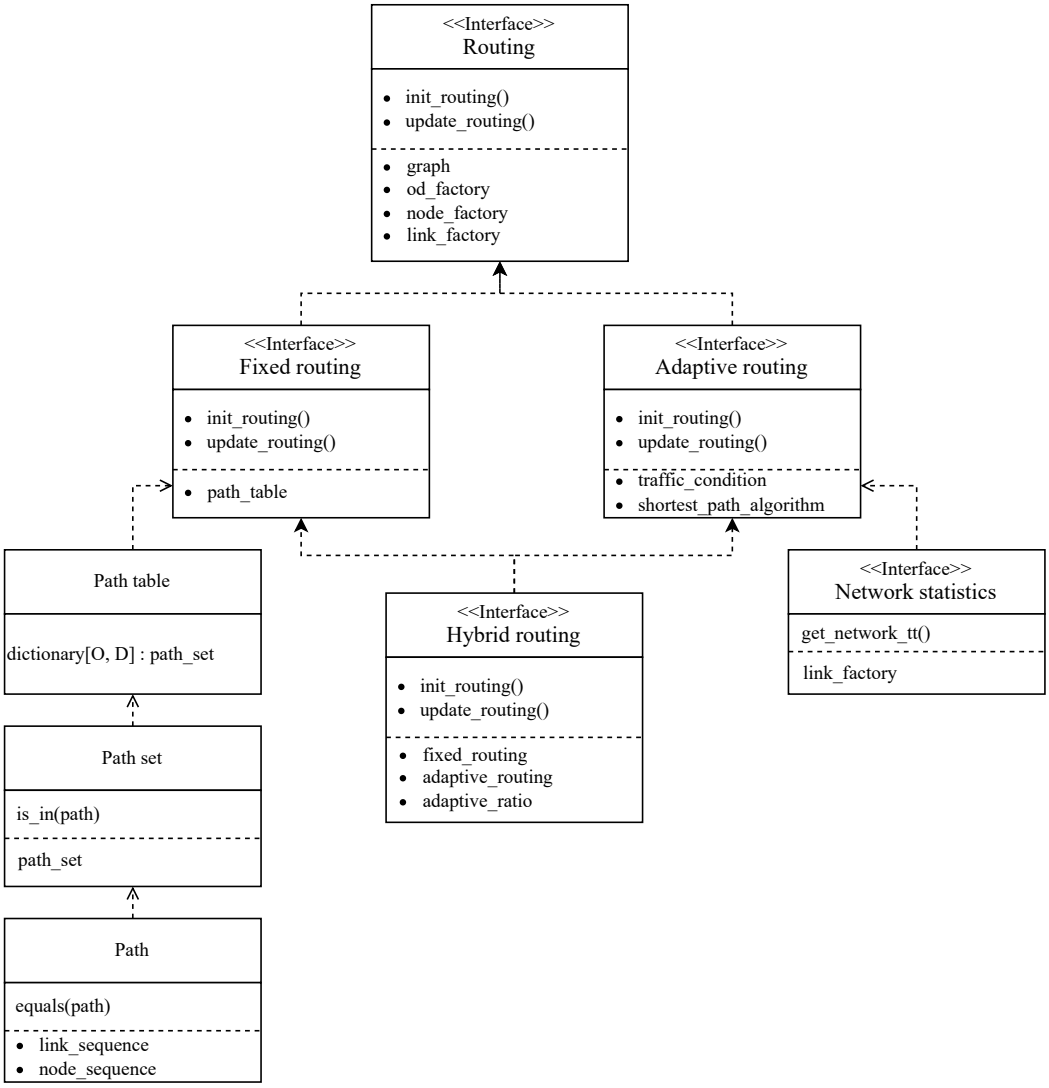

Figure 7: Overview of routing component

traffic dynamics on nodes and links and fuel consumption and emission calculations are different.

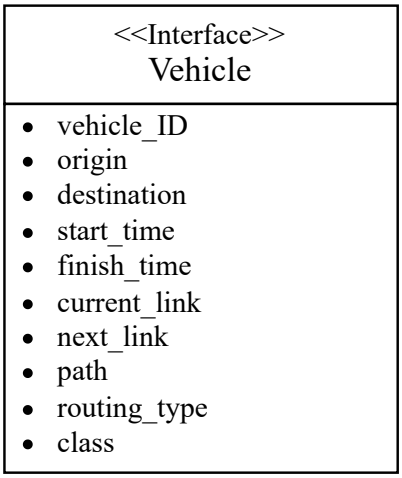

Figure 8: Overview of vehicle component

Figure 9 further illustrates traffic dynamics modeling with node, link, and vehicle components, where the arrows represent how we move different classes of vehicles within the links and among different links through the nodes.

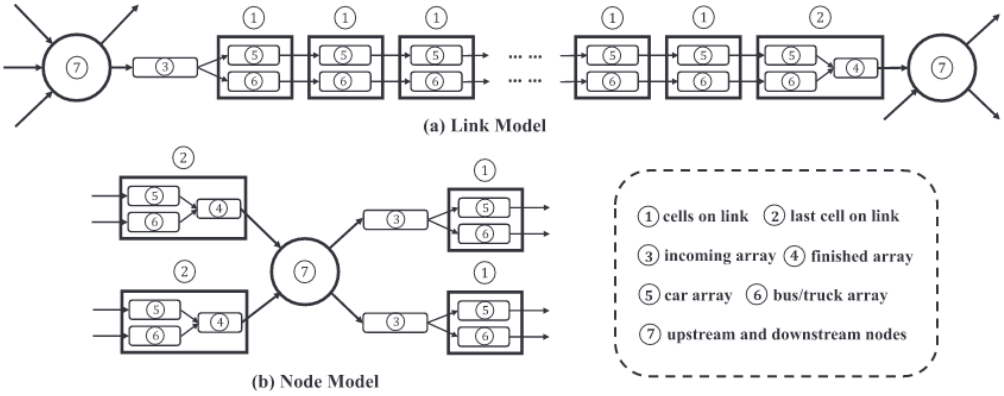

Figure 9: Illustration of modeling traffic dynamics with node, link, and vehicle components

4.2.5 Other components

There are also other components implemented in MAC-POSTS for specific tasks, e.g., *DMS* to model the Dynamic Message Signs, *Workzone* to model the construction zone on the link, and *IO* to manage the file Input/Output. All the components are interface based, so they can be easily plugged into different models without any modifications and adaptations.

Moreover, When developers encounter a new requirement or issue that the current code cannot address, they only need to implement the corresponding components

and design compatible interfaces, then they can combine the newly implemented components with existing components to build the new dynamic network models without drastic modifications to the existing code.

4.3 Algorithms

This section briefly discusses some algorithms in MAC-POSTS. The major difference between the algorithm and the component is that algorithm does not allocate memory itself while the component allocates all the memory for the models. Algorithms can be viewed as operators over different components.

4.3.1 Shortest path

Shortest Path algorithm is used to find the shortest path from one origin node to one destination node given the link cost (and possible node cost) of the network. Note that in the dynamic network modeling context, the link cost (and possible node cost) usually represents the network state at one particular time step. MAC-POSTS implements two label-correcting shortest path algorithms: *Dijkstra* and *FIFO*. And they share the same interface.

Input

- destination node ID
- graph: network connectivity
- link travel time: a list of (`link_ID`, `link_time`) pairs
- node travel time (optional): a list of (`in_link_ID`, `out_link_ID`, `node_time`) pairs
- link travel cost: a list of (`link_ID`, `link_cost`) pairs
- node travel cost (optional): a list of (`in_link_ID`, `out_link_ID`, `node_cost`) pairs

Output

A list of (`node_ID`, `next_link_ID`) pairs representing the next link to traverse at `node_ID`. And the shortest path from the origin of interest to the destination can be extracted from the output list.

Note that we differentiate the travel cost from the travel time here. This is because the travel cost usually can be defined in a more general sense. For example, we can define travel cost $=$ Value of time $*$ travel time $+$ other terms (e.g., toll). The shortest path algorithm aims to find the path with the minimum travel cost.

4.3.2 Time-dependent shortest path

Finding the time-dependent shortest path is different from finding the shortest path described above in that the time-varying link (and node) cost should be taken into account. Interested readers can refer to the discussion of "experienced travel time" and "instantaneous travel time" in Chiu et al. (2011).

MAC-POSTS implemented the decreasing order of time (DOT) algorithm to find the time-dependent shortest path (Z. S. Qian et al., 2012; Nie, 2006).

Input

- destination node ID
- graph: network connectivity
- total time intervals: N
- time-dependent link travel time: a list of (`link_ID`, `link_time_1`, `link_time_2`, `link_time_3`, ...) arrays of length $N+1$ representing the time-varying link travel times
- time-dependent node travel time (optional): a list of (`in_link_ID`, `out_link_ID`, `node_time_1`, `node_time_2`, `node_time_3`, ...) arrays of length $N+1$ representing the time-varying node travel time
- time-dependent link travel cost: a list of (`link_ID`, `link_cost_1`, `link_cost_2`, `link_cost_3`, ...) arrays of length $N+1$ representing the time-varying link travel costs
- time-dependent node travel cost (optional): a list of (`in_link_ID`, `out_link_ID`, `node_cost_1`, `node_cost_2`, `node_cost_3`, ...) arrays of length $N+1$ representing the time-varying node travel costs

Output

A list of (`node_ID`, `next_link_ID_1`, `next_link_ID_2`, `next_link_ID_3`, ...) arrays of length $N+1$ representing the next link to traverse at time step n and `node_ID`. And the time-dependent shortest path from the origin of interest to the destination departing at any time can be extracted from the output list.

4.3.3 Count/speed extraction

After the simulation is done, the time-varying traffic count and the travel time (or speed) of links can be extracted from the cumulative curves of links, such results can be further used in DODE and calculating other metrics. Here we assume that FIFO is strictly enforced dynamic link models. The DNL model here requires keeping the

order in which vehicles enter the link, and a queue structure is used to store the order of those vehicles. Figure 10 illustrates how the extraction is performed.

In the link component, we mentioned that each link has two cumulative curves, i.e., `incoming_cumulative_curve` and `outgoing_cumulative_curve` (see Figure 6), which are used to record cumulative numbers of vehicles entering and leaving the link, respectively, and correspond to the left curve and the right curve in Figure 10, respectively.

To calculate the number of vehicles traversing this link from t_1 to t_2, simply extract the two count readings of the `incoming_cumulative_curve` at t_1 and t_2, then $c_2 - c_1$ is the solution.

To calculate the link travel time at t_1, first extract the count reading, c_1, at t_1 for the `incoming_cumulative_curve`, and then find the time stamp t_3 for the `outgoing_cumulative_curve` so that the count reading at t_3 is equal to c_1. $t_3 - t_1$ is the desired link travel time. To obtain the speed, simply divide the link length by the calculated link travel time.

Note that all these procedures rely on the two functions of the cumulative curve component, `get_time()` and `get_flow()`, as shown in Figure 6.

It is also worth pointing out that since the mesoscopic DNL model discretizes time and vehicle flows, the cumulative curves in practice are not smooth curves but exhibit staircase shapes. Therefore, some rounding procedures are developed in extracting count/speed.

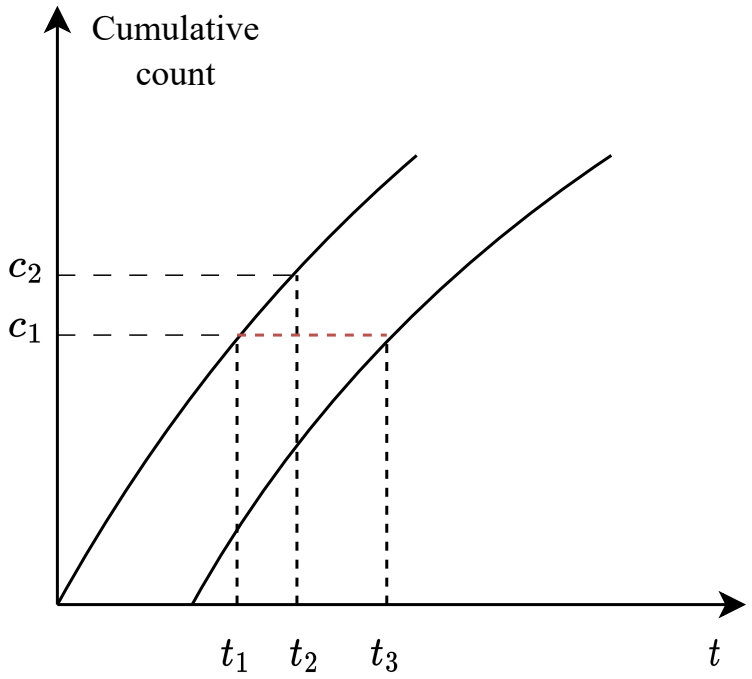

Figure 10: Illustration of count/speed extraction from cumulative curves

4.3.4 DUE solver

DUE aims to find a time-dependent flow pattern in a network such that no user can improve his/her experienced travel time by unilaterally switching routes for a given departure time. The DUE solver is iteration-based. Within one iteration, the DNL is performed first based on the path flows from the previous iteration and then the path flow is adjusted based on some criteria. This procedure goes on until the convergence is met.

MAC-POSTS adopts two solution methods: the traditional method of successive averages (MSA) and the closed-form gradient projection (GP) method (Pi et al., 2019). Note that the GP is usually solved by a quadratic programming solver, which can be computationally expensive. In comparison, the method proposed by (Pi et al., 2019) decomposes the whole problem into independent smaller problems at the OD level, which can be solved in parallel, and solves the problem analytically using the KKT conditions. Therefore, this GP method significantly reduces the computational cost, especially for large-scale networks.

As explained in 3, the DUE solver can be used to solve both DUE and DSO problems. MAC-POSTS implements the DUE solver under the *DUE* model. Function `update_path_table_fixed_departure_time_choice()` is for the MSA while function `update_path_table_gp_fixed_departure_time_choice()` for the GP.

4.3.5 PMC approximation

The basic idea of PMC approximation is to trace the link cumulative curve change by introducing a perturbation vehicle to the original traffic. Based on Z. S. Qian et al. (2012), the PMC for the CTM link is simpler than that for the PQ link

$$\mathrm{PMC}_{pt}^{rs} = \bar{sd}_{pt}^{rs} + \sum_{\xi}(t_{a_\xi}^{C} - t_{a_\xi}^{A}) + \sum_{\xi}\mathrm{fft}_{a_\xi} \tag{1}$$

where rs is the OD pair index, p is the path index, t is the time index, $\bar{sd}$ is the scheduled delay cost, fft is the free-flow travel time, $t_{a_\xi}^{A}$ is the time the perturbation occurs on the arrival curve of link a_ξ, and $t_{a_\xi}^{C}$ when the queue vanishes for the first time after $t_{a_\xi}^{A}$. Note that the vehicle at $t_{a_\xi}^{A}$ may not be the additional vehicle at the original perturbation due to the existence of bottlenecks. MAC-POSTS implements the PMC approximation in the function `get_link_marginal_cost()` under the *DSO* model.

4.3.6 Dynamic assignment ratio (DAR) calculation

Based on the definition of Ma et al. (2020), the DAR $\rho_{rsi}^{ka}(h_1, h_2)$ denotes the portion of the k-th path flow departing within time interval h_1 for OD pair rs which arrives at link a within time interval h_2 for vehicle class i. Computing the DAR matrix

is a key step in DODE. The DAR matrix is usually extracted from the DNL results. Computing the DAR matrix during the simulation is more efficient than obtaining the DAR matrix after the simulation, but since the dimension of DAR matrix increases exponentially with respect to the size of network and the number of time intervals, the naive method is computationally implausible for large-scale networks. Based on Ma et al. (2020), MAC-POSTS computes the DAR matrix through the tree-based cumulative curves, which is more efficient in both computational time (time complexity) and memory (space complexity). MAC-POSTS has a *tree-based cumulative curve* component for this purpose.

4.4 Models

As mentioned before, MAC-POSTS offers different components and algorithms for users to establish different dynamic models based on their needs. This subsection introduces some common models.

4.4.1 DNL

The DNL model is usually the foundation for other dynamic modeling tasks (e.g., DTA and DODE) and characterizes the evolution of the network condition over time. It is a combination of many components such as *node*, *link*, *vehicle*, *routing*, and *network statistics*. Figure 11 illustrates the structure of the DNL model.

Function `load_once()` loads the demand onto the network based on the route choice model and simulates the traffic dynamics at each time step during the analysis period. Function `loading()` models the evolution of the network conditions by calling `load_once()` multiple times until the end of the analysis period.

MAC-POSTS implements `Dta` and `Dta_Multiclass` models for single-class and multiclass DNL, respectively.

4.4.2 DUE

The DUE model is a combination of the DNL model, the DUE solver, and the time-dependent shortest path algorithm. In each iteration, the results of the DNL model are used to compute the link/path costs, based on which, `compute_gap()` computes the equilibrium gap and determines whether convergence is met. `update_path_table()` may add new time-dependent shortest paths to the path table and `update_path_flow()` adjusts the path flows based on the criteria specified in the DUE solver.

4.4.3 DSO

The DSO model is built on top of the DUE model with an additional function `get_link_marginal_cost()`, as shown in Figure 13.

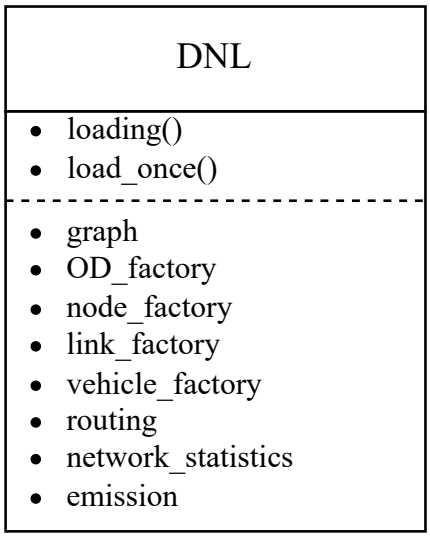

Figure 11: Overview of DNL model

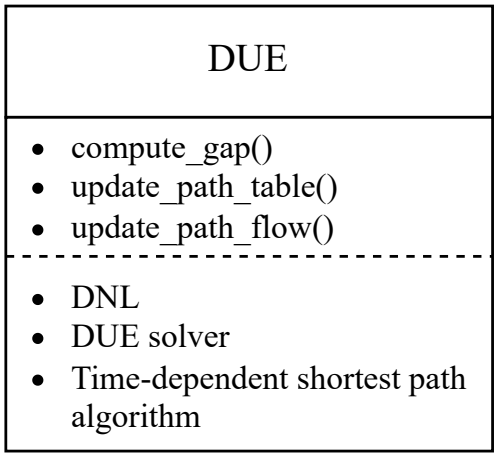

Figure 12: Overview of DUE model

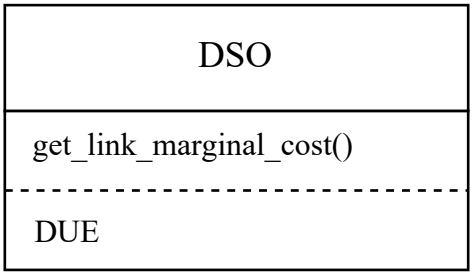

Figure 13: Overview of DSO model

4.4.4 DODE

Figure 14 illustrates the basic structure of the DODE model. The DTA model can be DUE, DSO, and hybrid routing. Recall that the hybrid routing model is iteration-free and thus is less computationally expensive than DUE and DSO models. In each iteration of DODE, function `get_dar()` constructs the DAR matrix based on the DTA results and DAR calculation algorithm. The DAR matrix is then used in function `get_gradient()` to obtain the gradient of the loss with respect to the demand. Function `update_demand()` updates the demand based on the gradient and the optimizer selected from PyTorch. Function `compute_loss()` calculates the loss.

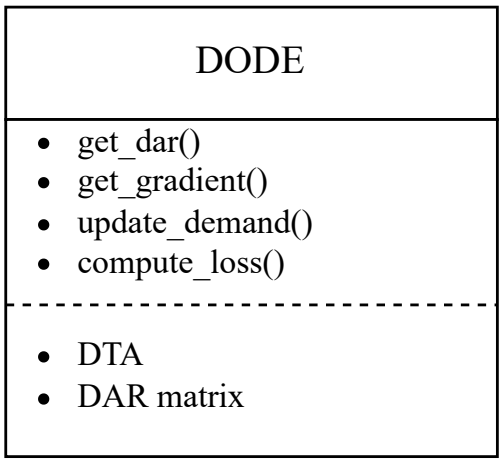

Figure 14: Overview of DODE model

5 File structure

This section introduces the file structure of MAC-POSTS. It is mainly to provide references for developers on how files are organized.

5.1 Overall structure

At the root of MAC-POSTS,

- `examples` (folder)
  Some examples of using MAC-POSTS
- `lib` (folder)
  Three open-source third-party libraries used:
  - SNAP[1]: used to create the `graph` component and perform some operations
  - Eigen[2]: used to create sparse matrices
  - pybind11[3]: used to create Python extension
- `macposts` (folder)
  Python APIs exposed to normal users
- `src` (folder)
  The core C++ source code
- `tests` (folder)
  Some test examples
- `.github/workflows` (folder)
  A directory in a GitHub repository that contains files defining continuous integration and deployment (CI/CD) workflows for the repository. Workflows typically consist of a series of steps that define tasks to be performed, such as building and testing the code, deploying to a staging environment, and publishing the code to a production environment. Steps can use various actions provided by GitHub or custom actions defined by the repository.
- `CMakeLists.txt` (file)
  A file used by the CMake build system to define how to build a project. It specifies the project name, version, dependencies, source files, libraries, executables, and other build options.

[1] `https://snap.stanford.edu/snap/index.html`
[2] `https://eigen.tuxfamily.org/index.php?title=Main_Page`
[3] `https://github.com/pybind/pybind11`

- `setup.cfg` (file)
  A configuration file used to specify metadata about a Python package, such as its name, version, author, license, and dependencies. It also allows configuring various aspects of package installation, such as which files to include, which packages to exclude, and which scripts to install.

- `setup.py` (file)
  A Python script used to build, package, and distribute a Python package. It reads the metadata and configuration from `setup.cfg`, and defines how the package should be built and distributed.

- `MANIFEST.in` (file)
  A file used in Python projects to specify additional files to include in the source distribution package that are not automatically included by default (in `setup.py`) but are necessary for the package to function properly, such as configuration files, data files, or documentation.

- `pyproject.toml` (file)
  A configuration file used in Python projects to define project metadata, build settings, and dependencies. It provides a standardized way to define project metadata, build settings, and dependencies that can be used across different build systems and packaging tools.

- `.gitmodules` (file)
  A file specifying directories and git repositories of the third-party libraries. It allows developers to keep a git repository as a subdirectory of another git repository. Git submodules are simply a reference to another repository at a particular snapshot in time. Git submodules enable a Git repository to incorporate and track version history of external code[4]. Note that due to some compilation issues, we maintained a modified SNAP here: `https://github.com/maccmu/snap`.

- `.gitignore` (file)
  A file used in Git repositories to specify files and directories that should be excluded from version control.

- `LICENSE` (file)
  License information. MAC-POSTSis licensed under the MIT License

- `README.md` (file)
  A file briefly introducing MAC-POSTS.

[4] `https://www.atlassian.com/git/tutorials/git-submodule`

5.2 C++ files inside `src` folder

Folder `src` contains the core C++ source code of MAC-POSTS. This subsection briefly introduces the contents of some important files in this folder. Please note that MAC-POSTS is under constant development, the code within these files in this folder can be updated frequently and more files corresponding new features may be added.

- `cc_tree.cpp`
  Tree-based cumulative curve component.
- `dlink.h, dlink.cpp`
  Cumulative curve component and *Link* component, including PQ, LQ, CTM, and LTM models.
- `dnode.h, dnode.cpp`
  Node component, including virtual SD junction and general road junction models.
- `dso.h, dso.cpp`
  DSO model, including PMC approximation algorithm
- `dta.h, dta.cpp`
  Single-class DNL model.
- `dta_gradient_utls.h, dta_gradient_utls.cpp`
  Some utility functions used to compute traffic count, travel time, and DAR record.
- `due.h, due.cpp`
  DUE model.
- `emission.h, emission.cpp`
  Emission component to calculate fuel consumption and emission.
- `enum.h`
  Some defined categorical variables.
- `factory.h, factory.cpp`
  Various *factory* components for link, node, vehicle, OD, etc.
- `gridlock_checker.h, gridlock_checker.cpp`
  Gridlock checker component used to detect the possible existence of gridlock.

- `io.h, io.cpp`
  IO component to handle input/output.
- `multiclass.h, multiclass.cpp`
  Relevant components and *multiclass DNL* model for multiclass modeling.
- `multimodal.h, multimodal.cpp`
  Relevant components and *multimodal DNL* model for multimodal modeling.
- `od.h, od.cpp`
  Origin and *destination* components.
- `path.h, path.cpp`
  Path, *path set*, and *path table* components.
- `routing.h, routing.cpp`
  Routing components, including fixed, adaptive, and hybrid routing models.
- `shortest_path.h, shortest_path.cpp`
  Shortest path and *time-dependent shortest path* algorithms.
- `statistics.h, statistics.cpp`
  Network statistics component to monitor the network conditions.
- `ults.h, ults.cpp`
  Some other utility functions.
- `vehicle.h, vehicle.cpp`
  Vehicle component.
- `workzone.h, workzone.cpp`
  Work zone component.

6 Input files

6.1 Input files for DTA

The DTA is the core simulation part in dynamic network modeling tasks. MAC-POSTS requires a folder including the following text files as input:

- `record` (folder, required)
- `Snap_graph` (file, required)
- `config.conf` (file, required)
- `MNM_input_demand` (file, required)
- `MNM_input_od` (file, required)
- `MNM_input_link` (file, required)
- `MNM_input_node` (file, required)
- `path_table` (file, depending on route choice model)
- `path_table_buffer` (file, depending on route choice model)
- `MNM_input_emission_linkID` (file, optional)
- `MNM_input_link_toll` (file, optional)
- `MNM_input_workzone` (file, optional)
- `MNM_origin_label_car` (file, optional)
- `MNM_origin_label_truck` (file, optional)

We use a 7-link multiclass DTA as an example to introduce these files.

6.1.1 `record`

It can be an empty folder used to store some DTA results. The folder name is specified in parameter `rec_folder` under section `[STAT]` in file `config.conf`.

6.1.2 Snap_graph

It contains the network topology. See the following for an example. The file name is specified in parameter `network_name` under section `[DTA]` in file `config.conf`.

```
#EdgeId FromNodeId ToNodeId
1 1 2
2 2 3
3 2 4
4 3 5
5 4 5
6 3 4
7 5 6
```

6.1.3 config.conf

It contains the DTA settings. See the following for an example and also the explanation of each parameter. A line starting with "#" denotes an explanation of the parameter in the next line. It includes four main sections: `DTA`, `STAT`, `FIXED`, and `ADAPTIVE`.

```
[DTA]
# file name for constructing directed graph for driving
network_name = Snap_graph
# loading interval, unit of time
unit_time = 5
# total loading intervals, -1: until network is empty
total_interval = -1
# number of loading intervals in each assign interval
assign_frq = 180
# starting assignment interval
start_assign_interval = 0
# total number of assignment intervals
max_interval = 10
# vehicle quanta
flow_scalar = 10
# number of links, number of rows in MNM_input_link
num_of_link = 7
# number of nodes, number of rows in MNM_input_od
num_of_node = 6
# number of origins, number of rows after "#Origin_ID <-> node_ID
    " in MNM_input_od
num_of_O = 1
# number of destinations, number of rows after "#Dest_ID <->
    node_ID" in MNM_input_od
num_of_D = 1
# number of OD pairs, number of rows in MNM_input_demand
```

```
OD_pair = 1
# ratio of cars using adaptive routing
adaptive_ratio_car = 0.4
# ratio of trucks using adaptive routing
adaptive_ratio_truck = 0.4

# route choice model, Biclass_Hybrid is the most flexible one, by
    tweaking adaptive_ratio_car and adaptive_ratio_truck, it can
   be changed to pure fixed or adaptive routing
routing_type = Biclass_Hybrid

# initial demand split mode affecting releasing vehicles: 0 = as
   is, 1 = uniform
init_demand_split = 0

# number of labels for cars (mainly used to label electrified
   vehicles in calculating emission), number of columns in
   MNM_origin_label_car
num_of_car_labels = 3
# number of labels for cars (mainly used to label electrified
   vehicles in calculating emission), number of columns in
   MNM_origin_label_truck
num_of_truck_labels = 2
# label for electrified car
ev_label_car = 1
# label for electrified truck
ev_label_truck = 0

# number of tolled links, number of rows in MNM_input_link_toll
num_of_tolled_link = 3


[STAT]
# network statistics recording mode, LRn: least recently n
   intervals
rec_mode = LRn
# For LRn, it represents the recording interval in number of
   loading intervals
rec_mode_para = 12
# folder name
rec_folder = record

# whether to record link volume
rec_volume = 0
# whether to record link volume in each loading interval
volume_load_automatic_rec = 0
# whether to record link volume in each recording interval
```

```
volume_record_automatic_rec = 0

# whether to record link travel time, when using adaptive routing
    , this should be 1
rec_tt = 1
# whether to record link travel time in each loading interval
tt_load_automatic_rec = 0
# whether to record link travel time in each recording interval
tt_record_automatic_rec = 0

# whether to record possible links that yield gridlocks
rec_gridlock = 0

[FIXED]
# file name for path table
path_file_name = path_table
# number of paths, number of rows in path_table
num_path = 3
# path flows or path flow ratios in path_table_buffer, -1: does
    not use buffer as choice portion
choice_portion = Buffer
# dimension of each path flow, for multiclass DTA, it is 2 *
    max_interval representing the coexistence of cars and trucks,
    number of columns in path_table_buffer
buffer_length = 20
# routing intervals, for habitual users, this equals to
    assign_frq (assignment interval)
route_frq = 180

[ADAPTIVE]
# routing intervals, how often the adaptive users search for
    routes, number of loading intervals
route_frq = 180
# value of time, money / hour
vot = 20
```

6.1.4 MNM_input_demand

It contains the time-dependent demand for each OD pair. See the following for an example.

```
#Origin_ID Destination_ID <car demand by interval> <truck demand
    by interval>
1 1 300 300 300 300 300 300 300 300 300 100 100 90 150 70 120 70
    80 60 90 40
```

In this example, only one OD pair (1, 1) has demand, namely, one record

in this file, which is specified in parameter OD_pair under section [DTA] in file config.conf. In the biclass modeling, the demand has two parts: the first half is for car and the second half is for truck. And the length of each part is equal to the total number of assignment interval specified in parameter max_interval under section [DTA] in file config.conf.

6.1.5 MNM_input_od

It includes the origin IDs, destination IDs, and the associated node IDs. See the following for an example.

```
#Origin_ID <-> node_ID
1 1
#Dest_ID <-> node_ID
1 6
```

This example shows that only one origin exists, specified in parameter num_of_O under section [DTA] in file config.conf, and only one destination exists, specified in parameter num_of_D under section [DTA] in file config.conf. Note that origin/destination has its own ID, which can be different from the associated node ID.

6.1.6 MNM_input_link

It includes the link attributes. See the following for an example. The second column "Type" specifies the link model, which can be PQ, CTM, LQ, or LTM. The last column "Convert_factor" is for the multiclass modeling. The total number of links corresponds to parameter num_of_link under section [DTA] in file config.conf.

```
#ID Type LEN(mile) FFS_car(mile/h) Cap_car(v/hour) RHOJ_car(v/
   miles) Lane FFS_truck(mile/h) Cap_truck(v/hour) RHOJ_truck(v/
   miles) Convert_factor(1)
1 PQ 1 99999 99999 99999 1 99999 99999 99999 2.1
2 CTM 0.55 35 2200 200 2 25 1200 80 2.1
3 CTM 0.55 35 2200 200 2 25 1200 80 2.1
4 CTM 0.55 35 2200 200 1 25 1200 80 2.1
5 CTM 0.55 35 2200 200 1 25 1200 80 2.1
6 CTM 0.55 35 2200 200 1 25 1200 80 2.1
7 PQ 1 99999 99999 99999 1 99999 99999 99999 2.1
```

6.1.7 MNM_input_node

It includes the node attributes. See the following for an example. The second column "Type" specifies the node model. Origin and destination nodes are indicated by DMOND and DMDND, respectively. For other junction nodes, FWJ indicates the virtual SD junction model and GRJ corresponds to the general road junction model. The

last column "Convert_factor" is for the multiclass modeling. The total number of nodes corresponds to parameter num_of_node under section [DTA] in file config.conf.

```
#ID Type Convert_factor(only for Inout node)
1 DMOND 2.1
2 FWJ 2.1
3 FWJ 2.1
4 FWJ 2.1
5 FWJ 2.1
6 DMDND 2.1
```

6.1.8 path_table

It includes paths for all OD pairs, mainly used for the fixed routing model. See the following for an example. This example only has one OD pair and three paths. Note that for DUE and DSO models, if column generation is used, this path_table is optional since new paths can be generated on the fly. The related parameters path_file_name and num_path can be found under section [FIXED] in file config.conf.

```
1 2 3 5 6
1 2 4 5 6
1 2 3 4 5 6
```

6.1.9 path_table_buffer

It includes path flows corresponding to the paths in file path_table, which is also mainly used for the fixed routing model. See the following for an example. The related parameters choice_portion and buffer_length can be found under section [FIXED] in file config.conf. In the biclass modeling, each path flow (one row) has two parts: the first half is for car and the second half is for truck. And the length of each part is equal to the total number of assignment interval specified in parameter max_interval under section [DTA] in file config.conf.

```
30 30 30 30 30 10 11 10 10 33 30 30 30 30 30 10 10 10 10 33
40 40 40 40 40 10 11 10 10 33 40 40 40 40 40 10 10 10 10 33
40 40 40 40 40 80 810 80 80 34 40 40 40 40 40 820 80 80 80 33
```

6.1.10 MNM_input_emission_linkID

This file is optional and is used to calculate fuel consumption and emission. It includes the IDs of links participating in this calculation. See the following for an example.

```
2
3
```

```
4
5
6
```

6.1.11 MNM_input_link_toll

This file is optional and is used to input tolled links. See the following for an example, which set up tolls for links 2, 4, and 5. Each row records tolled link ID, the toll for car, and the toll for truck. The related parameter is `num_of_tolled_link` under section `[DTA]` in file `config.conf`. When `num_of_tolled_link` is set to `0`, this file will be ignored.

```
#link_ID toll_car(USD) toll_truck(USD)
2 5 10
4 5 10
5 3 15
```

6.1.12 MNM_input_workzone

This file is optional and is used to input links with work zones. See the following for an example, which set up a work zone on link 3. The first line indicates the total number of links with the work zone. Starting from the second line, each line records the ID of the link with the work zone.

```
1
3
```

6.1.13 MNM_origin_label_car

This file is optional and is used to input ratios of different types of cars for each origin. See the following for an example, in which 10%, 30%, and 60% of cars departing from origin 1 are labeled as "Unclassified", "EV", and "ICE", respectively. The actual labels can be arbitrary and MAC-POSTS labels cars based on the order of these labels, e.g., 0 for "Unclassified", 1 for "EV", and 2 for "ICE". The total number of labels corresponds to parameter `num_of_car_labels` under section `[DTA]` in file `config.conf`. These labels are useful for fuel consumption and emission estimation considering the presence of electric vehicles. For example, when parameter `ev_label_car` under section `[DTA]` in file `config.conf` is set to 1, this indicates 30% of cars departing from this origin are considered as electric vehicles.

When parameter `num_of_car_labels` is set to 0, this file will be ignored.

```
#Origin_ID Unclassified EV ICE
1 0.1 0.3 0.6
```

6.1.14 MNM_origin_label_truck

This file is also optional and is used to input ratios of different types of trucks for each origin. See the following for an example. in which 10% and 90% of trucks departing from origin 1 are labeled as "EV" and "ICE", respectively. The actual labels can be arbitrary and MAC-POSTS labels trucks based on the order of these labels, e.g., 0 for "EV", and 1 for "ICE". The total number of labels corresponds to parameter `num_of_truck_labels` under section `[DTA]` in file `config.conf`. These labels are useful for fuel consumption and emission estimation considering the presence of electric vehicles. For example, when parameter `ev_label_truck` under section `[DTA]` in file `config.conf` is set to 1, this indicates 10% of trucks departing from this origin are considered as electric vehicles.

When parameter `num_of_truck_labels` is set to 0, this file will be ignored.

```
#Origin_ID EV ICE
1 0.1 0.9
```

6.2 Additional input files for DODE

DODE is to estimate the demand to calibrate the model based on real-world observations. So the value in file `MNM_input_demand` can be arbitrary. In addition to those files described above, MAC-POSTS can take the following data as real-world observations to carry out the DODE.

6.2.1 Traffic count

Traffic count data represents the vehicle counts passing by a certain location, and it is usually collected by loop detectors, tubes, or manual counting.

Suppose there are a total of M links ($M \leq$ `num_of_link`) that have time-dependent count records of length T ($T =$ `max_interval`), then we can use the following matrix to express the traffic count data

$$\begin{bmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,T} \\ c_{2,1} & c_{2,2} & \dots & c_{2,T} \\ \vdots & \vdots & \ddots & \vdots \\ c_{M,1} & c_{M,2} & \dots & c_{M,T} \end{bmatrix} \tag{2}$$

where the element $c_{m,t}$ represents the count data of link m at time interval t. To input this data to MAC-POSTS, the matrix needs to be flattened in column-major order to

a 1D NumPy array of length $M \times T$, which corresponds to `flatten(order='F')` operation in NumPy[5].

$$\begin{bmatrix} c_{1,1} & \dots & c_{M,1} & c_{1,2} & \dots & c_{M,2} & \dots & c_{1,T} & \dots & c_{M,T} \end{bmatrix} \tag{3}$$

For multiclass DODE, two such arrays for car and truck counts should be prepared.

6.2.2 Travel speed/time

MAC-POSTS can take observed time-dependent travel time of links as input (travel speed can be converted to travel time by dividing the link length by travel speed). The input format is similar to that for traffic count, namely, a flattened NumPy array in column-major order. For multiclass DODE, two such arrays for car and truck travel times should be prepared.

6.2.3 Vehicle registration data

In addition to the traditional traffic count and speed data, vehicle registration data can be also used to calibrate the dynamic network model. Since vehicle registration is mandatory for all vehicles, nearly all vehicles in the region are covered by this data. The data usually contains the vehicle identification number (VIN), zipcode of the driver's address, vehicle body type, odometer reading, and other auxiliary fields. As such, the vehicle registration data could serve as a high-quality vehicle census with regularly updated information and extensive coverage of the population, and could greatly aid the regional system modeling work. Integrating this data enables MAC-POSTS to estimate more fine-grained vehicle demands for further different research needs (e.g., more accurate emission estimation based on the vehicle's fuel types).

The biggest concern about using vehicle registration data is privacy. Because the VIN is a unique identifier of a vehicle, misuse of the data set may lead to catastrophic consequences. In processing the vehicle registration data of Pennsylvania, we employ the following steps to utilize the data:

- First, we decode all the VINs using the VIN Decoder of US National Highway Traffic Safety Administration (NHTSA, 2022).
- In the decoded information, we extract only the vehicle type, model year, primary fuel type, and secondary fuel type.
- Then we further aggregate all the vehicle types into a smaller set of vehicle classes used in this research.

[5] `https://numpy.org/doc/stable/reference/generated/numpy.matrix.flatten.html`

- Finally, we calculate the statistics for vehicles of different vehicle classes, age groups, and fuel types in each zip code zone in the study area. The results are the data used in further analysis.

With all those procedures of redaction and aggregation, the processed data should contain no sensitive information, but would still provide valuable information for transportation system modeling.

It is worth noting that different countries or regions may have different requirements on vehicle registration renewal, and the information contained may be different. However, in this study, we only rely on the most fundamental fields, namely VIN and zip code, in the data set, so the process should be easily reproducible in other regions.

We can rely on these two fields to calculate the number of vehicles of different types for each zip code zone and add this as an additional term in the objective function in the DODE task. More details can be found in Section 6.2.4.

We provide one way to use MAC-POSTS with the vehicle registration data. The vehicle registration data needs to be processed as a `csv` file. See the following for an example.

```
zipc,origin_ID,car,truck
18914,201531,10289.0,1916.0
18930,201502,1210.0,735.0
18938,201055,8068.0,1146.0
18940,"201571,201572",16193.0,2078.0
18944,201522,11251.0,3543.0
18947,201541,3050.0,1201.0
18951,201501,15439.0,5427.0
...
```

The first column represents the zip code. The second column is the origin IDs having the same zip code. Note multiple origin IDs may share the same one zip code. The third and fourth columns are the number of cars and trucks registered under this zip code, respectively.

6.2.4 Multiclass DODE with vehicle registration data

This section provides interested users with the mathematical formulation for the multiclass DODE with vehicle registration data implemented in MAC-POSTS, which is an extension to the original multiclass DODE proposed in Ma et al. (2020).

The multiclass DODE with vehicle registration data is formulated as an optimization problem below:

$$\min_{\{\mathbf{q}_i\}_{i\in I}} \quad w_1 \sum_{b\in B}\left(y_b - \sum_{i\in I}\sum_{a\in A}\sum_{\ddot{h}\in H} L_{ai}^{b\ddot{h}}\left(\sum_{od\in O}\sum_{r\in R_{rs}}\sum_{\dot{h}\in H} K_{odi}^{ra\dot{h}} p_{odi}^{r\dot{h}} q_{odi}^{\dot{h}}\right)\right)^2 \tag{4}$$

$$+ w_2 \sum_{e\in E}\left(u_e - \sum_{i\in I}\sum_{a\in A}\sum_{\ddot{h}\in H} M_{ai}^{e\ddot{h}} t_{ai}^{\ddot{h}}\right)^2 \tag{5}$$

$$+ w_3 \sum_{i\in I} d\left(\mathbf{v}_i, \mathbf{q}_i\right), \tag{6}$$

$$s.t. \quad \left\{\mathbf{t}_i, \mathbf{c}_i, \mathbf{K}_i\right\}_{i\in I} = \Lambda\left(\left\{\mathbf{f}_i\right\}_{i\in I}\right), \tag{7}$$

$$\mathbf{f}_i = \mathbf{p}_i \mathbf{q}_i, \forall i \in I, \tag{8}$$

$$\mathbf{x}_i = \mathbf{K}_i \mathbf{f}_i, \forall i \in I, \tag{9}$$

$$\mathbf{p}_i = \Phi_i\left(\left\{\mathbf{c}_i\right\}_{i\in I}, \left\{\mathbf{t}_i\right\}_{i\in I}\right), \forall i \in I, \tag{10}$$

$$\mathbf{q}_i \succeq \mathbf{0}, \forall i \in I, \tag{11}$$

where w_1, w_2, w_3 are weights for the three loss terms, $\Lambda(\cdot)$ is the DNL model, $\Phi_i(\cdot)$ is the routing model for vehicle class i, and $d(\cdot,\cdot)$ calculates the discrepancies between total numbers of vehicles of zones and the origin-destination travel demands. Explanations of other notations can be found in Table 1.

Vehicles are assumed registered in residential areas. For the morning traffic peak when the origin nodes are assumed to be residential areas, we have

$$d(\mathbf{v}_i, \mathbf{q}_i) = \sum_{z\in Z}\left(v_i^z - \sum_{o,d,\dot{h}} \mathbb{1}(o \in z) q_{odi}^{\dot{h}}\right)^2, \tag{12}$$

where $\mathbb{1}(o \in z)$ returns 1 when an origin node o is spatially within the zone z and 0 otherwise. Similarly, for the afternoon peak when the destination nodes are mostly residential areas, we instead have

$$d(\mathbf{v}_i, \mathbf{q}_i) = \sum_{z\in Z}\left(v_i^z - \sum_{o,d,\dot{h}} \mathbb{1}(d \in z) q_{odi}^{\dot{h}}\right)^2. \tag{13}$$

Table 1: List of notations.

Notation	Meaning
A	The set of all links.
B	The set of all links with vehicle count observations.

E	The set of all links with travel time observations.
I	The set of all vehicle classes.
T	The set of all time intervals.
Z	The set of all zones.
Ω	The set of all OD pairs.
R_{od}	The set of all paths between an OD pair $od \in \Omega$.
$\dot{h}$	The index of a specific time interval when vehicles depart from the origin.
$\ddot{h}$	The index of a specific time interval when vehicles arrive at the end of a link.
$q_{odi}^{\dot{h}}$	The travel demand from origin o to destination d (with $od \in \Omega$) in time interval $\dot{h}$ of vehicle class $i \in I$.
$\mathbf{q}_i$	The travel demands of vehicle class $i \in I$ for all OD pairs and time intervals. This is a packed, compact version of all $q_{odi}^{\dot{h}}$ for a specific vehicle class $i \in I$.
$\mathbf{v}_i$	Observed zone travel demands of vehicle class $i \in I$.
$\mathbf{x}_i$	Link traffic counts of vehicle class $i \in I$.
y_b	Observed link traffic count on link $b \in B$.
$\mathbf{y}$	Observed link traffic counts for all links in B.
$\mathbf{f}_i$	Path flows of vehicle class $i \in I$.
$t_{ai}^{\ddot{h}}$	Travel time of link $a \in A$ for vehicles of class $i \in I$ arriving at the end of the link at time interval $\ddot{h} \in H$.
$\mathbf{t}_i$	Link travel times of vehicle class $i \in I$.
$\mathbf{c}_i$	Link travel costs of vehicle class $i \in I$.
u_e	Observed link travel time of link $e \in E$.
$\mathbf{u}$	Observed link travel times of all links in E.

$p_{odi}^{k\dot{h}}$	Probability of vehicles of class $i \in I$ that depart from o and head to d (with $od \in \Omega$) in time interval $\dot{h} \in H$ choosing path $k \in R_{od}$.
$\mathbf{p}_i$	Route choice probabilities of vehicle class $i \in I$.
$K_{odi}^{ra\dot{h}}$	The portion of the flow of vehicle class $i \in I$ on path $r \in R_{od}$ departing from origin o in time interval $\dot{h} \in H$ which arrives at link $a \in A$ in time interval $\ddot{h} \in H$ among all paths between $od \in \Omega$. This term will be referred to as Dynamic Assignment Ratio (DAR).
$\mathbf{K}_i$	DAR matrix of vehicle class $i \in I$.
$L_{ai}^{b\ddot{h}}$	A binary number to indicate that link $a \in A$ is observed as $b \in B$ for vehicle class $i \in I$ during time interval $\ddot{h} \in H$.
$\mathbf{L}_i$	Incidence matrix for observed links and all links of vehicle class $i \in I$. This is an aggregated, packed representation of all $L_{ai}^{b\ddot{h}}$ above.
$M_{ai}^{e\ddot{h}}$	Weight of travel time $t_{ai}^{\ddot{h}}$ with $a \in A, i \in I, \ddot{h} \in H$ for the link with travel time observation $e \in E$.
$\mathbf{M}_i$	Travel time weight matrix of vehicle class $i \in I$.

7 Python APIs

Since MAC-POSTS works as a Python package, users will oftentimes use the Python APIs to solve different dynamic network modeling problems. All Python APIs are defined in `macposts/macposts/_macposts_ext.cpp`. Users can refer to these API names in `PYBIND11_MODULE`, which locates near the end of `macposts/macposts/_macposts_ext.cpp` and may look like the following code snippet:

```
...
...
PYBIND11_MODULE (_macposts_ext, m)
{
  m.def ("set_random_state", &MNM_Ults::set_random_state);

  py::class_<Tdsp> (m, "Tdsp")
    .def (py::init<> ())
    .def ("initialize", &Tdsp::initialize)
    .def ("read_td_cost_txt", &Tdsp::read_td_cost_txt)
    .def ("read_td_cost_py", &Tdsp::read_td_cost_py)
    .def ("build_tdsp_tree", &Tdsp::build_tdsp_tree)
    .def ("extract_tdsp", &Tdsp::extract_tdsp);

  py::class_<Dta> (m, "Dta")
    .def (py::init<> ())
    .def ("initialize", &Dta::initialize)
...
...
```

The Python APIs may be updated with the development of MAC-POSTS. But the basic steps to use these APIs in Python will remain the same and are as follows:

1. Import MAC-POSTS library in Python.
2. Find classes and functions to be used in `PYBIND11_MODULE`.
3. Take a look at the declarations of the found classes and functions (just search the name in file `_macposts_ext.cpp`).
4. Figure out the input and output.
5. Use the classes and functions accordingly in Python.

Particularly, the relationship between some common C++ and Python data types used in these APIs is listed in Table 2.

Table 2: Relationship between some common C++ and Python data types

C++	Python
`bool`	`bool`
`int`	`int`
`double`	`float`
`std::string`	`str`
`py::array_t<int>`	NumPy array with `dtype=int`
`py::array_t<double>`	NumPy array with `dtype=float`

MAC-POSTS currrently has the following main classes that can be invoked in Python:

- `Tdsp` for solving the time-dependent shortest path problem, which can be invoked using `Tdsp` or `tdsp_api` in Python.
- `Dta` for handling single-class DTA related problems, which can be invoked using `Dta`, `Dta_Api`, or `dta_api` in Python.
- `Mcdta` for handling multiclass DTA related problems, which can be invoked using `Mcdta`, `Mcdta_Api` or `mcdta_api` in Python.

We introduce some commonly used functions within these classes here.

7.1 `Tdsp`

7.1.1 `Tdsp::initialize`

```
int initialize(const std::string &folder, int max_interval,
               int num_rows_link_file, int num_rows_node_file);
```

It initializes a `Tdsp` object.

Input

- `folder`: the path to the directory containing the input network files
- `max_interval`: the maximum time interval considered for travel times and costs

- num_rows_link_file: the number of rows in the link file (usually a text or CSV file) containing information about the network links
- num_rows_node_file: the number of rows in the node file (also a text or CSV file) containing information about the network nodes.

7.1.2 Tdsp::read_td_cost_txt

```
int read_td_cost_txt(const std::string &folder,
                     const std::string &link_tt_file_name,
                     const std::string &node_tt_file_name,
                     const std::string &link_cost_file_name,
                     const std::string &node_cost_file_name);
```

It reads time-dependent costs for links and nodes from external plain text files in a specific folder.

Input

- folder: the path to the directory containing the input files
- link_tt_file_name: the name of the file containing the link travel times
- node_tt_file_name: the name of the file containing the node travel times
- link_cost_file_name: the name of the file containing the link travel costs
- node_cost_file_name: the name of the file containing the node travel costs

If the num_rows_node_file is not equal to -1, indicating that there are node costs in the input, the function also reads in the node travel times and costs from the node_tt_file_name and node_cost_file_name files. Otherwise, it only takes in link travel times and costs.

7.1.3 Tdsp::read_td_cost_py

```
int read_td_cost_py(py::array_t<double>td_link_tt_py,
                    py::array_t<double>td_link_cost_py,
                    py::array_t<double>td_node_tt_py,
                    py::array_t<double>td_node_cost_py);
```

It reads time-dependent costs for links and nodes from Numpy arrays.

Input

- `td_link_tt_py`: NumPy array containing the link travel times
- `td_link_cost_py`: NumPy array containing the link travel costs
- `td_node_tt_py`: NumPy array containing the node travel times
- `td_node_cost_py`: NumPy array containing the node travel costs

If the `num_rows_node_file` is not equal to `-1`, indicating that there are node costs in the input, the function also reads in the node travel times and costs from the `td_node_tt_py` and `td_node_cost_py` arrays. Otherwise, it only takes in link travel times and costs.

7.1.4 Tdsp::build_tdsp_tree

```
int Tdsp_Api::build_tdsp_tree(int dest_node_ID);
```

It builds a time-dependent shortest path tree (a data structure in `Tdsp`) given a destination node ID. Based on this, time-dependent shortest paths from any nodes departing at any timestamp leading to this destination node can be extracted afterward.

Input

- `dest_node_ID`: destination node ID

7.1.5 Tdsp::extract_tdsp

```
py::array_t<double> Tdsp_Api::extract_tdsp(int origin_node_ID,
                                           int timestamp);
```

It extracts the time-dependent shortest path from the specified origin node departing at the specified timestamp.

Input

- `origin_node_ID`: origin node ID
- `timestamp`: departing timestamp

Output

It returns a 2D NumPy array containing information about the time-dependent shortest path from the origin node to the destination node departing at the input timestamp, as well as other information about the path. This array has a dimension of number of nodes $\times$ 4. The first column records the nodes on the path. The second column records the links on the path, in which the last element is set to -1 as a placeholder

since number of links = number of nodes − 1. In the third column, the first element indicates the travel cost while other elements are set to -1 as placeholders. In the fourth column, the first element indicates the travel time while other elements are set to -1 as placeholders.

7.2 Dta

7.2.1 Dta::initialize

```
int initialize(const std::string &folder);
```

It initializes a `Dta` object from files in a specified folder, which are introduced in Section 6.1.

Input

- `folder`: the path to the directory containing the input files

7.2.2 Dta::register_links

```
int register_links(py::array_t<int> links);
```

It stores the links which will be deemed "observable". Some link-level simulation results (e.g., traffic count, travel time, and DAR) can only be extracted from these observable links.

Input

- `links`: 1D NumPy array containing the link IDs

7.2.3 Dta::register_paths

```
int register_paths(py::array_t<int> paths);
```

It stores the IDs of paths in the `path_table` and is mainly for DAR calculation.

Input

- `paths`: 1D NumPy array containing the path IDs

7.2.4 Dta::install_cc

```
int install_cc();
```

It installs both incoming and outgoing cumulative curves for the links stored by `register_paths`.

7.2.5 Dta::install_cc_tree

```
int install_cc_tree();
```

It installs tree-based cumulative curves for the links stored by `register_paths`, which are used for DAR calculation. This function is only needed in the DODE task.

7.2.6 Dta::run_whole

```
int run_whole(bool verbose=true);
```

It runs the single-class DNL model.

Input

- `verbose`: `true` means to output all the intermediate information, `false` means to suppresses the output.

7.2.7 Dta::run_due

```
int run_due(int max_iter, const std::string &folder,
            bool verbose=true, bool with_dtc=false,
            const std::string &method);
```

It runs the DUE model.

Input

- `max_iter`: maximum number of iterations.
- `folder`: path to the folder containing the input files.
- `verbose`: `true` means to output all the intermediate information, `false` means to suppresses the output.
- `with_dtc`: `true` means to consider departure time choice in DUE, `false` means not to.
- `method`: `MSA` means to use MSA to solve DUE, `GP` means to use GP to solve DUE.

7.2.8 Dta::run_dso

```
int run_dso(int max_iter, const std::string &folder,
            bool verbose=true, bool with_dtc=false,
            const std::string &method);
```

It runs the DSO model. Input is the same as those to `Dta::run_due`.

7.2.9 Dta::get_travel_stats

```
py::array_t<double> get_travel_stats();
```

It obtains network statistics of a single-class DNL model.

Output

1D NumPy array of the length of 4:

- the total number of released vehicles
- the total vehicle hours traveled (VHT) of released vehicles
- the number of enroute vehicles (i.e., vehicles still traveling)
- the number of finished vehicles

7.2.10 Dta::print_emission_stats

```
std::string print_emission_stats();
```

It prints out the estimation of fuel consumption and emission of a single-class DNL model, which is based on the MOVES Lite model (Liu & Frey, 2012; Zhou et al., 2015).

Output

A string including information about the fuel consumption (gallon), CO2 emission (g), HC emission (g), CO emission (g), NOX emission (g), vehicle miles traveled (VMT) for all released vehicles (mile), and VMT for released electric vehicles (mile).

7.2.11 Dta::build_link_cost_map

```
int build_link_cost_map(bool with_congestion_indicator=false);
```

It calculates time-dependent link travel costs and times after running a single-class DNL model. The results are stored internally and will be used by `get_path_tt` and in some tasks (e.g., DODE with travel times).

Input

- `with_congestion_indicator`: `true` means to also determine whether the congestion occurs at each timestamp, which is used in some tasks (e.g., PMC approximation, DSO, and DODE with travel times).

7.2.12 Dta::get_link_inflow

```
py::array_t<double> get_link_inflow(
                            py::array_t<int> start_intervals,
                            py::array_t<int> end_intervals);
```

It computes the time-dependent link inflows for given time intervals and links stored by `Dta::register_links`.

Input

- `start_intervals`: 1D NumPy array storing start times of intervals.
- `end_intervals`: 1D NumPy array storing end times of intervals.

Output

2D NumPy array of size of number of links stored by `Dta::register_links` $\times$ length of `start_intervals` storing the time-dependent link inflows. The order of links is the same as the order of links input to `Dta::register_links`.

`start_intervals` and `end_intervals` should have the same length. The timestamp in `start_intervals` and `end_intervals` is in the number of loading intervals (`unit_time` defined in file `config.conf`). For example, if `Dta::register_links` takes three links [2, 3, 4], `start_intervals` $=$ $[0, 180, 360, 540]$, and `end_intervals` $= [180, 360, 540, 720]$, this function will compute the link inflows for four intervals [0, 180], [180, 360], [360, 540], and [540, 720]. The output array's size is 3×4.

7.2.13 Dta::get_link_tt

```
py::array_t<double> get_link_tt(py::array_t<int>start_intervals,
                                bool return_inf = false);
```

It computes the time-dependent travel time for vehicles entering links stored by `Dta::register_links` at given timestamps.

Input

- `start_intervals`: 1D NumPy array storing timestamps.
- `return_inf`: `true` means to output very long travel time as infinity, `false` means to replace very long travel time with an upper bound (e.g., 20 times of free-flow travel time). This is mainly for smoothing the travel time gradient in DODE.

Output

2D NumPy array of size of number of links stored by `Dta::register_links` $\times$ length of `start_intervals` storing the time-dependent travel times. The order of links is the same as the order of links input to `Dta::register_links`.

The timestamp in `start_intervals` is in the number of loading intervals (`unit_time` defined in file `config.conf`). For example, if `Dta::register_links` takes three links [2, 3, 4] and `start_intervals` $= [0, 180, 360, 540]$, this function will compute the travel times for vehicles entering these links at four timestamps, i.e., 0, 180, 360, and 540. The output array's size is 3×4.

7.2.14 `Dta::get_path_tt`

```
py::array_t<double> get_path_tt(py::array_t<int>link_IDs,
                                py::array_t<int>start_intervals);
```

It computes the time-dependent travel time and cost for vehicles using the given path and departing at given timestamps. Must run `Dta::build_link_cost_map` before using this function.

Input

- `link_IDs`: 1D NumPy array containing the IDs of links of a path.
- `start_intervals`: 1D NumPy array storing timestamps.

Output

2D NumPy array of size of 2 $\times$ length of `start_intervals`, in which the first row and the second row store the time-dependent travel time and cost, respectively.

7.2.15 `Dta::save_dar_matrix`

```
int save_dar_matrix(py::array_t<int>link_start_intervals,
                    py::array_t<int>link_end_intervals,
                    py::array_t<double> f,
                    const std::string &file_name);
```

It computes the DAR matrix after running a single-class DNL model and saves the DAR matrix in a specified folder.

Input

- `link_start_intervals`: 1D NumPy array of length of `max_interval` specified in file `config.conf` containing start times of assignment intervals.

- `link_end_intervals`: 1D NumPy array of length of `max_interval` specified in file `config.conf` containing end times of assignment intervals.
- `f`: 1D NumPy array of length of number of paths stored by `Dta::register_paths` × `max_interval`.
- `file_name`: a string specifying the directory and the file name to store the DAR matrix.

Output

The full DAR matrix's size is (number of links × `max_interval`) × (number of paths × `max_interval`). It saves the DAR matrix as a sparse matrix using the triplet format (row number, column number, value of nonzero element). The file's location and name are specified in input argument `file_name`.

The DAR matrix associates the time-dependent link flows with the time-dependent path flows

$$\mathbf{x} = \mathbf{K}\,\mathbf{f} \tag{14}$$

where **K** is the DAR matrix, and **x** and **f** are the time-dependent link flows and the time-dependent path flows expressed in flattened vectors, respectively (see Section 6.2.1 for the flattening procedure).

The values of `link_start_intervals` and `link_end_intervals` usually rely on `unit_time`, `assign_frq`, and `max_interval` in file `config.conf`. The timestamp in `link_start_intervals` and `link_end_intervals` is in the number of loading intervals (`unit_time`). The length of `link_start_intervals` and `link_end_intervals` is equal to `max_interval`. And the increment between two successive timestamps in `link_start_intervals` and `link_end_intervals` is equal to `assign_frq`. For example, if `unit_time=5`, `assign_frq=180`, and `max_interval=4`, then `link_start_intervals` = [0, 180, 360, 540] and `link_end_intervals` = [180, 360, 540, 720].

7.3 Mcdta

7.3.1 Mcdta::initialize

```
int initialize(const std::string &folder);
```

It initializes a `Mcdta` object from files in a specified folder, which are introduced in Section 6.1.

Input

- `folder`: the path to the directory containing the input files

7.3.2 Mcdta::register_links

```
int register_links(py::array_t<int> links);
```

It stores the links which will be deemed "observable". Some link-level simulation results (e.g., traffic count, travel time, and DAR) can only be extracted from these observable links.

Input

- `links`: 1D NumPy array containing the link IDs

7.3.3 Mcdta::register_paths

```
int register_paths(py::array_t<int> paths);
```

It stores the IDs of paths in the `path_table` and is mainly for DAR calculation.

Input

- `paths`: 1D NumPy array containing the path IDs

7.3.4 Mcdta::install_cc

```
int install_cc();
```

It installs incoming and outgoing cumulative curves for the links stored by `register_paths`. Links have separate cumulative curves for cars and trucks.

7.3.5 Mcdta::install_cc_tree

```
int install_cc_tree();
```

It installs tree-based cumulative curves for the links stored by `register_paths`, which are used for DAR calculation. This function is only needed in the DODE task. Links have separate tree-based cumulative curves for cars and trucks.

7.3.6 Mcdta::run_whole

```
int run_whole(bool verbose=true);
```

It runs the multiclass DNL model.

Input

- `verbose`: `true` means to output all the intermediate information, `false` means to suppresses the output.

7.3.7 Mcdta::get_travel_stats

```
py::array_t<double> get_travel_stats();
```

It obtains network statistics of a multiclass DNL model.

Output

1D NumPy array of the length of 4:

- the total number of released cars
- the total number of released trucks
- the total vehicle hours traveled (VHT) of released cars
- the total vehicle hours traveled (VHT) of released trucks

7.3.8 Mcdta::print_emission_stats

```
std::string print_emission_stats();
```

It prints out the estimation of fuel consumption and emission of a multiclass DNL model.

Output

A string including separate information for cars and trucks about the fuel consumption (gallon), CO2 emission (g), HC emission (g), CO emission (g), NOX emission (g), vehicle miles traveled (VMT) for all released cars/trucks (mile), VMT for released electric cars/trucks (mile), VHT for all released cars/trucks (hour), and the total number of trips of cars/trucks.

7.3.9 Mcdta::build_link_cost_map

```
int build_link_cost_map(bool with_congestion_indicator=false);
```

It calculates time-dependent link travel costs and times for cars and trucks after running a multiclass DNL model. The results are stored internally and will be used by `get_path_tt_car` and `get_path_tt_truck` and in some tasks (e.g., DODE with travel times).

Input

- `with_congestion_indicator`: `true` means to also determine whether the congestion occurs at each timestamp, which is used in some tasks (e.g., PMC approximation, DSO, and DODE with travel times).

7.3.10 Mcdta::get_link_car_inflow

```
py::array_t<double> get_link_car_inflow(
                            py::array_t<int> start_intervals,
                            py::array_t<int> end_intervals);
```

It computes the time-dependent car inflows for given time intervals and links stored by `Mcdta::register_links`.

Input

- `start_intervals`: 1D NumPy array storing start times of intervals.
- `end_intervals`: 1D NumPy array storing end times of intervals.

Output

2D NumPy array of size of number of links stored by `Mcdta::register_links` $\times$ length of `start_intervals` storing the time-dependent car inflows. The order of links is the same as the order of links input to `Mcdta::register_links`.

`start_intervals` and `end_intervals` should have the same length. The timestamp in `start_intervals` and `end_intervals` is in the number of loading intervals (`unit_time` defined in file `config.conf`). For example, if `Mcdta::register_links` takes three links [2, 3, 4], `start_intervals` $=$ $[0, 180, 360, 540]$, and `end_intervals` $= [180, 360, 540, 720]$, this function will compute the car inflows for four intervals [0, 180], [180, 360], [360, 540], and [540, 720]. The output array's size is 3×4.

7.3.11 Mcdta::get_link_truck_inflow

```
py::array_t<double> get_link_truck_inflow(
                            py::array_t<int> start_intervals,
                            py::array_t<int> end_intervals);
```

It computes the time-dependent truck inflows for given time intervals and links stored by `Mcdta::register_links`.

Input

- `start_intervals`: 1D NumPy array storing start times of intervals.
- `end_intervals`: 1D NumPy array storing end times of intervals.

Output

2D NumPy array of size of number of links stored by `Mcdta::register_links` $\times$ length of `start_intervals` storing the time-dependent truck inflows. The order of links is the same as the order of links input to `Mcdta::register_links`.

`start_intervals` and `end_intervals` should have the same length. The timestamp in `start_intervals` and `end_intervals` is in the number of loading intervals (`unit_time` defined in file `config.conf`). For example, if `Mcdta::register_links` takes three links [2, 3, 4], `start_intervals` $= [0, 180, 360, 540]$, and `end_intervals` $= [180, 360, 540, 720]$, this function will compute the truck inflows for four intervals [0, 180], [180, 360], [360, 540], and [540, 720]. The output array's size is 3×4.

7.3.12 `Mcdta::get_car_link_tt`

```
py::array_t<double> get_car_link_tt(
                            py::array_t<int>start_intervals,
                            bool return_inf = false);
```

It computes the time-dependent travel time for cars entering links stored by `Mcdta::register_links` at given timestamps.

Input

- `start_intervals`: 1D NumPy array storing timestamps.
- `return_inf`: `true` means to output very long travel time as infinity, `false` means to replace very long travel time with an upper bound (e.g., 20 times of free-flow travel time). This is mainly for smoothing the travel time gradient in DODE.

Output

2D NumPy array of size of number of links stored by `Mcdta::register_links` $\times$ length of `start_intervals` storing the time-dependent travel times. The order of links is the same as the order of links input to `Mcdta::register_links`.

The timestamp in `start_intervals` is in the number of loading intervals (`unit_time` defined in file `config.conf`). For example, if `Mcdta::register_links` takes three links [2, 3, 4] and `start_intervals` $= [0, 180, 360, 540]$, this function will compute the travel times for cars entering these links at four timestamps, i.e., 0, 180, 360, and 540. The output array's size is 3×4.

7.3.13 Mcdta::get_truck_link_tt

```
py::array_t<double> get_truck_link_tt(
                            py::array_t<int>start_intervals,
                            bool return_inf = false);
```

It computes the time-dependent travel time for trucks entering links stored by `Mcdta::register_links` at given timestamps.

Input

- `start_intervals`: 1D NumPy array storing timestamps.
- `return_inf`: `true` means to output very long travel time as infinity, `false` means to replace very long travel time with an upper bound (e.g., 20 times of free-flow travel time). This is mainly for smoothing the travel time gradient in DODE.

Output

2D NumPy array of size of number of links stored by `Mcdta::register_links` $\times$ length of `start_intervals` storing the time-dependent travel times. The order of links is the same as the order of links input to `Mcdta::register_links`.

The timestamp in `start_intervals` is in the number of loading intervals (`unit_time` defined in file `config.conf`). For example, if `Mcdta::register_links` takes three links [2, 3, 4] and `start_intervals` $= [0, 180, 360, 540]$, this function will compute the travel times for trucks entering these links at four timestamps, i.e., 0, 180, 360, and 540. The output array's size is 3×4.

7.3.14 Mcdta::get_path_tt_car

```
py::array_t<double> get_path_tt_car(
                            py::array_t<int>link_IDs,
                            py::array_t<int>start_intervals);
```

It computes the time-dependent travel time and cost for cars using the given path and departing at given timestamps. Must run `Mcdta::build_link_cost_map` before using this function.

Input

- `link_IDs`: 1D NumPy array containing the IDs of links of a path.
- `start_intervals`: 1D NumPy array storing timestamps.

Output

2D NumPy array of size of 2 × length of `start_intervals`, in which the first row and the second row store the time-dependent travel time and cost, respectively.

7.3.15 Mcdta::get_path_tt_truck

```
py::array_t<double> get_path_tt_truck(
                            py::array_t<int>link_IDs,
                            py::array_t<int>start_intervals);
```

It computes the time-dependent travel time and cost for trucks using the given path and departing at given timestamps. Must run `Mcdta::build_link_cost_map` before using this function.

Input

- `link_IDs`: 1D NumPy array containing the IDs of links of a path.
- `start_intervals`: 1D NumPy array storing timestamps.

Output

2D NumPy array of size of 2 × length of `start_intervals`, in which the first row and the second row store the time-dependent travel time and cost, respectively.

7.3.16 Mcdta::save_car_dar_matrix

```
int save_car_dar_matrix(py::array_t<int>link_start_intervals,
                        py::array_t<int>link_end_intervals,
                        py::array_t<double> f,
                        const std::string &file_name);
```

It computes the DAR matrix after running a multiclass DNL model and saves the DAR matrix for cars in a specified folder. It is similar to `Dta::save_dar_matrix` (Section 7.2.15) except that the input argument `f` represents the path flows for cars.

7.3.17 Mcdta::save_truck_dar_matrix

```
int save_truck_dar_matrix(py::array_t<int>link_start_intervals,
                            py::array_t<int>link_end_intervals,
                            py::array_t<double> f,
                            const std::string &file_name);
```

It computes the DAR matrix after running a multiclass DNL model and saves the DAR matrix for trucks in a specified folder. It is similar to `Dta::save_dar_matrix` (Section 7.2.15) except that the input argument `f` represents the path flows for trucks.

8 Demonstrations

This section demonstrates the functionalities of MAC-POSTS in dealing with dynamic network modeling tasks using three real-world large-scale regional transportation networks.

8.1 Southwestern Pennsylvania Commission (SPC) regional network

8.1.1 Network description

The SPC network is shown in Figure 15. This network covers ten counties of the southwestern Pennsylvania region, with Pittsburgh city located in the center. There are around 2.57 million population and 7,112 square miles area in the network. All parameters for the SPC network are listed in the Table 3.

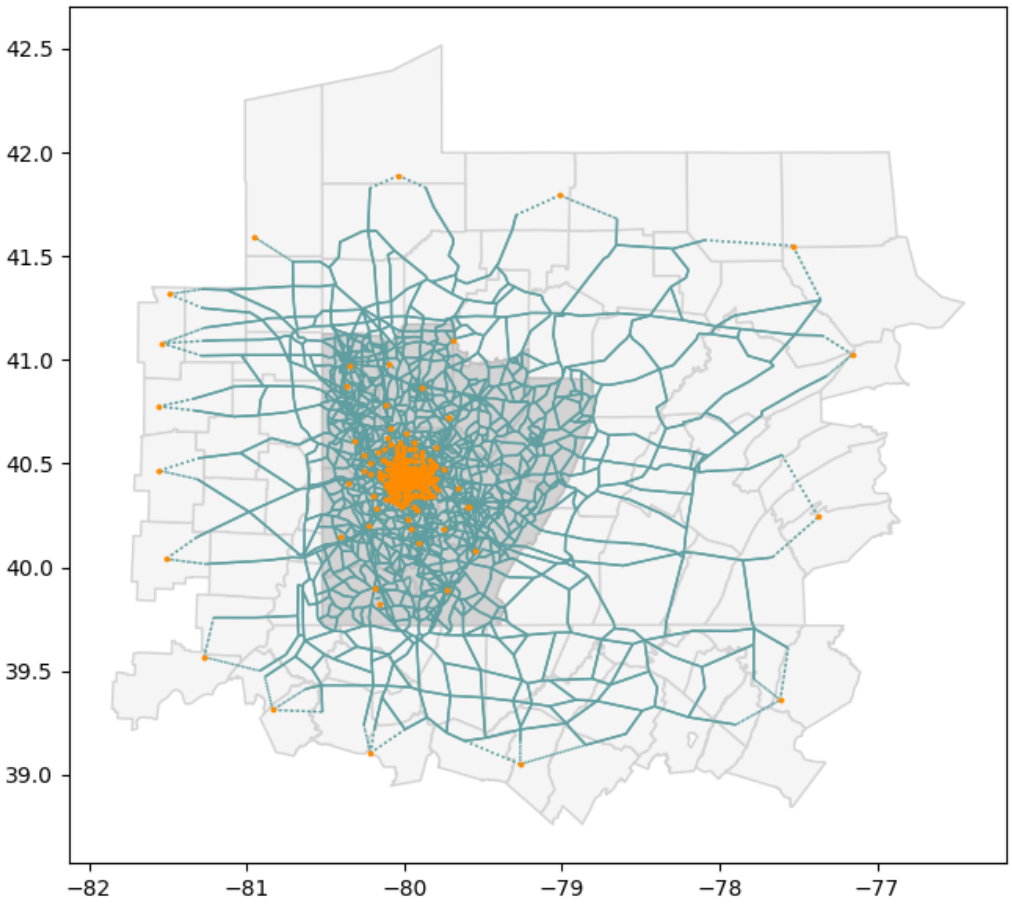

Figure 15: SPC network (the area in dark gray contains the ten counties of the SPC region, and the point markers show all the origin/destination nodes in the region).

Table 3: SPC network parameters

Name	Value
Analysis period	6:00 AM - 11:00 AM
DNL unit time	5 s
Length of assignment interval	15 min
Number of assignment intervals	20
Number of links	16,110
Number of nodes	6,297
Number of origins	283
Number of destinations	283
Number of O-D pairs	80,089

8.1.2 Traffic data

To calibrate the model, traffic data used includes traffic flow data, traffic speed data, and vehicle registration data, optionally with some additional data sets like the historical travel demands, etc.

The traffic flow data is from the Pennsylvania Department of Transportation (PennDOT). The data are collected hourly on selected locations on certain state-owned roadways. To ease the computation, we interpolate the hourly observations into traffic counts per 15 minutes. The car traffic volume counts and truck traffic volume counts are collected separately, where car traffic volume counts are measured for all passenger cars and passenger trucks, and truck traffic volume counts include all kinds of trucks (e.g., light commercial trucks, short-haul trucks, and long-haul trucks) at the measured location. There are 608 locations in total that have valid car and truck volume counts.

The traffic speed data comes from the Federal Highway Administration. In the data set, traffic speed values were collected every five minutes on some highway segments. Similarly, we aggregate the data into 15-minute intervals. In total, there are 945 locations with valid car and truck travel time observations.

For vehicle registration data, we use the 2017 vehicle registration records from PennDOT, with the necessary pre-processing and obfuscation steps described in Section 6.2.3.

8.1.3 DODE results

The DODE results are depicted in Figure 16. We use the coefficient of determination r^2 to measure the goodness of fit of our model. The r^2 scores for estimates of traffic counts are 0.32 and 0.70 for cars and trucks, respectively, and for zone travel demands those scores are 0.34 for cars and 0.89 for trucks.

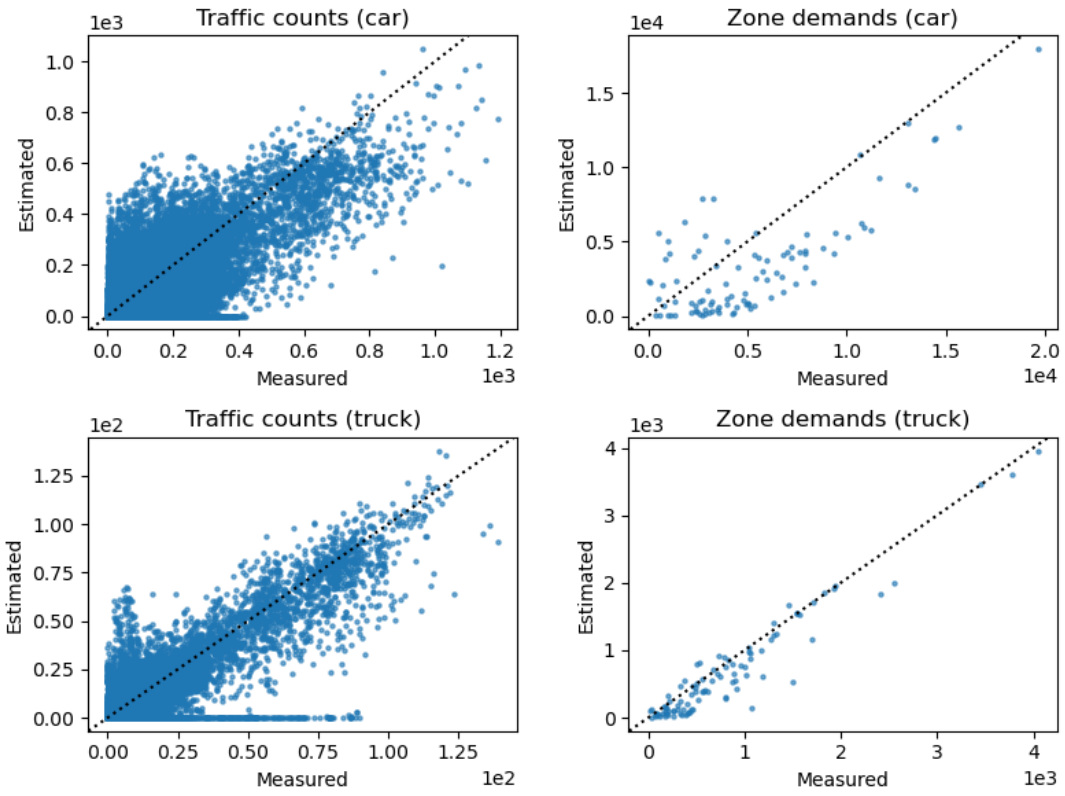

Figure 16: Comparison of estimated and observed link traffic counts and zone demands for the SPC network.

It can be seen that the estimates for cars are worse than those for trucks in this example. Inspection of the learning curves shows that the worse performance for cars is likely due to the discrepancy between the link-level traffic counts and zone-level vehicle registration records. Unlike trucks, cars tend to be used in a wide variety of ways. Therefore, simply aggregated values of registered passenger cars and passenger trucks may not be good proxies for travel demands of cars. It might be helpful to process and use the vehicle registration data in a fine-grained way, but that also brings the concern of revealing private information about drivers. How to better utilize vehicle registration data for cars remains an important direction of future work.

8.1.4 Scenario: impacts of increasing roadway capacities on emissions

We found that on certain short roads near the downtown Pittsburgh area the vehicles produce a large amount of emissions, which implies that those roads may not be able to handle the travel demands in the region and may often be congested. One straightforward way to tackle this issue is to simply increase the capacities of the congested links.

To begin with, we select the top 150 links from the network ordered by the amount of emissions and energy consumption. The selected links, shown in Figure 17, take less than 1% of all links, but generate more than 10% of total CO2 emissions.

Those links are either major highways on which there are a lot of demands, or in the downtown Pittsburgh area where the traffic conditions are highly complicated and roadways are often congested.

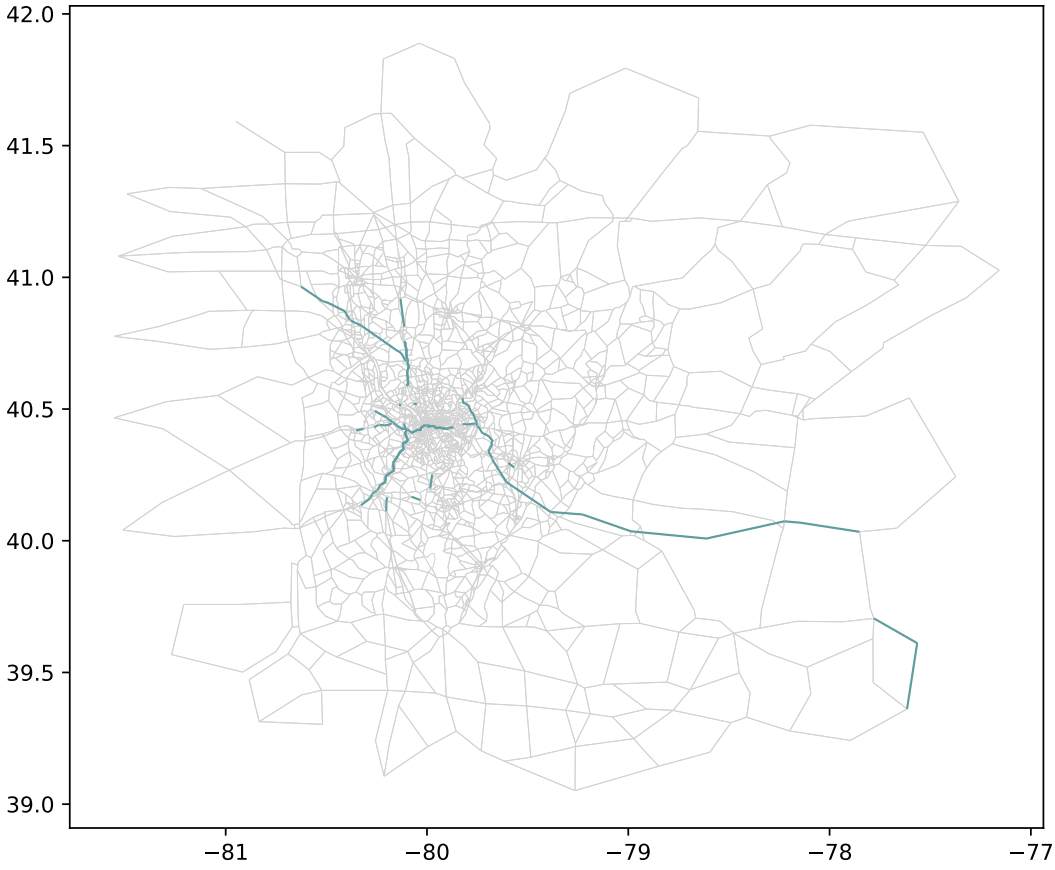

Figure 17: selected links to increase capacities.

We increase the capacities of those links by from 5% to 40%, and the total emissions and energy consumption of the whole network under different scenarios are shown in Figure 18.

Unexpectedly, the total emissions do not monotonically go down as the capacities of those links increase. A raise in link capacities does help reduce total emissions and energy consumption. However, a larger increase in link capacities does not guarantee a smaller amount of emissions and energy consumption.

The non-monotonicity is because of the complex interplay between the network and travelers' route choices. In the DTA model, we set 65% of vehicles to follow the adaptive routing strategy. That is, they will change their routes periodically to the currently shortest route to their destinations. Therefore, there exist two contradictory effects on the link travel times. Increasing the capacities of a link reduces the link travel time when the traffic volume stays the same, but those adaptive travelers will soon be aware of that and reroute towards those links, making them potentially more

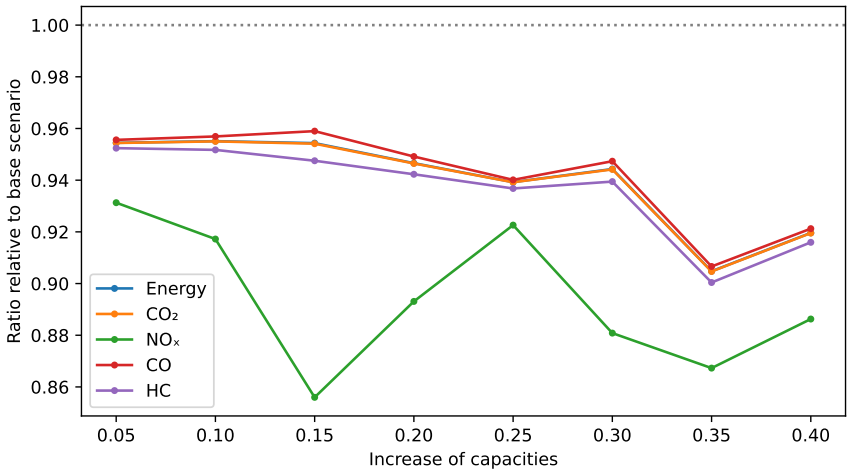

Figure 18: Ratios of energy consumption and emissions of the whole network with respect to the base scenario under different levels of capacity increase.

congested, which can be seen from Figure 19—increasing roadway capacities does not necessarily reduce average link travel times. Even worse, because the downtown area generally has a high traffic volume, all the roadway segments nearby could also be in the congested state. So it would be difficult for those adaptive travelers to reroute to other roadways, making the traffic conditions more complicated and unpredictable.

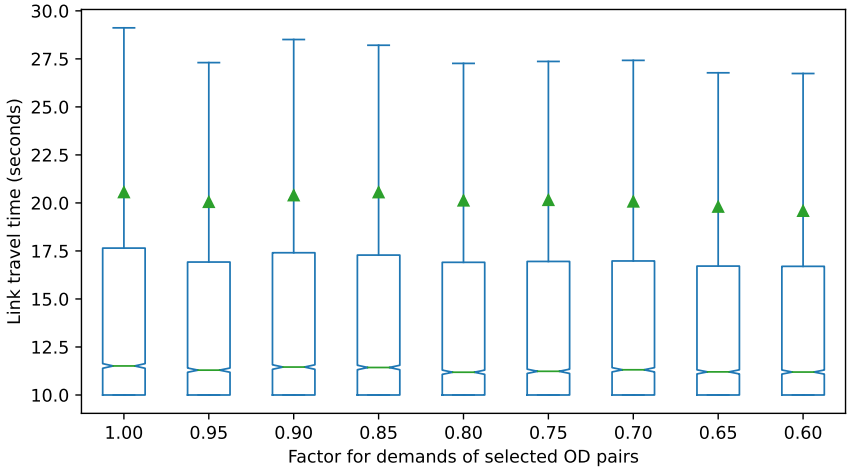

Figure 19: Distributions of link travel times under different capacities for the selected links in Figure 18 (the triangle markers are for mean travel times across all links).

8.2 Delaware Valley Regional Planning Commission (DVRPC) regional network

8.2.1 Network description

The DVRPC network covers nine counties in the Greater Philadelphia Region with Philadelphia city in the central area, as shown in Figure 20. Note that the original network data is trimmed so that there are no isolated nodes and links. The isolated nodes and links represent the parkways in the real world, and the absence of such nodes and links does not affect the network analysis. In addition, some neighboring links with small lengths and the same speed limit are further consolidated, which can substantially reduce the network size. More importantly, network consolidation has great potential in improving the accuracy of network analysis (S. Qian, 2016). All parameters for the DVPRC network are listed in the Table 4.

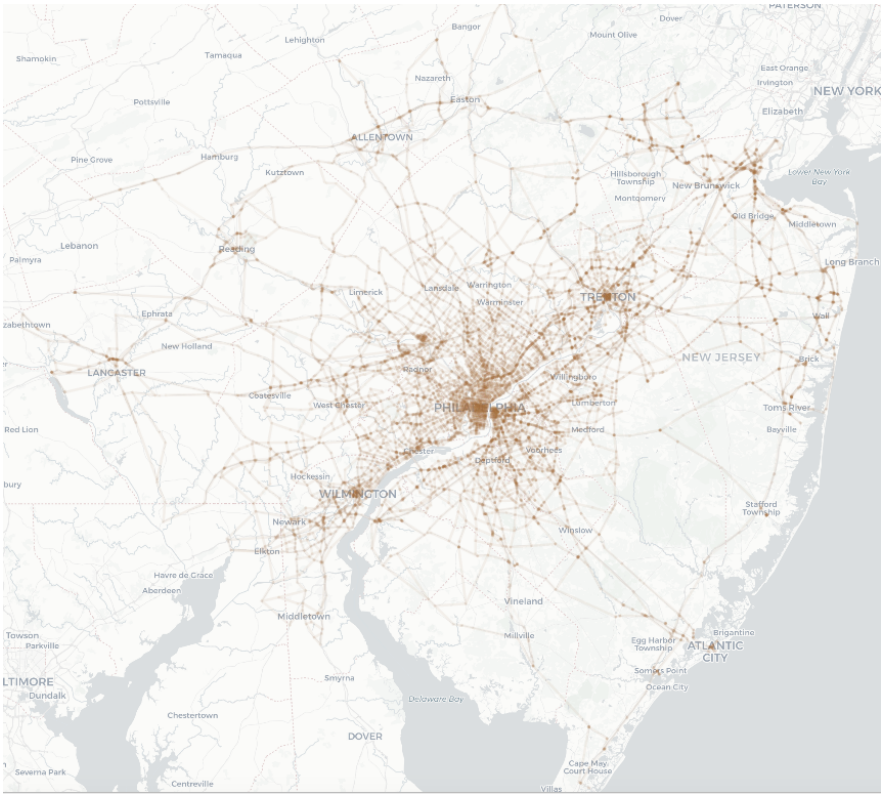

Figure 20: DVRPC network.

8.2.2 Traffic data

We also use traffic count, travel speed, and vehicle registration data to calibrate our model.

The traffic count data is provided by the DVRPC. The original count data contains 15-minute traffic volume counts for different vehicle types at selected locations in this region each day in 2019. The count data is carefully examined and cleaned

Table 4: DVRPC network parameters

Name	Value
Analysis period	5:00 AM - 10:00 AM
DNL unit time	5 s
Length of assignment interval	15 min
Number of assignment intervals	20
Number of links	41,586
Number of nodes	18,294
Number of origins	205
Number of destinations	205
Number of O-D pairs	41,820

and matched to the links in the transportation network. Two vehicle types, i.e., cars and trucks, are counted separately in the data, which represent smaller private or ride-hailing vehicles and larger freight trucks, respectively. In total, there are 1,920 locations with valid car and truck volumes, as shown in Figure 21.

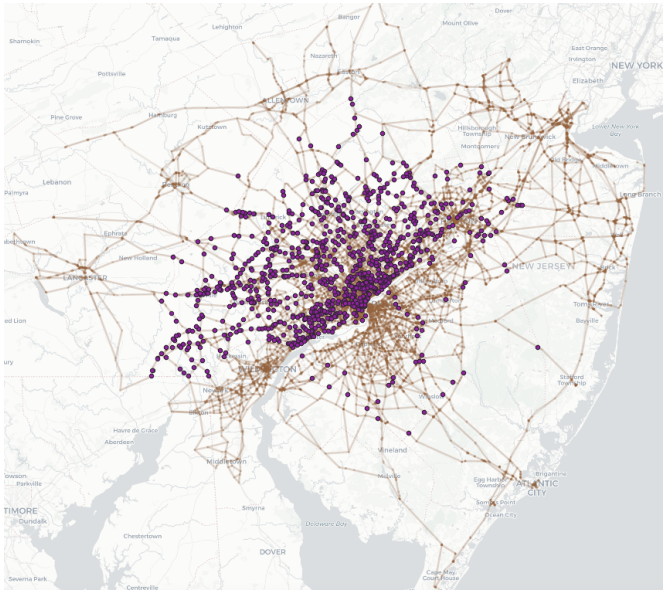

Figure 21: Traffic count locations in DVRPC network.

Traffic speed data is obtained from INRIX for the year 2019. Speeds of different vehicle types are measured separately, and hence both passenger car speeds and freight truck speeds are available. All the speed data is measured every 5 minutes of each day, and we average the data for different days in 2019 and aggregate the data to 15-minute intervals. There are a total of 3,420 locations with valid car and truck speed measurements, as shown in Figure 22.

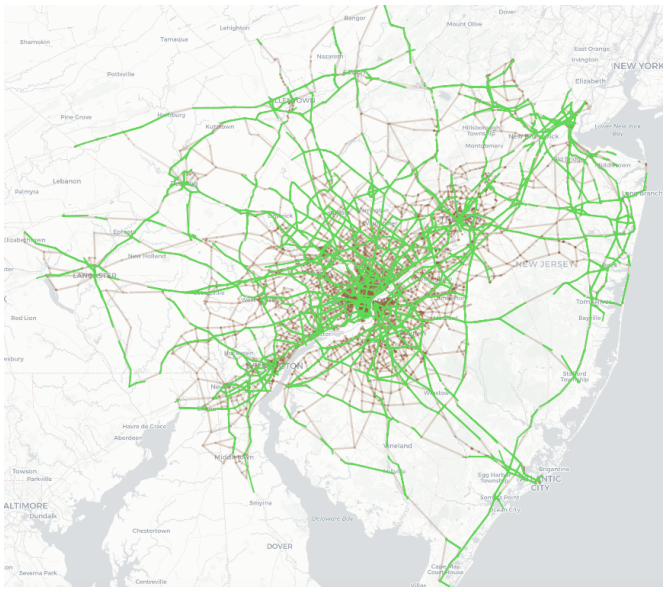

Figure 22: Links with speed records in DVRPC network.

Vehicle registration data is also from PennDOT.

8.2.3 DODE results

Figure 23 presents the comparison between simulated traffic volumes and observed traffic volumes, in which the y-axis is the simulated counts and the x-axis is the observed counts. The r^2 is 0.37, 0.42, and 0.97, for the car flow, truck flow, and origin vehicle demand, respectively.

Figure 23: Comparison of estimated and observed link traffic counts and zone demands for the DVRPC network (figures are for car counts, truck counts, and total zone demand from left to right).

Figure 24 depicts the comparison between simulated travel speeds and observed travel speeds, in which the y-axis is the simulated travel speed and the x-axis is the observed travel speed. The r^2 is 0.27 and 0.51, for the car travel speeds and truck travel speeds, respectively.

Note that calibration for large regional dynamic networks is known to be a challenging task and the result can be affected by different factors such as noise in the data and network simplification. The count estimation accuracy is expected to be improved with more DODE iterations. Overall, our model shows relatively good

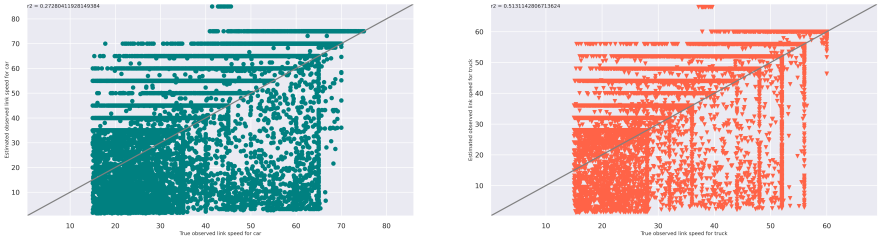

Figure 24: Comparison of estimated and observed link travel speeds for the DVRPC network (figures are for car speeds and truck speeds from left to right).

performance in capturing the trend of the observed data and this indicates that the proposed regional model can reflect the actual traffic dynamics in the whole DVRPC region to some extent.

8.2.4 Scenario: impacts of tolling a bridge

We consider setting a toll on Girard Point Bridge on the I-95 interstate in Philadelphia county, as shown in Figure 25. We set up three scenarios to investigate the impacts of the tolling on regional traffic. The baseline scenario represents the current no-toll situation. Scenarios 1 and 2 assume a toll of $1.00 and a toll of $2.00, respectively, for both directions for each vehicle using this bridge. In our simulation, the 65% of travelers will adapt to this tolling and may change their routes while the remaining 35% travelers will still stick to the pre-determined routes.

The aggregated traffic metrics within the DVRPC region, including travel time, delays, Vehicle Miles Traveled (VMT), and emissions are presented in Table 5. The metric changes from the baseline scenario to different tolling scenarios are shown in Table 6. In each table, the average (total) travel time indicates the average (total) time spent within the DVRPC region, and the average delay represents the average waiting time at each intersection in the DVRPC region.

As can be seen, with the introduction of tolling, travel time, fuel consumption, and emissions are increased. Since this is a large regional network, the relative changes in these metrics at the system level caused by only one bridge tolling may not be very significant, but in the absolute values, it still indicates some substantial impacts on fuel consumption and emissions. It is expected that these metrics changes for a more local area near the bridge can be more significant. And comparing Scenario 1 with Scenario 2, it can be found that the increase in the toll results in an increase in travel time, fuel consumption, and emissions, because facing a higher toll, travelers are more likely to take detours.

The total toll revenues in different scenarios can be estimated by multiplying the toll with the number of passing vehicles and are presented in Table 7.

We can also examine the primary vehicle detour routes due to the tolling on Gi-

Figure 25: Tolled bridge and its nearby bridges.

Table 5: General metrics in the AM peak hours.

		Total vehicles #	Total travel time hour	Average travel time hour	Average delay second	VMT mile	Fuel gallon	CO2 ton	CO ton	HC ton	NOX ton
Baseline	Car	1,604,373.5	2,052,337.1	1.3	11.4	52,975,339.9	1,908,235.5	16,958.5	63.8	44.6	59.5
	Truck	399,947.5	571,819.9	1.4	11.4	17,331,507.4	941,201.6	8,364.5	82.3	30.2	117.6
Scenario 1	Car	1,604,387.5	2,113,781.6	1.3	11.4	53,163,655.5	1,919,841.5	17,061.6	64.1	44.9	59.8
	Truck	400,143.5	585,932.5	1.5	11.4	17,419,338.4	947,672.6	8,422.0	82.8	30.4	118.4
Scenario 2	Car	1,604,330.5	2,157,446.0	1.3	12.0	53,890,143.4	1,947,461.6	17,307.1	64.9	45.8	60.6
	Truck	400,043.0	599,026.1	1.5	12.0	17,636,433.9	960,328.7	8,534.4	83.5	31.0	120.1

Table 6: Change of general metrics in the AM peak hours.

		Total vehicles #	**Total travel time hour**	**Average travel time hour**	**Average delay second**	**VMT mile**	**Fuel gallon**	**CO2 ton**	**CO ton**	**HC ton**	**NOX ton**
Baseline	Car	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Truck	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Scenario 1	Car	14.0	61,444.5	0.0	0.0	188,315.6	11,606.0	103.1	0.3	0.3	0.3
	Truck	196.0	14,112.6	0.1	0.0	87,831.0	6,471.0	57.5	0.5	0.2	0.8
Scenario 2	Car	-43.0	105,108.9	0.0	0.6	914,803.5	39,226.1	348.6	1.1	1.2	1.1
	Truck	95.5	27,206.2	0.1	0.6	304,926.5	19,127.1	169.9	1.2	0.8	2.5

Table 7: Total toll revenues in the AM peak hours ($).

	Scenario 1		Scenario 2	
	Cars	Trucks	Cars	Trucks
Westbound	12,774.5	4,005.5	23,231.0	6,910.0
Eastbound	10,381.0	3,333.5	12,465.0	4,095.0
Total revenue	30,494.5		46,701.0	

rard Point Bridge, in order to help public agencies understand how travelers' behavior changes.

As shown in Figure 25, there are two primary detour routes travelers can take to avoid the toll on Girard Point Bridge: George C. Platt Memorial Bridge and Passyunk Avenue Bridge.

The percentage changes in traffic flows of three scenarios are presented in Table 8. It can be found that the introduction of a toll on Girard Point Bridge indeed induces travelers to take detours and thus reduce its own traffic flows. More specifically, a toll of $1.00 in Scenario 1 can decrease the traffic flow in the baseline by almost 50% and a toll of $2.00 in Scenario 2 can decrease the traffic flow in the baseline by 60%. In comparison, the traffic flows on both George C. Platt Memorial Bridge and Passyunk Avenue Bridge are increased significantly. Due to its closer distance to Girard Point Bridge, George C. Platt Memorial Bridge experiences more traffic growth than Passyunk Avenue Bridge. It can also be found that a higher toll leads to more traffic reduction on Girard Point Bridge and more traffic increase on the other two bridges.

Table 8: Percentage change of traffic flows of three bridges in the AM peak hours.

		Baseline		Scenario 1		Scenario 2	
		Cars	Trucks	Cars	Trucks	Cars	Trucks
Girard Point	Westbound	0.0%	0.0%	-49.1%	-46.2%	-53.7%	-53.6%
Bridge	Eastbound	0.0%	0.0%	-33.3%	-23.1%	-60.0%	-52.7%
George C. Platt	Westbound	0.0%	0.0%	76.4%	77.0%	79.7%	79.0%
Memorial Bridge	Eastbound	0.0%	0.0%	24.3%	12.1%	58.6%	48.7%
Passyunk Avenue	Westbound	0.0%	0.0%	7.1%	33.7%	20.1%	35.9%
Bridge	Eastbound	0.0%	0.0%	14.8%	20.9%	27.7%	32.1%

8.3 Mid-Ohio Regional Planning Commission (MORPC) regional network

8.3.1 Network description

The MORPC network covers 15 counties in mid-Ohio with Columbus city in the central area, as shown in Figure 26. The original network data is also consolidated for better dynamic modeling purpose. All parameters for the MORPC network are listed in the Table 9.

Figure 26: MORPC network (red lines indicate links with speed records and red dots indicate locations with traffic count records).

Table 9: MORPC network parameters

Name	Value
Analysis period	5:00 AM - 9:00 AM
DNL unit time	5 s
Length of assignment interval	15 min
Number of assignment intervals	16
Number of links	26,217
Number of nodes	8,566
Number of origins	473
Number of destinations	473
Number of O-D pairs	138,560

8.3.2 Traffic data

We also use traffic count, travel speed, and vehicle registration data to calibrate our model.

Traffic count data is from the Ohio Department of Transportation (ODOT) where car traffic volume counts are measured for all passenger cars and truck traffic volume counts include all kinds of trucks at the measured location in 2019. There are a total of 883 auto links with valid car or truck count data.

Traffic speed data is obtained from INRIX for the year 2019. Speeds of different vehicle types are measured separately, and hence both passenger car speeds and freight truck speeds are available. All the speed data is measured every 5 minutes of each day, and we average the data for different days in 2019 and aggregate the data to 15-minute intervals.

Vehicle registration data is provided by BlastPoint[6], which does not have sensitive information.

8.3.3 DODE results

Figure 27 presents the comparison between simulated traffic volumes and observed traffic volumes, in which the y-axis is the simulated counts and the x-axis is the observed counts. The r^2 is 0.80, 0.82, and 0.99, for the car flow, truck flow, and origin vehicle demand, respectively.

Figure 28 depicts the comparison between simulated travel times and observed travel times, in which the y-axis is the simulated travel time and the x-axis is the observed travel time. The r^2 is 0.44 and 0.57, for the car travel times and truck travel times, respectively.

[6] `https://blastpoint.com/`

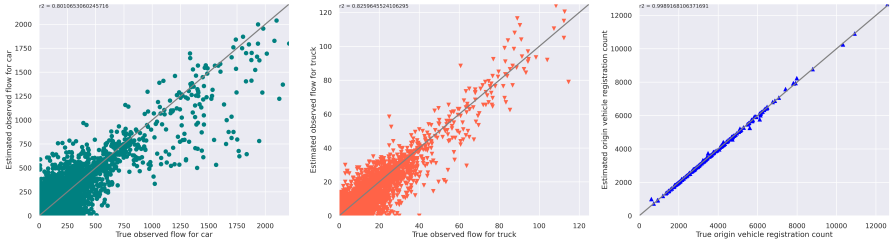

Figure 27: Comparison of estimated and observed link traffic counts and zone demands for the MORPC network (figures are for car counts, truck counts, and total zone demand from left to right).

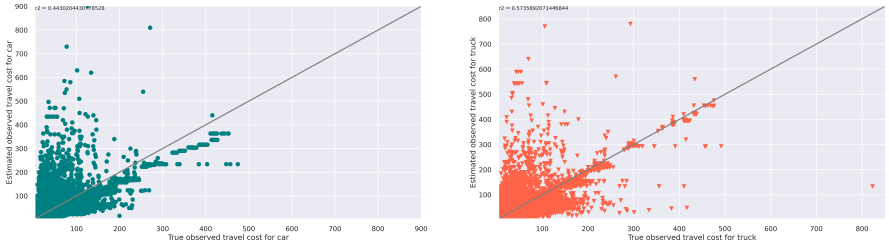

Figure 28: Comparison of estimated and observed link travel times for the MORPC network (figures are for car travel times and truck travel times from left to right).

8.3.4 Scenario: impacts of increasing penetration ratio of electric vehicles (EVs) on energy consumption

We investigate the impacts of increasing the penetration ratio of EVs on emissions. By adjusting the EV penetration ratio, we set up four scenarios. The base scenario is the status quo, in which different origins have different EV penetration ratios ranging from 0% to 6% based on the vehicle registration data. In scenarios 1, 2, and 3, origins are assumed to have uniform 5%, 10%, and 20% EV penetration ratios, respectively. We compare the energy consumption in different scenarios and the results are listed in Table 10. It can be seen that as expected, the total fuel consumption decreases with the increase in the EV penetration ratio.

Table 10: Energy consumption under different EV penetration ratio

Scenario	Non-EV fuel consumption (gallon)	EV VMT (mile)	EV electricity consumption (kWh)	EV equivalent fuel consumption (gallon)	Total equivalent fuel consumption (gallon)
Base	754,814.57	35,020.15	12,116.97	359.55	755,174.13
S1	730,937.90	606,077.06	209,702.66	6,222.63	737,160.53 (-2%)
S2	709,635.61	1,221,963.79	422,799.47	12,545.98	722,181.59 (-4%)
S3	662,225.13	2,416,791.53	836,209.87	24,813.35	687,038.48 (-9%)

Note: conversion from miles to electricity consumption is based on an average electricity consumed per mile for an EV = 0.346 kWh/mile and conversion from electricity consumption to fuel consumption is based on 33.7 kWh of electricity = 1 gallon of gas.

Acknowledgments

This project is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Vehicle Technology Office's award number DE-EE0008466. The contents of this report reflect the views of the authors, who are responsible for the facts and the accuracy of the information presented herein. The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

Chiu, Y.-c., Bottom, J., Mahut, M., Paz, A., Balakrishna, R., Waller, T., & Hicks, J. (2011). Dynamic traffic assignment: A primer. *Transportation Research Circular E-C153*, paper #09–3721. (ISBN: 0097-8515) doi: 10.1016/j.trd.2016.06.003

Jin, W.-L. (2017). A riemann solver for a system of hyperbolic conservation laws at a general road junction. *Transportation Research Part B: Methodological*, *98*, 21–41.

Liu, B., & Frey, H. C. (2012). Development and evaluation of a simplified version of moves for coupling with a traffic simulation model. In *Presented at transportation research board annual meeting* (Vol. 31, p. 32).

Ma, W., Pi, X., & Qian, Z. (2020). Estimating multi-class dynamic origin-destination demand through a forward-backward algorithm on computational graphs. *Transportation Research Part C: Emerging Technologies*, *102747*. (arXiv: 1903.04681)

NHTSA. (2022). *Vin decoder.* Retrieved from `https://vpic.nhtsa.dot.gov/decoder/`

Nie, Y. (2006). *A Variational Inequality Approach For Inferring Dynamic Origin-Destination Travel Demands* (Ph.D. Dissertation). University of California, Davis.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc.

Peeta, S., & Mahmassani, H. S. (1995). System optimal and user equilibrium time-dependent traffic assignment in congested networks. *Annals of Operations Research*, *60*(1), 81–113. doi: 10.1007/BF02031941

Pi, X., Ma, W., & Qian, Z. S. (2019). A general formulation for multi-modal dynamic traffic assignment considering multi-class vehicles, public transit and parking. *Transportation Research Part C: Emerging Technologies*, *104*(May), 369–389. (Publisher: Elsevier) doi: 10.1016/j.trc.2019.05.011

Qian, S. (2016, September). *Dynamic Network Analysis & Real-time Traffic Management for Philadelphia Metropolitan Area* (Final Report No. WO-004). Pittsburg, PA: Carnegie Mellon University. Retrieved from `http://weima171.com/docs/WO004_final.pdf`

Qian, S., Li, J., Li, X., Zhang, M., & Wang, H. (2017, May). Modeling heterogeneous traffic flow: A pragmatic approach. *Transportation Research Part B: Methodological*, *99*, 183–204. Retrieved 2021-01-12, from http://www.sciencedirect.com/science/article/pii/S0191261517300565 doi: 10.1016/j.trb.2017.01.011

Qian, Z. S., Shen, W., & Zhang, H. M. (2012). System-optimal dynamic traffic assignment with and without queue spillback: Its path-based formulation and solution via approximate path marginal cost. *Transportation Research Part B: Methodological*, *46*(7), 874–893. doi: 10.1016/j.trb.2012.02.008

Qian, Z. S., & Zhang, H. M. (2013). A Hybrid Route Choice Model for Dynamic Traffic Assignment. *Networks and Spatial Economics*, *13*(2), 183–203. (ISBN: 1106701291) doi: 10.1007/s11067-012-9177-z

Zhou, X., Tanvir, S., Lei, H., Taylor, J., Liu, B., Rouphail, N. M., & Frey, H. C. (2015). Integrating a simplified emission estimation model and mesoscopic dynamic traffic simulator to efficiently evaluate emission impacts of traffic management strategies. *Transportation Research Part D: Transport and Environment*, *37*, 123–136.