AVDTA

Michael W. Levin

December 31, 2016

Table of Contents

Chapter 1

Introduction

AVDTA is a research software for studying DTA and mesoscopic simulation models of autonomous vehicle technologies. AVDTA extends the typical kinematic wave theory mesoscopic models of simulation-based DTA by adding a number of revised models based on CAV technologies. These include the following:

- Multiclass cell transmission model where the fundamental diagram changes in space and time [11]
- Dynamic lane reversal [5, 10]
- Conflict region model of reservation-based intersection control [9, 13]
- Backpressure and P_0 policies for reservation-based intersection control
- Dynamic transit lanes

This documentation is divided into two parts. Part I explains the user interface and input data formats, and is intended for users who wish to use the existing software. The user interface contains two parts: a control interface that can modify all network, transit, and demand data and run DTA, and an editor that can visualize the network and modify individual nodes and links. For software development, Part II discusses the code structure and gives some examples of working with the code. Most of the code, including all classes relevant to traffic flow and simulation, also has Javadocs.

1.1 Contributors

Most software development was by Michael W. Levin. Sudesh Agrawal contributed to calibration and testing of the conflict region model and cell transmission model. Josiah P. Hanna and Guni Sharon contributed to the microtolling package.

1.2 Distribution

AVDTA is not available for commercial use, and may not be used or distributed without the permission of the authors.

Chapter 2

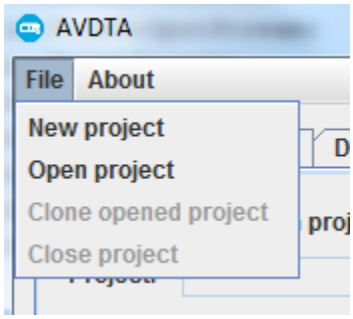
Project structure

2.1 Introduction

AVDTA is organized around *projects*. A project is best thought of as a specific scenario under study. For instance, a project might represent a certain demand scenario or network configuration for a city network or subnetwork. The AVDTA GUI contains methods to modify project options, such as link types or intersection controls. Data files can also be copied to and from Excel for easy modification.

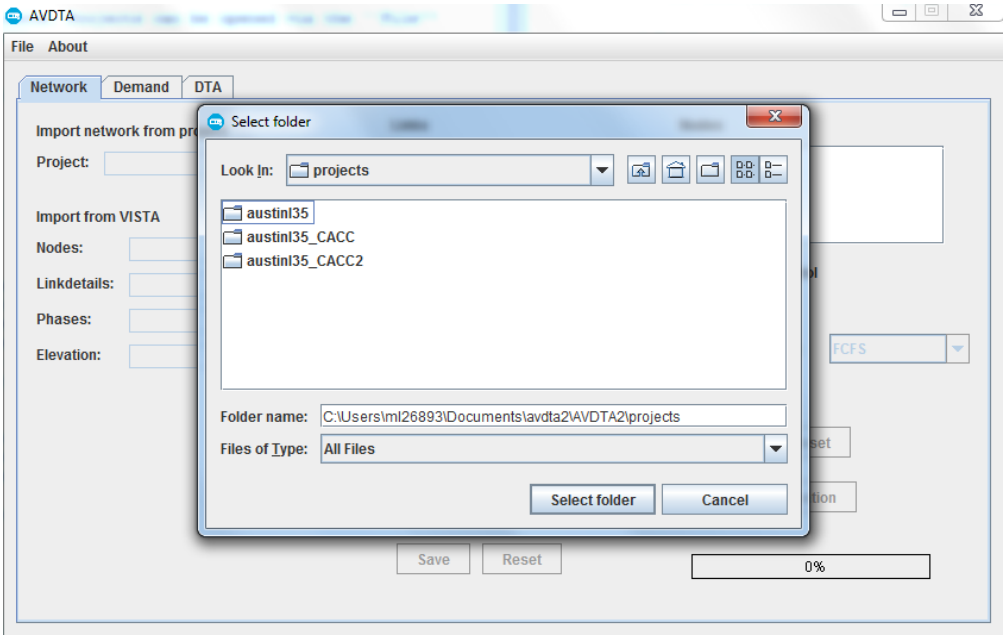
2.2 GUI

The AVDTA GUI is designed to interact with projects. Each instance of AVDTA can have a single project open at a time. Projects can be created and opened via the "File" menu.

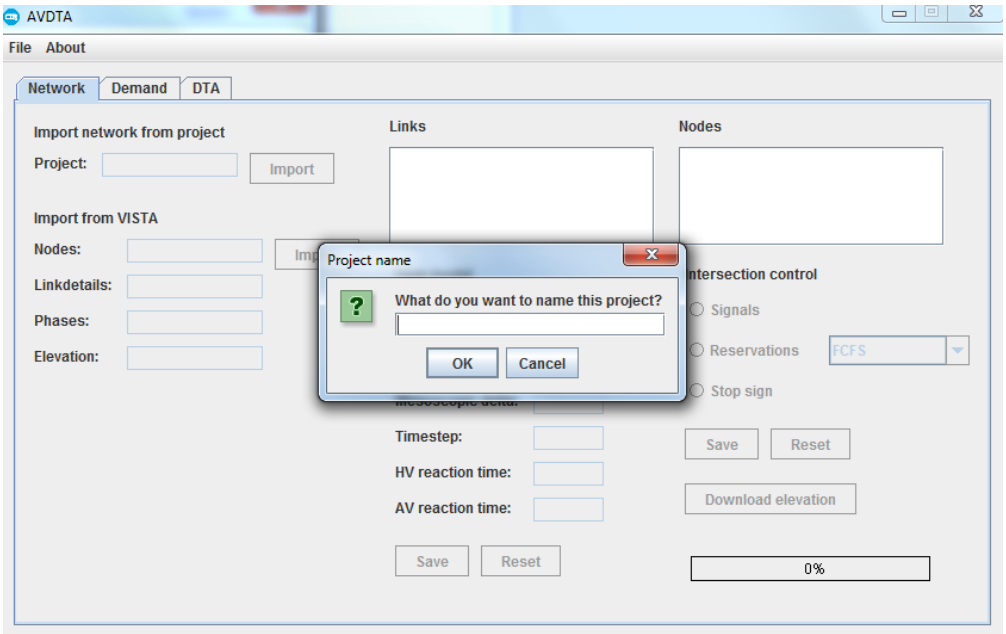


2.2.1 New project

To create a new project, you will first be asked to select the root folder. By default, this folder is `avdta/projects`, but it may be changed.

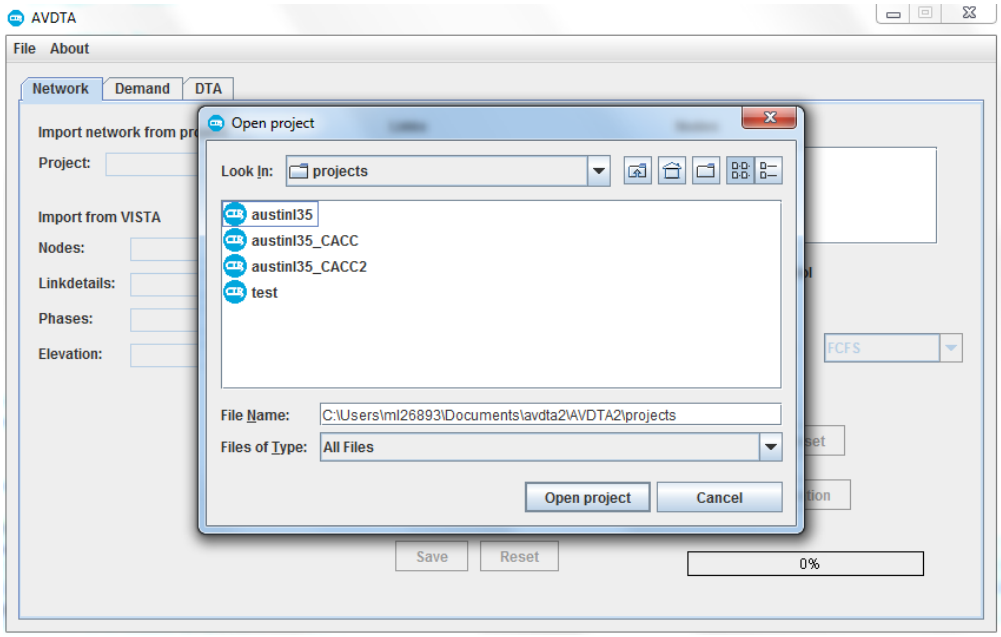


You will then be asked to enter the name of the new project. A project folder with the entered name will be created in the selected root folder. New projects are created with empty data files.



2.2.2 Open project

AVDTA project folders are shown with a special icon. Select a project folder and click "open project" to open it.



2.2.3 Clone opened project

If a project is open, the AVDTA GUI enables the option of creating a clone. Follow the instructions in Section 2.2.1 to choose the location and name of the clone. You can also create a copy of a project folder within the file system.

2.3 Files and folders

A project consists of a file folder containing several specific files and subfolders. The project folder, along with many of its files and subfolders, is automatically generated when a new project is created. However, many files will be empty, requiring the import of data from other projects or other sources. Note that removal or modification of these files outside of the AVDTA GUI could result in errors when loading the project. However, adding files or folders will not affect the project.

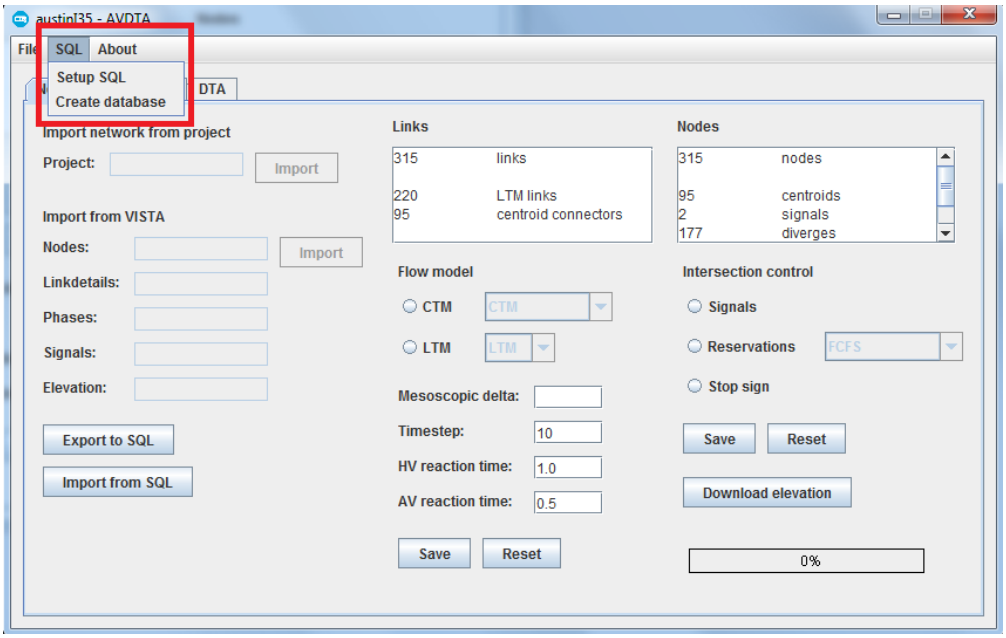
Projects contain several types of files. Text files (`.txt`) contain project inputs or outputs and are intended to be read or modified. However, if modifying these files, ensure that the format and units are correct. Text files are tab-delimited, and have a header indicating the data in each column. Text files may be copied to and from Excel. In addition, AVDTA contains SQL support. A separate SQL installation is required to use SQL.

Data files (`.dat`) are used to load the project and are not intended to be opened or modified. Similarly, files with other unknown extensions are not intended to be opened.

2.3.1 Working with SQL

AVDTA uses an internal file structure that does not require the use of SQL. The purpose of the internal file structure is to allow easy installation and modification of files through other programs (such as Excel). However, if SQL is installed, AVDTA can export data as SQL tables and import data from SQL tables, which allows the use of SQL to work with input data and results.

To use SQL in a project requires two initialization steps. First, set up the SQL connection. AVDTA requires the connection information (IP, port — default 3306) and the username and password of an user account with the permissions to create databases. To set up the SQL connection, select the "Setup SQL" menu item. Note that if SQL is not set up, interface options for SQL support will not be enabled



Next, the project must have an associated SQL database. To create a database, select the "Create database" option. The created database will have the same name as the project, except that all spaces will be replaced with underscores.

2.3.2 Files

2.3.2.1 `project.txt`

The `project.txt` file contains project properties used to load the project within AVDTA. The file consists of two columns: The file consists of two columns:

key	value

name This is the project name that is displayed when AVDTA loads a project. This does not have to correspond to the project folder name.

seed This is the random number generator seed. If two projects have matching seeds, actions involving random numbers performed in the same order should have identical outputs.

type This denotes the project type.

2.3.2.2 `options.txt`

The `options.txt` file contains project parameters that define the loading and simulation of the project. The file consists of two columns:

key	value

Keys are case insensitive. Values are a string that could represent multiple data types, such as integers, floating-point numbers, or booleans ("true" or "false"). The options file is automatically generated with default values when a project is created.

av-reaction-time This is used to determine the capacity and congested wave speed increase due to autonomous vehicles when using the multiclass CTM [11]. A typical value is 0.5 (s). Note that capacity and congested wave speed are scaled from the values in the `networks/links.txt` file.

hv-reaction-time This is used to determine the baseline capacity and congested wave speed when using the multiclass CTM [11]. A typical value is 1 (s). Note that capacity and congested wave speed are scaled from the values in the `networks/links.txt` file.

dynamic-lane-reversal If set to "true", DLR will be activated on DLR links. DLR is a specific type of CTM link, and the type of links can be set in the `networks/links.txt` file.

hvs-use-reservations If set to "true", HVs will not avoid reservation-controlled intersections in their route choices. Reservations will use the legacy early method for intelligent traffic management [2], adapted to DNL [11] for HVs. Otherwise, HVs will avoid reservations in their route choices, passing through reservations only if no other route is available.

simulation-duration This is the duration of the simulation, in seconds. The duration should be sufficiently longer than the demand departure times interval to allow all vehicles to exit the network.

simulation-mesoscopic-step This is the time step used in simulation. For CTM, a typical value is 6 (s). For LTM, a typical value is 10 (s).

ast-duration This the interval for averaging travel times for calculating shortest paths. A typical value is 900 (s).

2.3.2.3 `dta.dat`

The `dta.dat` file marks the project as a DTA project, and should not be modified.

2.3.2.4 `paths.dat`

The `paths.dat` file contains a list of all known paths in the network. AVDTA does not generate a complete list of paths, but all paths that are created during DTA are saved here. The file is used when loading and saving assignments.

2.3.3 Folders

`network` folder The `network` folder contains all network data files, discussed in Chapter 3.

`results` folder The `results` folder is used to store and organize results files.

`demand` folder The `demand` folder contains all demand data files, discussed in Chapter 5.

`assignments` folder The `assignments` folder stores previous assignments for the project. Each assignment consists of a subfolder, usually named by the time at which the assignment was created. Previous assignments can be loaded through the AVDTA GUI.

Each assignment subfolder contains several files. The `log.txt` file contains a log of the DTA run that created the assignment. It includes results from each iteration of DTA as well as summary statistics for the assignment. The `vehicles.dat` file contains summary statistics in a more readable form for AVDTA, and then a list of vehicles and the path they are assigned to. The path ids correspond to a path in the `paths.dat` file. Do not modify the `vehicles.dat` file.

Part I

User interface

Chapter 3

Traffic network

3.1 Introduction

A *network* in AVDTA defines the road structure through which vehicles can flow. It does not include vehicular demand or trips because these depend on the application. For instance, the vehicle trips for DTA models is different from vehicle trips for SAV models. The network in AVDTA compartmentalizes the road network, with vehicle trips handled separately depending on the application.

A network consists of *nodes* and (directed) *links* connecting nodes. Nodes are further divided into *intersections*, which connect roads, and *zones*, where demand and vehicles can enter or exit the network. AVDTA includes several options for intersection controls and link flow models, which are discussed in Sections 3.2 and 3.3, respectively. The discussion in this document pertains to using and modifying node and link data within AVDTA. References to the theoretical developments of flow models are included.

Network data files are contained in the `network` subfolder within a project folder. Network files include `nodes.txt`, `links.txt`, `link_coordinates.txt`. `signals.txt`, and `phases.txt` (which defines signal phases for nodes). All data files include a header as the first line.

3.2 Nodes

Nodes are either intersections or zones. Intersections represent physical intersections in the road network, and zones represent locations where vehicles or travelers enter and exit. It is typical for zones to double intersections. Each zone should be either a source or a sink.

Each node has sets of incoming and outgoing links. During simulation, nodes determine vehicle flow from incoming links to outgoing links. Incoming links define *sending flows* as a set of vehicles, and outgoing links define *receiving flows*. Vehicle flow is constrained by sending and receiving flows, and may be additionally constrained by intersection conflict constraints.

3.2.1 `nodes.txt`

The `nodes.txt` file contains the data to describe intersections and zones, and consists of the following:

id	type	longitude	latitude	elevation

Columns are tab-delimited, and each line corresponds to a new node. Additional columns will be ignored, and may be deleted if the GUI is used to modify the file.

id Each node must be given an unique id, which is used to identify the node in other data files. Ids should be positive, but do not need to be consecutive.

type The type specifies whether the node is a zone or an intersection. If an intersection, the type also identifies the intersection control. Type options are given below.

Category	Type	Description
Reservations	301	FCFS
	332	Pressure-based
	333	P_0
	334	Q^2 [13]
	335	DE^4 [13]
	321	PHASED
	322	WEIGHTED
Traffic signal	100	Standard traffic signal
Stop sign	200	4-way stop
Centroid	1000	Centroid

Note that merges and diverges will be automatically used. An intersection with only one incoming link will become a diverge, and an intersection with only one outgoing link will become a merge. This is because merges/diverges avoid the need for intersection controls, and also performed better than some reservation policies [12].

longitude/latitude Longitude and latitudes are specified in decimal format, with the sign indicating west/east and north/south, respectively. The longitude/latitude data is used only for display purposes. False data will not affect vehicle loading or simulation. Zones doubling an intersection typically have the same coordinates.

elevation Elevation data is used to determine link grade for estimating vehicle fuel consumptions [8]. False data will not affect vehicle loading or simulation. The GUI contains a method to download elevation data from Google Maps, given longitudes and latitudes.

3.2.2 `phases.txt`

The `phases.txt` file contains information about signal phases. Note that the order in which the phases appear is the order in which they will be created for the signal cycle. Each line in the `phases.txt` file is a separate phase, and the columns are

nodeid	type	sequence	time_red	time_yellow	time_green	num_moves	link_from
link_to							

nodeid The node id for which the phase is used.

type denotes the type of the phase. Currently 1 for all phases.

sequence the order in which the phase is used in the signal cycle. The sequence count starts at 1.

time_red, time_yellow Clearance interval times for this phase.

time_green Green time for this phase.

num_moves The number of protected turning movements, also the size of the "link_from" and "link_to" arrays.

link_from, link_to Protected turning movements for this phase. These consist of two equal-length arrays, one of incoming links and one of outgoing links. A valid turning movement is a pair of an incoming link and an outgoing link. Consequently, links may appear multiple times in the arrays to enumerate all protected turning movements.

3.2.3 `signals.txt`

The `signals.txt` file contains signal offsets for each node. Each node may have one entry. Duplicate entries will overwrite previous ones. A missing entry indicates an offset of 0. The `signals.txt` file columns are

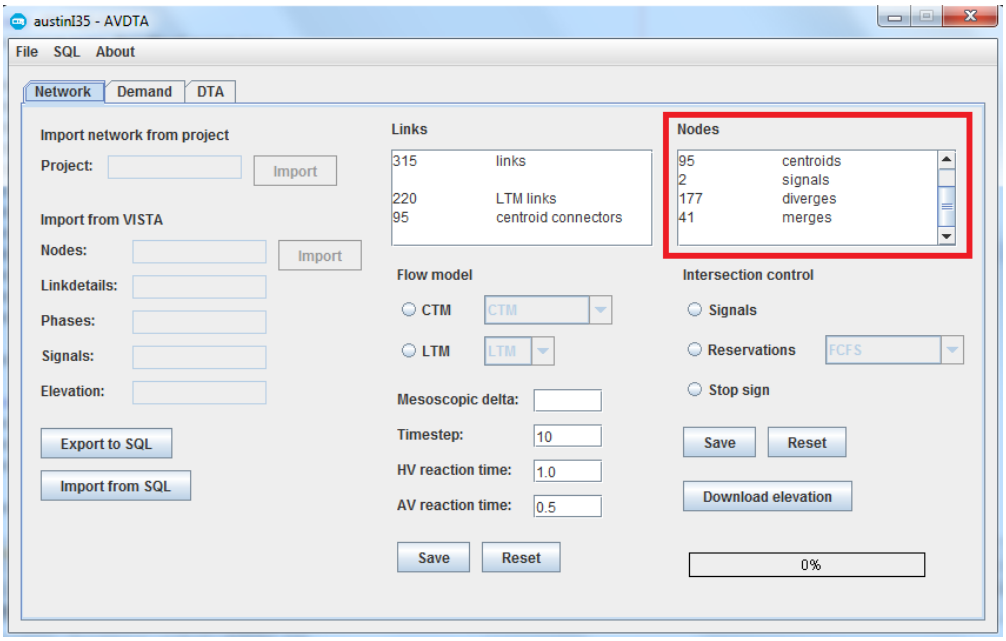
nodeid	offset

nodeid the node id for this signal

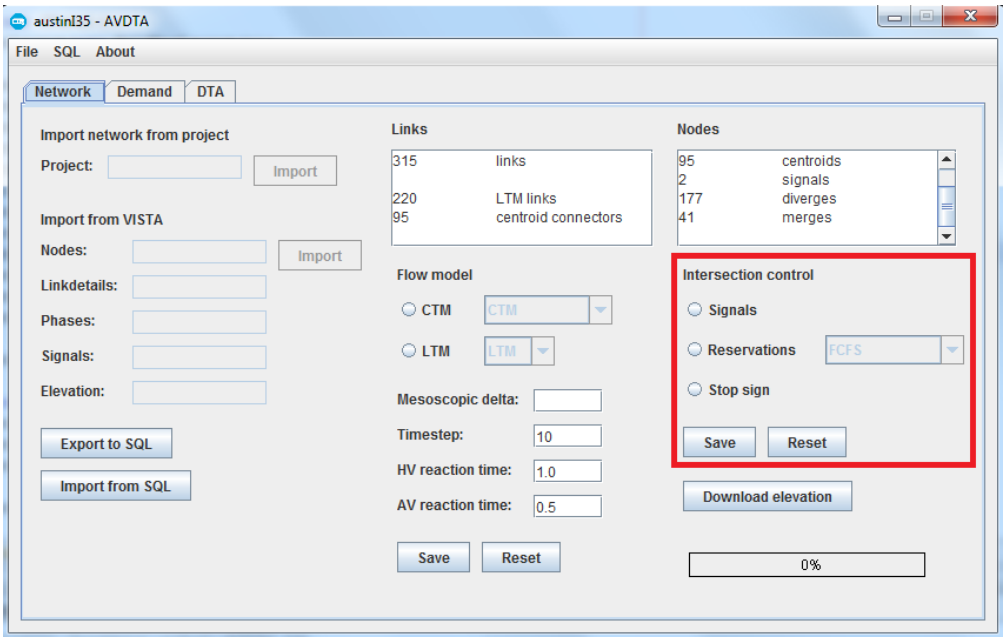
offset the time offset (in seconds). This can be a floating point number.

3.2.4 GUI panel

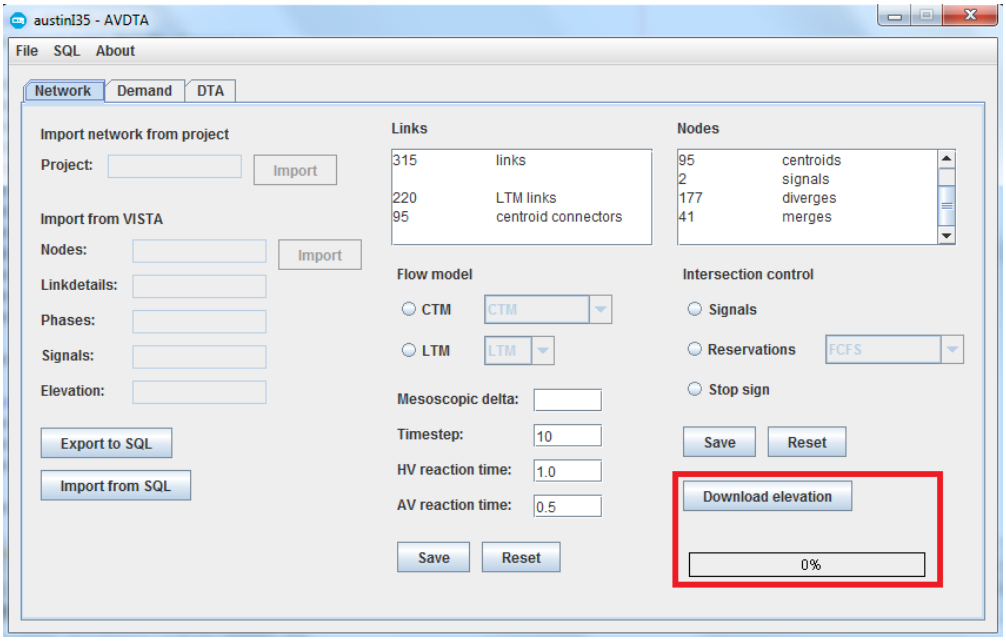
The AVDTA GUI nodes panel is on the right hand side of the "network" tab. The uppermost text area displays information about the network, including the number of total nodes, and nodes of each type.



The next panel controls intersections. Selecting an intersection control, then pressing the "save" button, will result in all intersections converted to the specified intersection control.



The final panel will attempt to download elevation data from the Google Maps API. This will only affect nodes for which the elevation is 0. This requires a connection to the internet. The download may take some time because a delay between requests is necessary to avoid being rejected as a DOS attack. In addition, the Google Maps API may limit the number of requests per day. If the process exits with an error, restart it later. It will save progress and continue from where it left off.



3.3 Links

3.3.1 `links.txt`

Each line in the `links.txt` file corresponds to a link in the network. The `links.txt` file has columns

id	type	source	dest	length	ffspd	w	capacity	num_lanes

id The id of the link. Each link must have an unique, positive id, but the ids do not have to be consecutive.

type This determines the flow model used for the link. The possible types are

Flow model	Type	Description
CTM	100	Multiclass CTM [11]
	102	CTM with DLR [10]
	103	CTM with dynamic transit lane
	104	CTM with dedicated transit lane
LTM	200	Standard LTM [17, 18]
	205	LTM with CACC
Centroid connector	1000	Link between a centroid and a node

Standard CTM is achieved through type 100 with HVs.

source The id of the source node.

dest The id of the destination node.

length The length of the link, in feet.

ffspd The free flow speed of the link, in miles per hour. Note that the free flow travel time is rounded up to the nearest time step.

w The congested wave speed of the link, in miles per hour. For LTM links, the free flow speed, capacity, and congested wave speed must be consistent as the fundamental diagram is over-determined. This is not an issue with CTM because CTM accepts a trapezoidal fundamental diagram. A typical value is half of the free flow speed.

capacity The capacity *per lane* for the link, in vehicles per hour.

num_lanes The number of lanes on the link. This affects the total capacity as well as intersection dynamics.

Note that for centroid connectors, the free flow travel time does not depend on the length and free flow speed, but is a constant 1 time step. Also, centroid connectors are not restricted by wavespeed or capacity limitations.

With CTM, the minimum number of cells per link is two. This means the minimum free flow travel time is two time steps.

3.3.2 `link_coordinates.txt`

The `link_coordinates.txt` file contains an optional list of coordinates for each link used for display purposes only. The contents of this file will not affect simulation results. The columns are

id	coordinates

id The id of the link. This corresponds to the id used in the `links.txt` file.

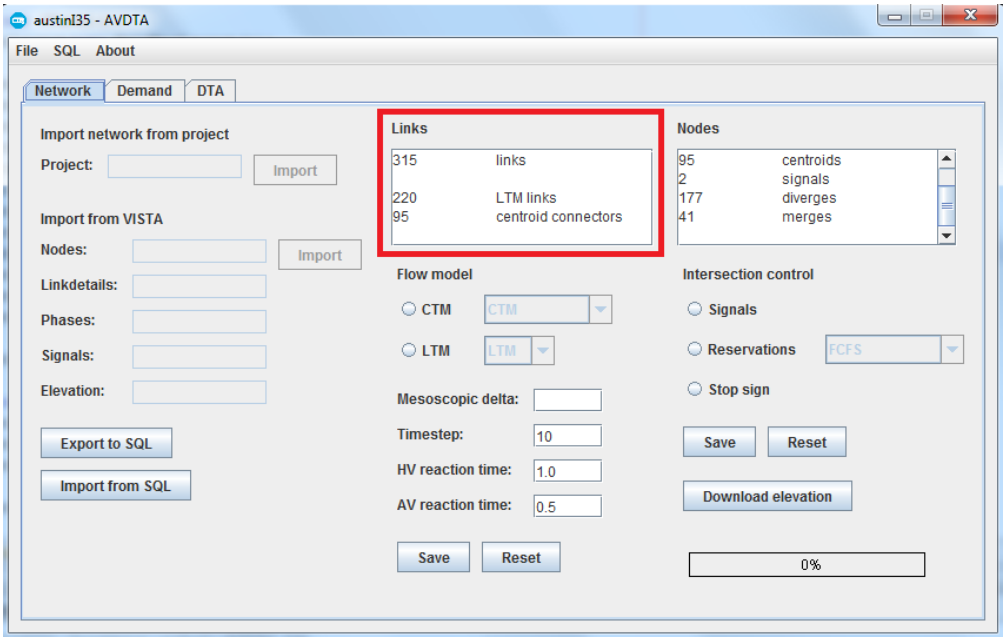
coordinates The list of coordinates used to display the link. Each coordinate should be surrounded by parentheses, with the x and y components separated by a comma. The list of coordinates should be delimited by commas. An example is

```
[(-97.7388,30.2788),(-97.739,30.2783)]
```

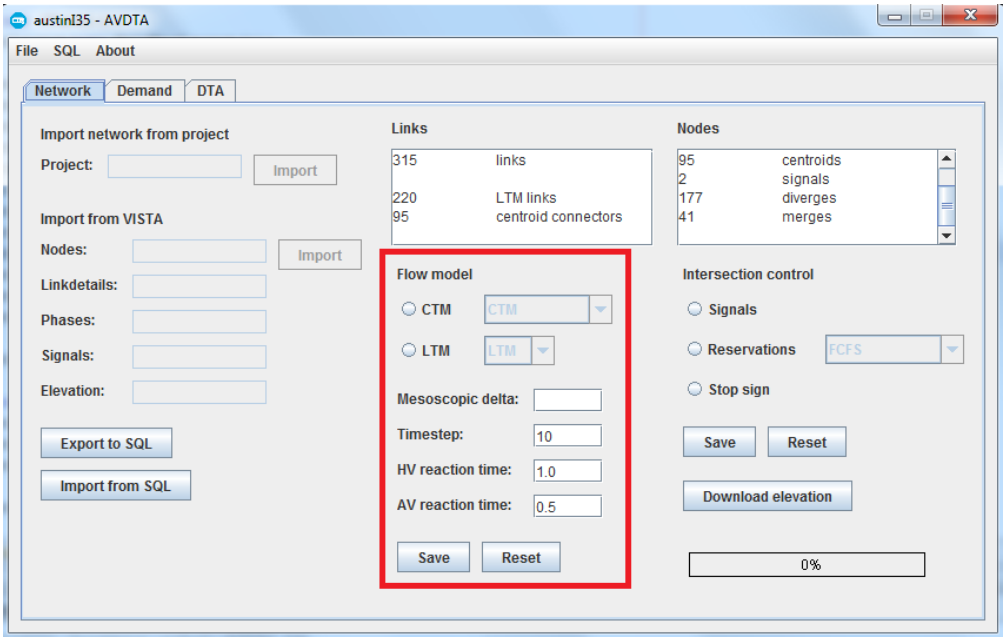
The default link visualization is a straight line from the source node to the destination node. If used, these coordinates will replace the link visualization, so it is advisable to include the source and destination node coordinates in the list.

3.3.3 GUI

The center panel of the "networks" tab is used to interact with links. The uppermost text area displays information about the network, including the number of total links, and the number of links of various types.

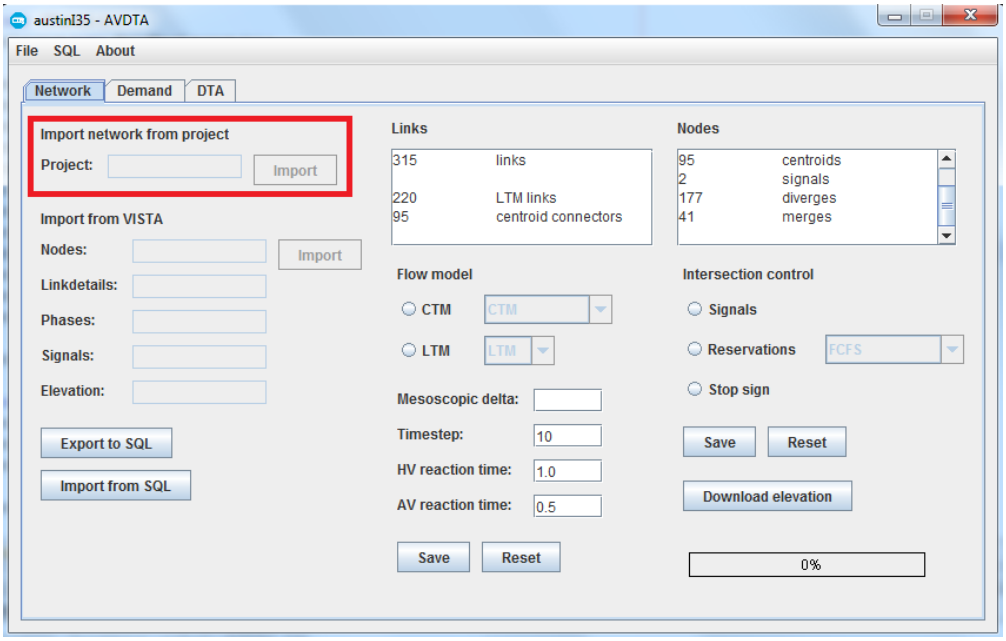


The lower panel provides an interface for modifying the link flow model and network options. Selecting the "CTM" or "LTM" radio buttons, then pressing "save", will change all non-centroid connector links to the selected type. The "mesoscopic delta" field will set the congested wave speed to the specified fraction of the free flow speed for each link. The remaining text fields are network options (Section 2.3.2.2).



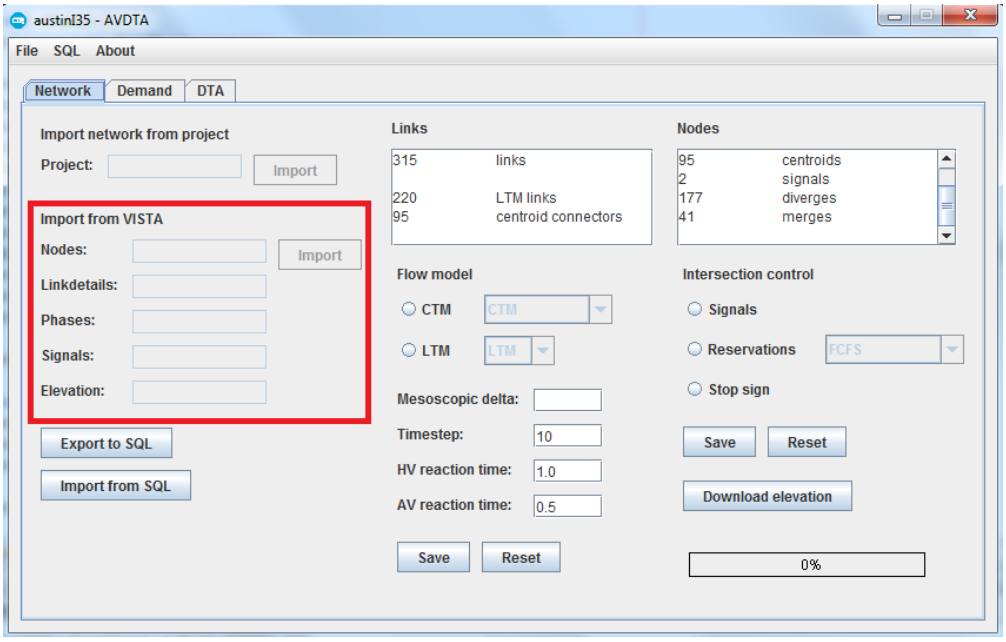
3.4 Import network

The left hand side of the "network tab" contains an interface to import data from other sources. Note that using this interface will overwrite the network data of the project. There are two options. First, the network can be imported from any AVDTA project. This includes DTA projects and projects of other types.



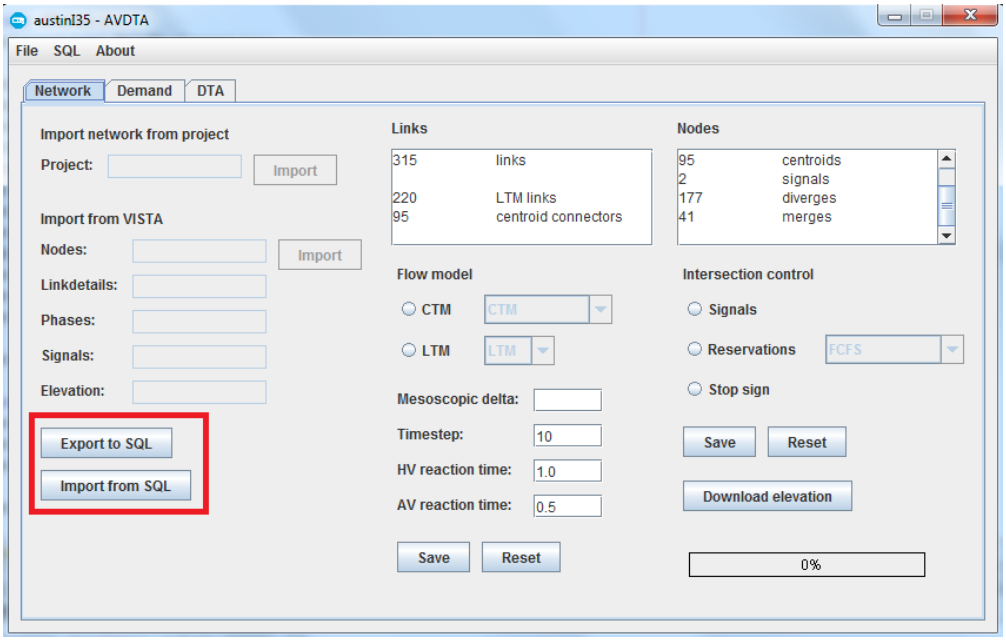
Click on the text field to select a project file.

The second option is to import from VISTA. Copy the `nodes`, `linkdetails`, `phases`, and `signals` tables into text files, and select them by clicking on the text field. An optional elevation file can be used to fill node elevations. It is not required to import the network; if left blank, elevations of 0 will be used.



AVDTA and VISTA use different file formats, so it is not recommended to copy VISTA tables into AVDTA directly.

The third option is to export or import data to/from SQL. This allows data to be manipulated within SQL, then returned to AVDTA in the appropriate format. To use SQL, the connection must first be set up (Section 2.3.1). Exporting will create and populate `nodes`, `linkdetails`, `phases`, `signals`, and `options` tables in the SQL database. Importing will copy the contents of these tables into AVDTA.


austinI35 - AVDTA
File SQL About
Network Demand DTA
Import network from project
Project:
Import
Import from VISTA
Nodes:
Import
Linkdetails:
Phases:
Signals:
Elevation:
Export to SQL
Import from SQL
Links
315 links
220 LTM links
95 centroid connectors
Nodes
95 centroids
2 signals
177 diverges
41 merges
Flow model
CTM
CTM
LTM
LTM
Mesoscopic delta:
Timestep: 10
HV reaction time: 1.0
AV reaction time: 0.5
Save
Reset
Intersection control
Signals
Reservations
FCFS
Stop sign
Save
Reset
Download elevation
0%

Chapter 4

Transit

4.1 Introduction

4.2 Files

Transit is specified by four files, located in the `transit` subfolder within a project folder. The `bus_period.txt` and `bus_frequency.txt` files are used to generate buses. The `bus.txt` and `bus_route_link.txt` files actually specify the buses themselves. To add transit demand, define the routes in the `bus_route_link.txt` file. Define the frequency of each route in the `bus_frequency.txt` file, and the period for which the frequency applies in the `bus_period.txt` file. Then, use AVDTA to generate the `bus.txt` file.

4.2.1 `bus.txt`

The `bus.txt` file contains a list of all buses. These are added as vehicles to the simulation. As buses travel from stop to stop, time-dependent transit travel times between pairs of stops are recorded. The `bus.txt` file consists of the following columns:

id	type	route	dtime

id The unique id for the bus. These ids should not overlap with vehicle ids in demand. Ids must be positive and unique but do not have to be consecutive.

type

type The type indicates the type of vehicle, including the driver, engine, and vehicle behavior. The options for types are indicated below:

Category	type	description
Driver	10	HV
	20	AV
Engine	1	ICV
	2	BEV
Behavior	500	transit (fixed route)
hline		

For instance, the type 511 indicates a bus that is human-driven and uses an internal combustion engine vehicle.

route The id of the bus route, which corresponds to the id in the `bus_route_links.txt` file. This links an individual bus to a specific route.

dtime The departure time of the bus.

4.2.2 bus_route_link.txt

The `bus_route_link.txt` file specifies the routes that buses follow. Each route is an ordered set of connected links. Buses stop at some of the links. The columns are as follows:

route	sequence	link	stop	dwelltime

route This is the route id that connects this bus route with individual buses in the `bus.txt` file.

sequence The order in which the bus visits links. The sequence must start at 1 and be consecutive. Consecutive links must be connected.

stop This indicates whether a stop exists on this link. The values are either 1 (a stop exists) or 0 (no stop exists). The stop, if it exists, is set to the link's downstream node.

dwelltime The time spent waiting at the stop on this link. This is currently not yet implemented.

4.2.3 bus_frequency.txt

The `bus_frequency.txt` file specifies the frequency at which buses operate on each route. (Routes are specified in the `bus_route_link.txt` file.) Note that each route may have multiple entries in this file if it operates at different frequencies in different periods. Each (route, period) combination is unique. The columns are as follows:

route	period	frequency	offset

route This is the route id used in the `bus_route_link.txt` file.

period This references a period in the `bus_period.txt` file.

frequency The time between individual buses on the specified route during the period.

offset The time of the first bus during the period.

4.2.4 `bus_period.txt`

The `bus_period.txt` file specifies bus periods. Routes may have a different frequency for each period (specified in `bus_frequency.txt`) which is used for creating individual buses. Bus periods can overlap. The columns are as follows:

id	starttime	endtime

id The unique id for this bus period. Ids do not have to be consecutive, but they must be positive and unique.

starttime, endtime The start and end time for this bus period.

4.3 GUI

Chapter 5

Demand

5.1 Introduction

The demand specifies personal vehicle trips in terms of origins, destinations, and departure times. This chapter discusses the organization of the demand inputs to DTA. All related data files are contained within the `demand` subfolder of a project. The demand is specified through four files. The `static_od.txt` file is a static (time-invariant) trip table that is representative of the static data available to many planning organizations. The `dynamic_od.txt` file is a dynamic trip table that specifies the number of trips per *assignment interval* (AST). The `demand_profile.txt` file specifies the weight, start time, and duration of each AST. The `demand.txt` file contains a list of discrete vehicles, each with a specific origin, destination, and departure time. The `dynamic_od.txt` file can be generated from the `static_od.txt` and the `demand_profile.txt` files. The `demand.txt` file can be generated from the `dynamic_od.txt` file and the `demand_profile.txt` file. Vehicle types (i.e. HV, AV, etc.) are specified in the `static_od.txt`, `dynamic_od.txt`, and `demand.txt` files. The AVDTA GUI includes methods to generate the `dynamic_od.txt` and `demand.txt` files given appropriate inputs.

5.2 Files

5.2.1 `static_od.txt`

The `static_od.txt` file is a time-invariant trip table. Each line indicates the number of trips of some type between some origin and destination. The columns are

id	type	origin	destination	demand

id An unique id for the trip table entry. Ids do not have to be consecutive, but must be unique and positive.

type The type indicates the type of vehicle, including the driver, engine, and vehicle behavior. The options for types are indicated below:

Category	type	description
Driver	10	HV
	20	AV
Engine	1	ICV
	2	BEV
Behavior	100	personal vehicle (UE routing)
	500	transit (fixed route)

A valid type is the sum of a driver type, an engine type, and a vehicle behavior. Typical types are 111 for HVs and 121 for AVs.

origin, destination These are ids of origin and destination zones in the network (see Section 3.2).

demand A floating point number indicating the number of trips of the specified type from the origin to the destination. Duplicate entries (in terms of type, origin, and destination) are accepted.

5.2.2 `dynamic_od.txt`

The `dynamic_od.txt` file is like the `static_od.txt` file, but with the addition of an AST. Each line The columns are

id	type	origin	destination	ast	demand

ast This is the id of an AST from the `demand_profile.txt` file (Section 5.2.3).

The remainder of the columns are the same as in the `static_od.txt` file (Section 5.2.1). The ids in `dynamic_od.txt` do not need to correspond to the ids in `static_od.txt`. The `dynamic_od.txt` file can be generated within AVDTA from the `static_od.txt` and `demand_profile.txt` files. The distribution of flow over ASTs is determined by the weights in the `demand_profile.txt` file.

5.2.3 `demand_profile.txt`

The `demand_profile.txt` file specifies the ASTs. Each line is an individual AST. The columns are

id	weight	start	duration

id The id must be positive and unique, but does not need to be consecutive.

weight This is the proportion of total flow departing within this AST. Proportions will be scaled to sum to 1 if necessary.

start This is the starting time of the AST, measured in seconds from the beginning of the simulation period.

duration This is the duration of the AST. A typical value is 900 (s).

5.2.4 `demand.txt`

Each line in the `demand.txt` file is an individual vehicle trip. The columns are

id	type	origin	dest	dtime	vot

id The unique vehicle id. Ids must be positive and unique but do not have to be consecutive.

type The vehicle type (see Section 5.2.1 for a list of types).

origin, dest The ids of the origin and destination zones for the vehicle (see Section 3.2).

dtime The departure time of the vehicle, measured in seconds from the start of the simulation period.

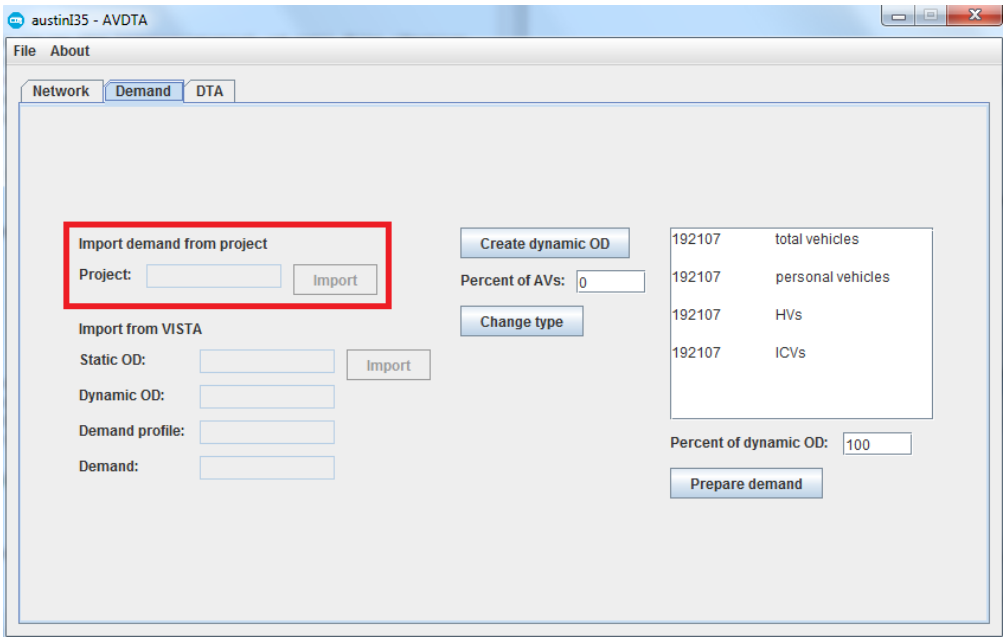
vot The value of time of the vehicle. This is used in certain control policies, such as auctions for reservations. Values of time must be non-negative.

5.3 GUI

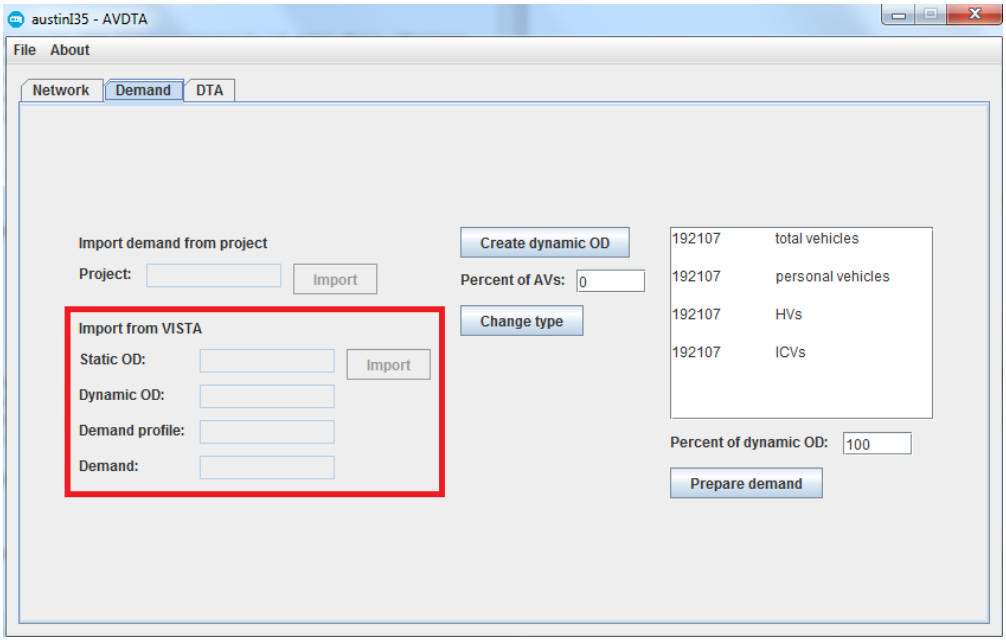
This section describes how to interact with the demand through the AVDTA GUI. The "demand" tab contains all demand interactions.

5.3.1 Import demand

The left hand side contains options to import demand from other sources. The first option is to import the demand from another DTA project. This will copy all demand files from the specified project, overwriting any demand files in the current project. Click on the text field to select the project.

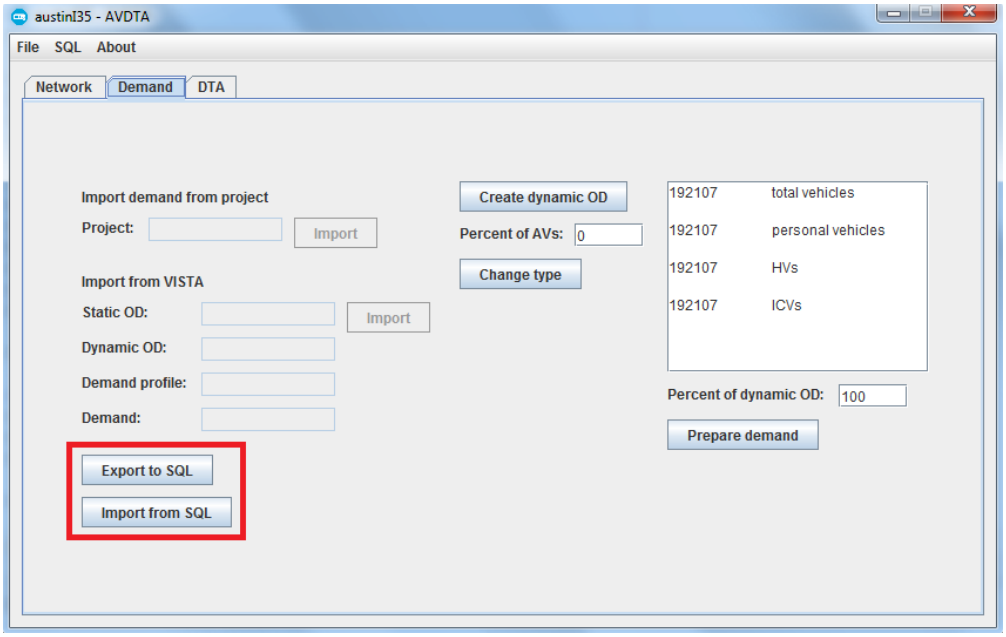


The second option is to import demand from VISTA. This requires the `static_od`, `dynamic_od`, `demand_profile`, and `demand` tables from the VISTA database. Copy them into text files, and select the text files by clicking on the text fields. Note that the file format of AVDTA is not the same as the file format of VISTA, so using the GUI to import demand is recommended.



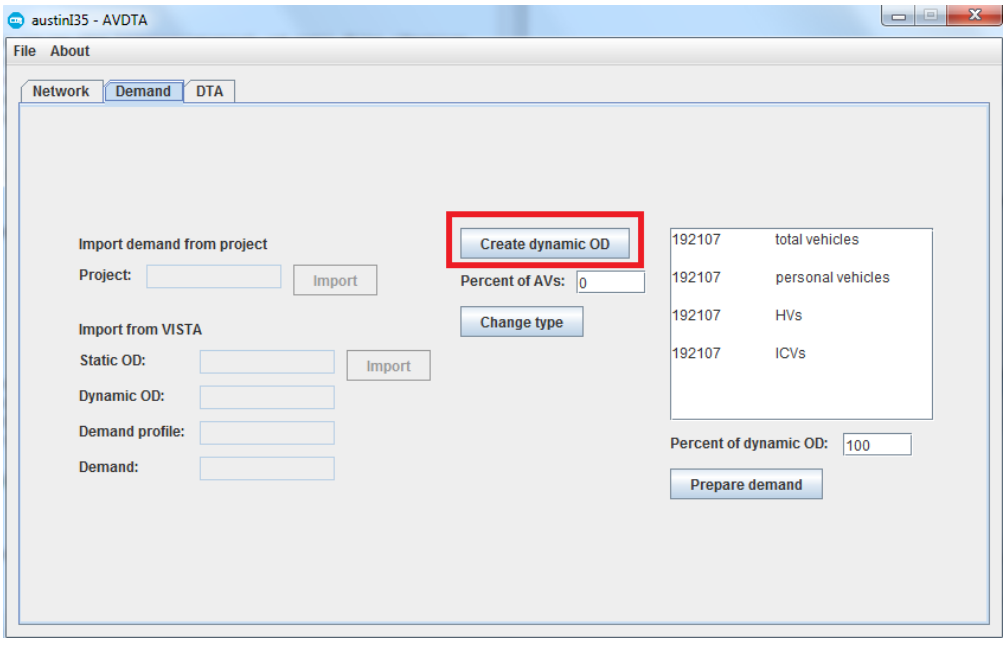
The third option is to export or import data to/from SQL. This allows data to be manipulated within SQL, then returned to AVDTA in the appropriate format. To use SQL, the connection must first be set up (Section 2.3.1). Exporting will create and populate

`static_od`, `dynamic_od`, `demand_profile`, and `demand`, tables in the SQL database. Importing will copy the contents of these tables into AVDTA.

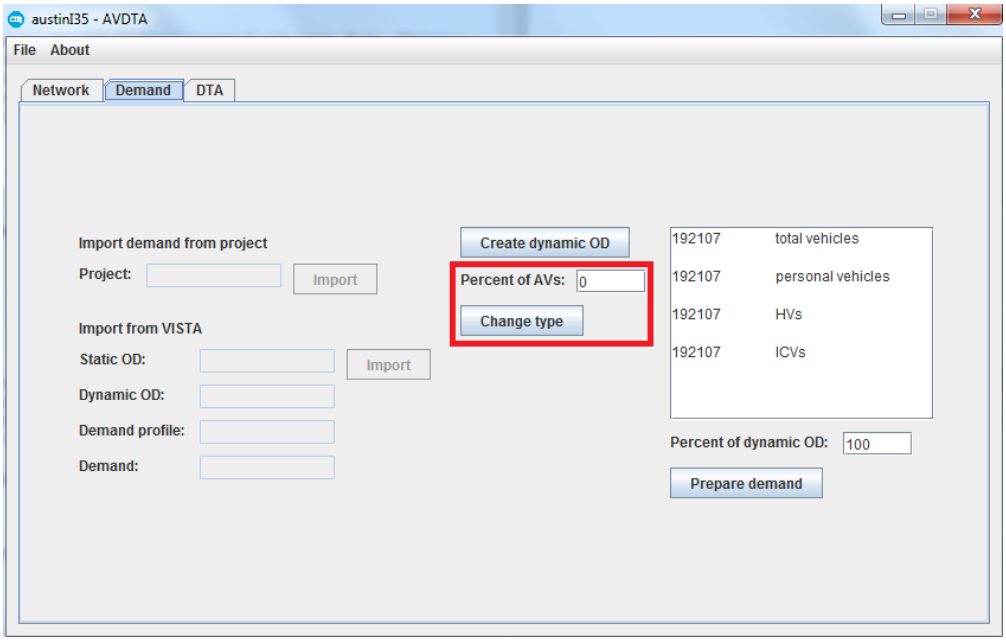


5.3.2 Creating demand

The middle section includes options to change the `dynamic_od.txt` file. The first button will generate the `dynamic_od.txt` file based on the `static_od.txt` and the `demand_profile.txt` files, as discussed in Section 5.2.2.

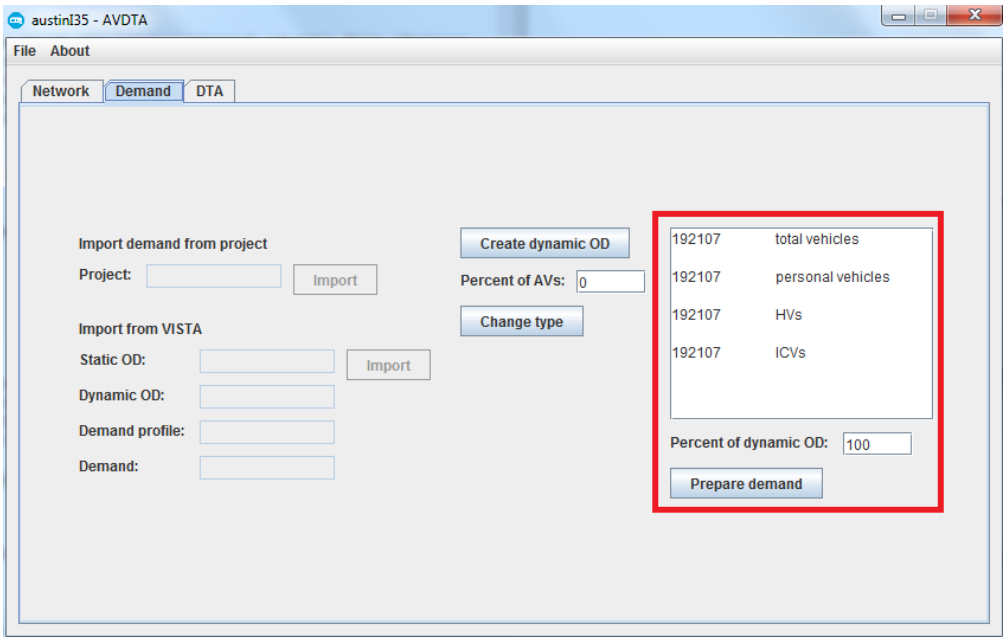


The second option will change the type of vehicles to 111 for HVs or 121 for AVs (see Section 5.2.1 for types) in the `dynamic_od.txt` file based on the specified percent of AVs.



The right hand side describes the individual vehicle trips. The text area lists the total numbers of vehicles, then breaks down the numbers of vehicles by type (driver, engine, and behavior). Click the "prepare demand" button to generate the `demand.txt` file from the `dynamic_od.txt` and `demand_profile.txt` files. You can also scale the percent of demand.

Preparing demand uses a random number generator when the number of trips is not an integer. If d is the number of vehicle trips in `dynamic_od.txt`, prepare demand will create either $\lfloor d \rfloor$ or $\lceil d \rceil$ trips depending on the outcome of the random number generator.


austinI35 - AVDTA
File
About
Network
Demand
DTA
Import demand from project
Project:
Import
Import from VISTA
Static OD:
Import
Dynamic OD:
Demand profile:
Demand:
Create dynamic OD
Percent of AVs:
0
Change type
192107
total vehicles
192107
personal vehicles
192107
HVs
192107
ICVs
Percent of dynamic OD:
100
Prepare demand

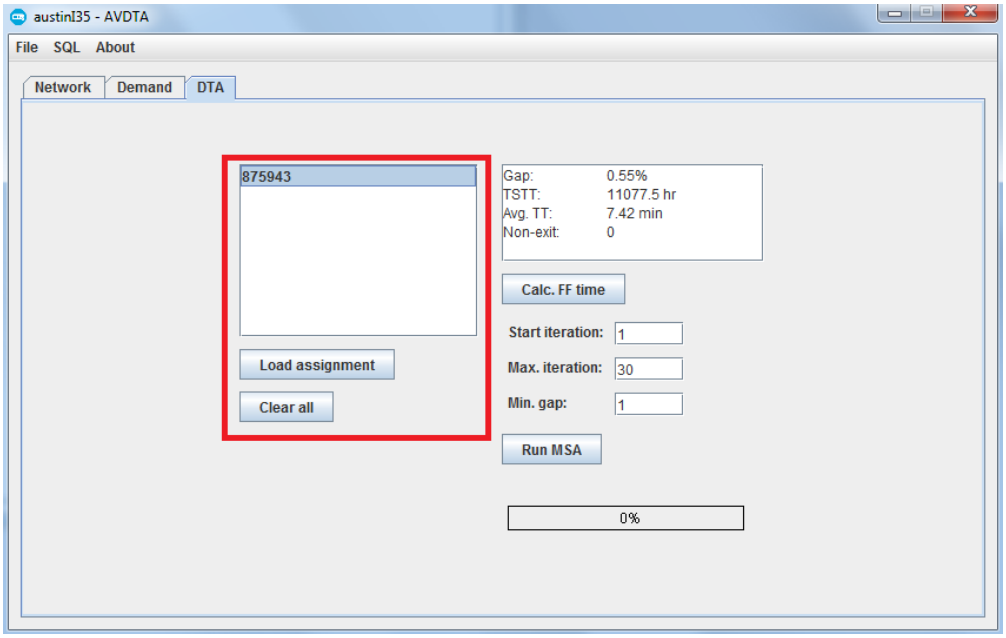
Chapter 6

Dynamic traffic assignment

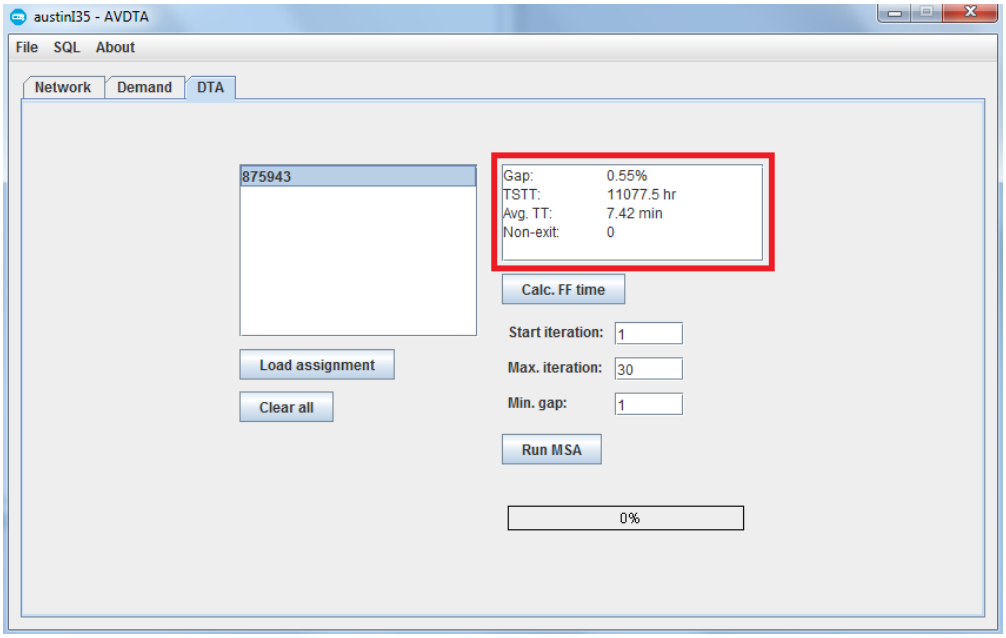
The purpose of AVDTA is to solve DTA with AVs, and the "DTA" tab contains options for loading and creating assignments. (An *assignment* determine the route each vehicle takes.) Currently, AVDTA uses the method of successive averages to find an UE assignment.

When a project is opened, no assignment is loaded by default. The `assignments` folder contains all previously stored assignments. Every time MSA is run, AVDTA will store the last iteration as a new assignment. Assignments are created with a random number as the name. However, assignments can be renamed as follows: Each assignment is a subfolder in the `assignments` folder. Renaming the subfolder will rename the assignment.

The AVDTA GUI provides a list of all existing assignments:

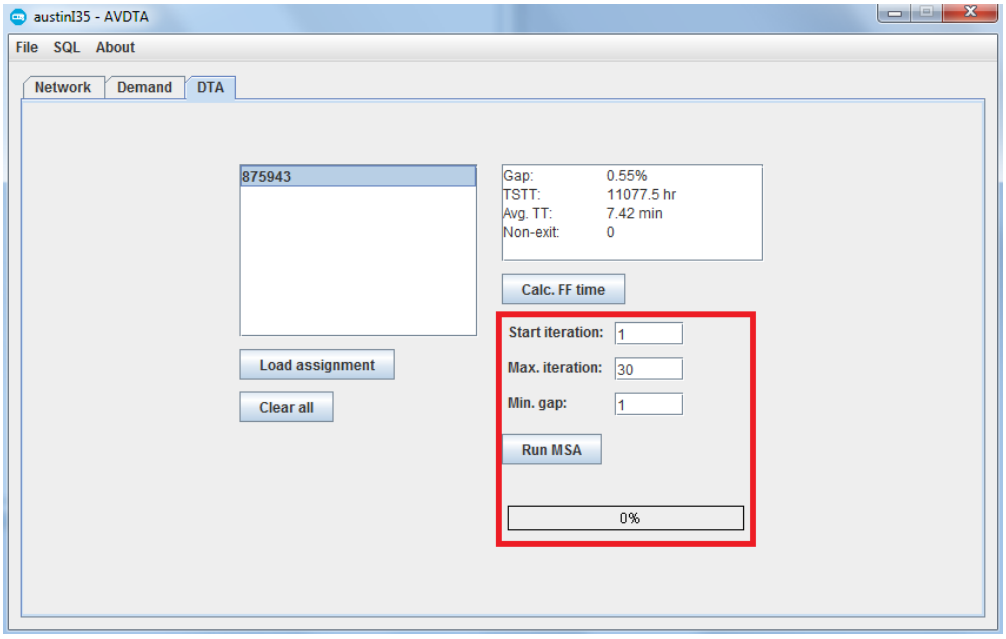


Selecting an assignment will fill the text area on the right with some summary statistics.



Clicking "Load assignment" will assign vehicles to the selected assignment.

Options to run MSA are on the right hand side. The starting iteration controls the initial step size ($\lambda = \frac{1}{i}$). When working with a previous assignment, it may be helpful to start at a lower step size. The stopping criteria are the maximum number of iterations and minimum gap. If either is reached, MSA will stop.

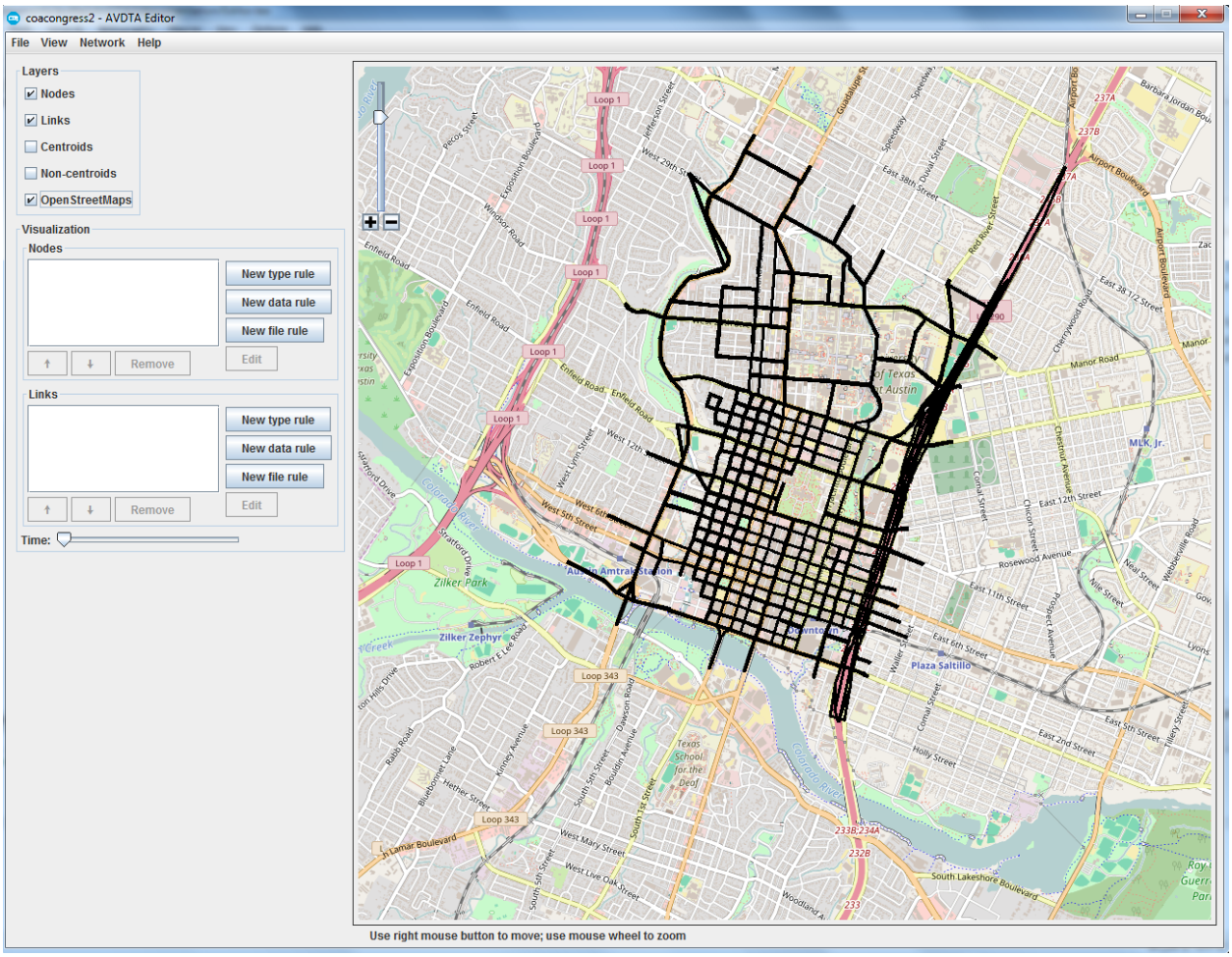


The status bar will estimate the completion and remaining time, based on computation time for previous iterations and the stopping criteria of the maximum number of iterations.

Chapter 7

Editor

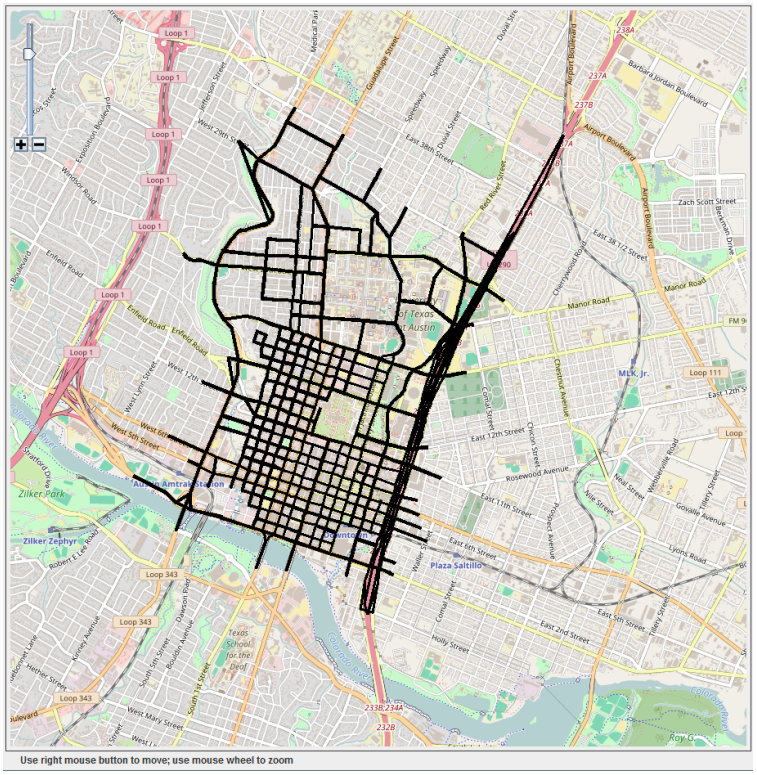
The editor provides methods to modify individual nodes and links and visualize data across the entire network. This chapter will explain the various options contained within the Editor and how to use them.



7.1 View options

7.1.1 Map

The main feature of the visualization is an interactive map of the network. This map relies on the node and link coordinates in the `nodes.txt` and `link_coordinates.txt` files to display properly. To move around the map, hold the right mouse button and drag the mouse. To zoom in or out, use the zoom slider, the mouse wheel, or the view menu options. There are also various selection modes (such as selected nodes or links) that may be used. Follow the instructions at the bottom of the map.

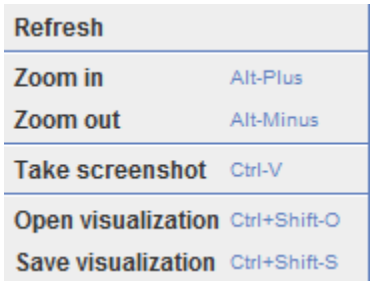


7.1.2 View menu

The view menu contains several options for controlling the view. The refresh button will repaint the view, and can be used if graphic glitches occur. Normally, though, the screen will automatically refresh as options are selected. The zoom in and out buttons provide an alternative way to control the zoom of the map.

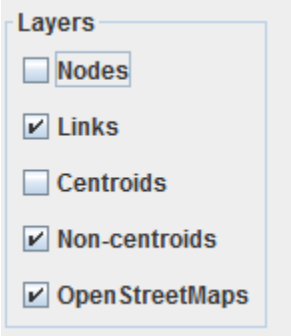
The screenshot button will automatically take a screenshot of the current map display at its current resolution. (Note that the resolution used depends on the monitor resolution). User interface components such as the zoom slider will not appear in the screenshot, but any visualization applied will be captured.

The open and save visualization methods will open and save the *visualization rules*. Section 7.2 discusses visualization rules in more detail. Note that the visualization rules are independent of the project, and can be opened and applied to any network. Of course, their effects will vary depending on the network.



7.1.3 Layers panel

The layers menu controls which data is displayed on the map. The nodes and links checkboxes control whether nodes and links are displayed. Note that the default node display is only to display node ids. However, visualization rules may alter the way nodes are displayed. The centroid and non-centroid checkboxes control whether centroids and centroid connectors, or intersections, are displayed. Normal links will be displayed regardless of whether these boxes are checked. The OpenStreetMaps checkbox controls whether the OpenStreetMaps background is shown.



7.2 Visualization panel

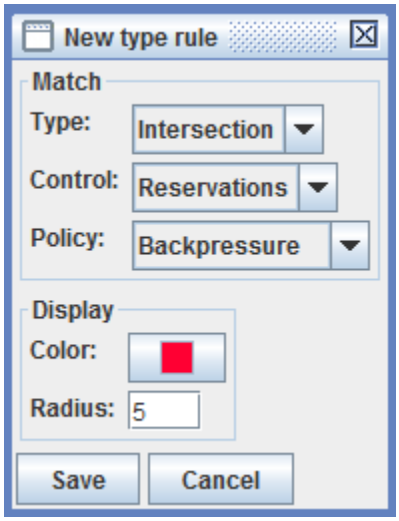
The visualization panel provides options for color-coding nodes and links in response to data (such as observed travel times), type (such as the intersection control), or via an external data source. There are two types of visualization rules: nodes and links. Rules are applied in the order shown, and it is possible to have multiple rules that could match a type of link. The time slider is used to control the input time for rules that might vary over time (such as observed travel times). If two links overlap, any links with a matching visualization rule will be displayed first.

7.2.1 Node visualization

There are three types of node visualizations: node type rule, node data rule, and node file rule. Click on the buttons to create new rules. There are also options to edit or remove existing rules and to change the order.

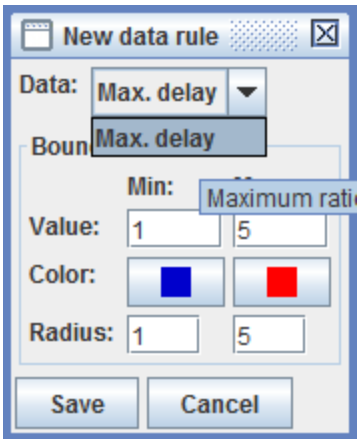
7.2.1.1 Node type rule

The node type rule matches nodes of a certain type, such as centroids, specific intersection controls, or specific policies for reservations. Use the comboboxes to select the type of nodes to match, and then select the color and radius to display matching nodes.



7.2.1.2 Node data rule

The data rule color-codes nodes on a scale between the minimum and maximum colors. The position on the scale for each node is determined by the data source and the minimum and maximum cutoff values. Nodes with a value less than the minimum use the minimum draw parameters; nodes with a value greater than the maximum use the maximum draw parameters. Hover over the data sources to obtain a description.



7.2.1.3 Node file rule

The node file rule allows external visualization data to be applied. The input is a file with a specific format:

Node id	Radius	Red	Green	Blue

Node id the id of the node to be matched.

Radius the draw radius.

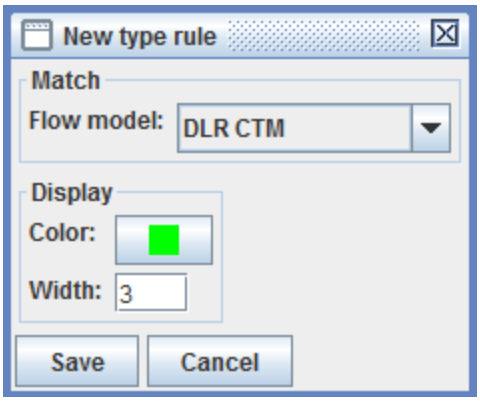
Red, Green, Blue Integers (between 0 and 255) that specify the color to draw the node.

7.2.2 Link visualization

There are three types of link visualizations: link type rule, link data rule, and link file rule. Click on the buttons to create new rules. There are also options to edit or remove existing rules and to change the order.

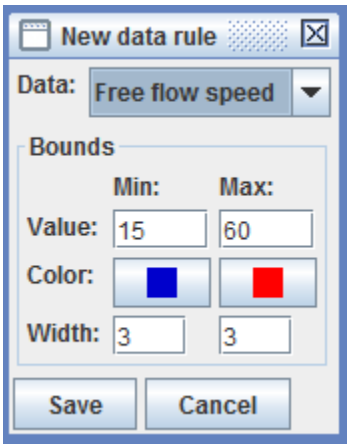
7.2.2.1 Link type rule

The node type rule matches links of a certain type, such as dynamic lane reversal, LTM, etc.. Use the comboboxe to select the type of links to match, and then select the color and width to display matching links.



7.2.2.2 Link data rule

The data rule color-codes nodes on a scale between the minimum and maximum colors. The position on the scale for each link is determined by the data source and the minimum and maximum cutoff values. Links with a value less than the minimum use the minimum draw parameters; links with a value greater than the maximum use the maximum draw parameters. Hover over the data sources to obtain a description.



7.2.2.3 Link file rule

The link file rule allows external visualization data to be applied. The input is a file with a specific format:

Link id	Width	Red	Green	Blue

Link id the id of the node to be matched.

Width the draw radius.

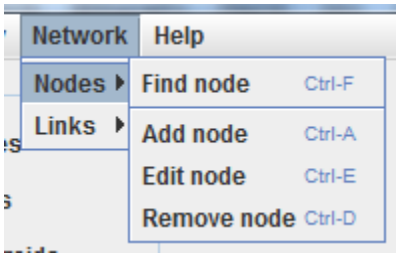
Red, Green, Blue Integers (between 0 and 255) that specify the color to draw the node.

7.3 Network modification

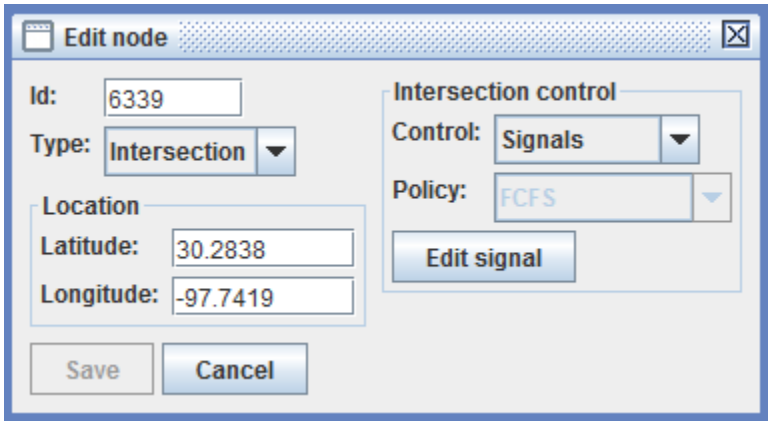
The editor also contains options to modify the network, found in the Network menu. Node and link options allow the modification of individual nodes and links, specifically.

7.3.1 Node options

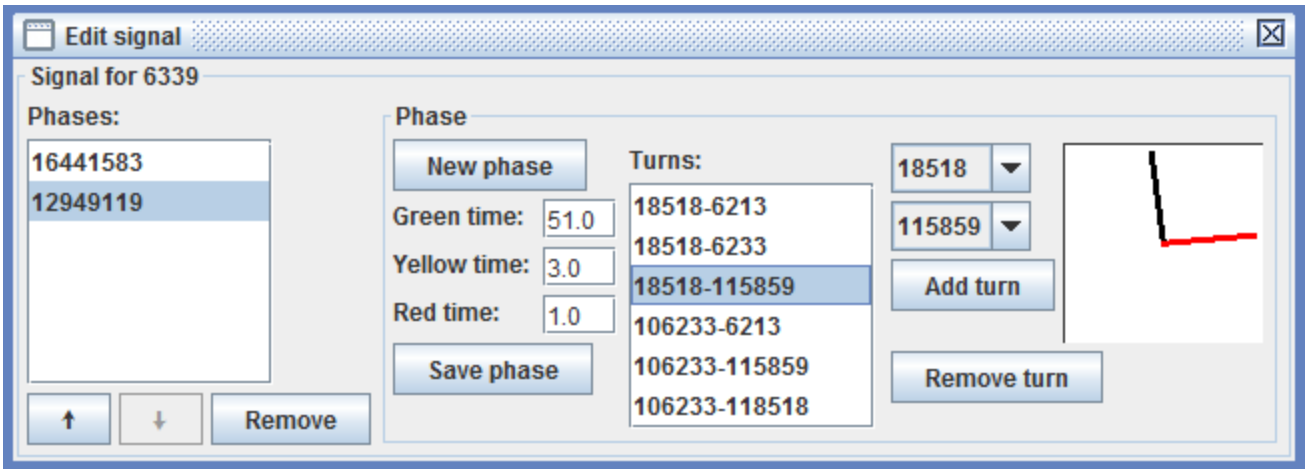
Node options can be accessed from the Network menu and the Nodes submenu.



The find a node option allows you to enter a node id, and will center the map on the selected node. The add node option allows you to choose a location, then create a node at the selected location. The edit node option will allow you to click on an existing node and modify it. (If multiple nodes are selected, the editor will ask you to choose one). Both will bring up the edit node panel, which contains options to modify the node type, id, and location. When finished, click save.



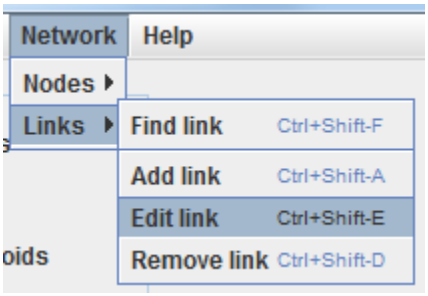
Nodes with an intersection control that requires a signal also have the Edit Signal option. This opens the edit signal panel:



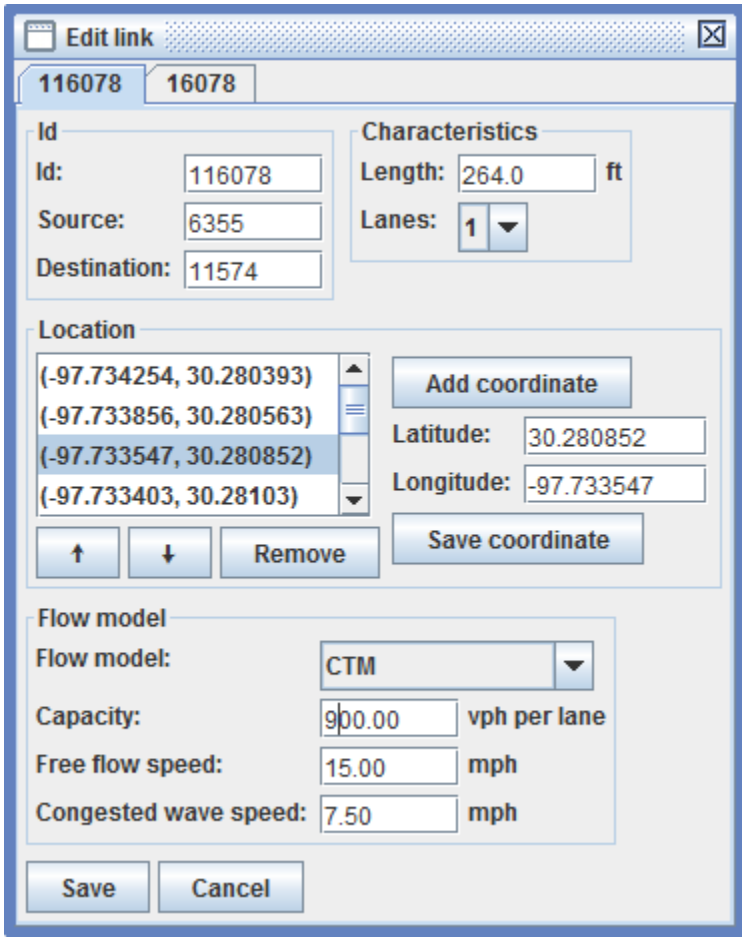
The left side contains a list of phases, and options to remove or change the order of individual phases. Selecting a phase will allow it to be modified on the right side. This contains textfields to change the green time and clearance interval times, as well as a list of allowed turning movements. A turning movement consists of an incoming link and an outgoing link, and is displayed visually on the far right side. After editing a phase, click the save phase button to save it. The cycle will be saved automatically.

7.3.2 Link options

Link options can be accessed from the Network menu and the Links submenu.



The find a link option allows you to enter a link id, and will center the map on the selected link. The add link option allows you to choose two nodes, then create a link connecting the two nodes. The edit link option will allow you to click on an existing link and modify it. When multiple links are selected, they will appear as separate tabs in the panel. Both will bring up the edit link panel, which contains options to modify the link flow model, id, and location. The location is a list of coordinates used for display purposes only. When finished, click save.


Edit link
116078
16078
Id
Id:
116078
Source:
6355
Destination:
11574
Characteristics
Length:
264.0
ft
Lanes:
1
Location
(-97.734254, 30.280393)
(-97.733856, 30.280563)
(-97.733547, 30.280852)
(-97.733403, 30.28103)
Add coordinate
Latitude:
30.280852
Longitude:
-97.733547
Remove
Save coordinate
Flow model
Flow model:
CTM
Capacity:
900.00
vph per lane
Free flow speed:
15.00
mph
Congested wave speed:
7.50
mph
Save
Cancel

Part II

Application programming interface

Chapter 8

`avdta` package

This part gives an overview of the application programming interface (API), and is intended for users who wish to write code to run the existing software or modify the software itself. The code is organized into a number of packages based on function. Most of the code also has Javadoc documentation, which is also included with AVDTA.

AVDTA is set up as NetBeans project, and is easiest to open and edit through NetBeans. NetBeans is a free Java integrated development environment. When installing NetBeans, also install the latest Java Development Kit. AVDTA is also setup to work with Git for version control, and most functionality is through the NetBeans interface. The easiest way to access NetBeans is to generate a SSH key for Git and upload it to the appropriate Github (`https://help.github.com/articles/generating-an-ssh-key/`). Then, see `https://netbeans.org/kb/docs/ide/git.html` for using Git within NetBeans.

Certain code in AVDTA relies on two external libraries. The libraries are included in the GIT distribution.

1. CPLEX

   Although it is not necessary for CPLEX to be installed, the CPLEX library must be included to compile.

2. Java Open Street Maps (`https://josm.openstreetmap.de/`), which is used for visualization in the editor.

The network and simulation packages are organized as follows:

- `avdta.demand`
- `avdta.dta`
- `avdta.fourstep`
- `avdta.microtoll`
- `avdta.moves`

- avdta.network
    - avdta.network.cost
    - avdta.network.link
        - avdta.network.link.cell
        - avdta.network.link.transit
    - avdta.network.node
        - avdta.network.node.obj
        - avdta.network.node.policy
- avdta.project
- avdta.sav
    - avdta.sav.dispatch
- avdta.util
- avdta.vehicle
    - avdta.vehicle.fuel
    - avdta.vehicle.wallet

Chapter 9

Package `avdta.project`

`Projects` provide an API for reading and writing data, given that data appears in a specific format. AVDTA is based around using `Projects` to retrieve data due to the simplicity. Projects contain data access interfaces such as

```
1: Project.getNodesFile()
2: Project.getLinksFile()
```

with similar methods for all relevant data files.

9.1 Organization

`Projects` are organized according to various data requirements. Initial levels of abstraction (`Project`, `TransitProject`, `DemandProject`) are abstract classes and not meant to be instantiated. `DTAProject`, `FourStepProject`, and `SAVProject` are subclasses that refer to types of projects (DTA, four-step planning model, and SAVs, respectively). The entire `avdta.project` package is shown below, with abstract classes italicized.

- *`avdta.project.Project`*
  - *`avdta.project.TransitProject`*
    This class extends `Project` to add transit-related data.
    - *`avdta.project.DemandProject`*
      This class extends `TransitProject` to add demand-related data.
      - `avdta.project.DTAProject`
        - `avdta.project.FourStepProject`
      - `avdta.project.SAVProject`

`Project` represents a project with standard network data (nodes, links, intersection controls). `TransitProject` adds transit data (buses), and `DemandProject` adds traveler demand data, which may be used to create travelers or personal vehicles.

9.2 Working with `Projects`

This section discusses how to create and open `Projects`, and presents some code examples.

9.2.1 Creating a new project

`Projects` follow a very specific file format, and it is therefore recommended not to attempt to create a `Project` folder by hand. Instead, AVDTA provides a method to create new projects, `Project.createProject(String, File)`. For instance,

```
1: DTAProject project = new DTAProject();
2: project.createProject("Sioux Falls",
   new File("projects/SiouxFalls"));
```

will create all the files used by a `DTAProject`. All files will be empty except for the header line, and data should be copied in or imported from other sources.

9.2.2 Importing data

There are several methods to import data. `Project.cloneFromProject(Project)` will copy appropriate data from the parameter `Project`. The data copied will depend on the types of projects; for instance, a `DemandProject` would attempt to copy demand data if the parameter is also a `DemandProject`. Data can also be imported from VISTA [19], but the methods to do so are separated into other packages based on data type. See

- `avdta.network.ImportFromVISTA`

  This imports network and traffic flow-related data files (see Chapter 3).

- `avdta.transit.TransitImportFromVISTA`

  This imports transit-related data files (see Chapter 4).

- `avdta.demand.DemandImportFromVISTA`

  This imports demand-related data files (see Chapter 5).

- `avdta.dta.DTAImportFromVISTA`

  This uses the `vehicle_path` and `vehicle_path_time` tables from VISTA to create an assignment within AVDTA.

These classes require specifying the input files (which are VISTA database tables) and the output `Project`.

9.2.3 Opening a project

`Projects` also include a constructor `Project.Project(File)`, which constructs the `Project` using the specified project folder. This should be the same file used when the project was created via `Project.createProject(String, File)`. The usual purpose of opening a project is to access its `Simulator` (see Cha 10), which is accomplished via

```
1: DTAProject project =
   new DTAProject(new File("projects/SiouxFalls"));
2: DTASimulator sim = project.getSimulator();
```

The `Simulator` can also be loaded manually via `Project.loadSimulator()`. However, calling `Project.getSimulator()` will call `Project.loadSimulator()` if necessary.

Chapter 10

Package `avdta.network`

The `avdta.network` package contains classes that model traffic networks and handle the propagation of traffic through the network. This package includes the DNL code, and is currently the most extensive. The main classes in the package are `Network`, which includes network data (nodes and links), and `Simulator` (which adds vehicle propagation methods). Note that `Simulator` extensions appear in other packages as well, such as `DTASimulator` and `SAVSimulator`. The `avdta.network` package also includes many subpackages which include nodes, links, and other network-related code.

10.1 Package `avdta.network` organization

The `avdta.network` package is organized as follows:

- `avdta.network.Network`

  This class contains network data (nodes and links) and methods for working with networks, such as shortest path calculations.

  - `avdta.network.Simulator`

    This class adds vehicle propagation to the `Network` class.

- `avdta.network.ReadNetwork`

  This class contains methods to read data. Although the methods are typically not called directly by user applications, it also contains constants used for data type codes (see Chapter 3).

- `avdta.network.Path`

  A path is a list of links, and this class extends `ArrayList<Link>`. It also adds methods for calculating or accessing link parameters, such as costs, free flow travel times, and lengths.

- `avdta.network.PathList`

  An internal class used to avoid storing duplicate paths.

- `ImportFromVISTA`

  This class imports network data from VISTA (see Chapter 3).

- `TransitImportFromVISTA`

  This class imports transit data from VISTA (see Chapter 4).

There are several notable methods that are often important for working with `Networks` and `Simulators`. `Network` requires sets of nodes and links, and can be constructed using `Network.Network(Set<Node>, Set<Link>)`. Alternatively, these can be provided after construction using `Network.setNetwork(Set<Node>, Set<Link>)`. After reading data is complete, nodes and links should be initialized prior to any simulation using `Network.initialize()`. Network data can be accessed via `Network.getNodes()` and `Network.getLinks()`. There are also methods to create mappings from ids to nodes or links: `Network.createNodeIdsMap()` and `Network.createLinkIdsMap()`, respectively.

10.1.1 Shortest paths

`Network` has several options for finding shortest paths. There are two types of Dijkstra's implementations — node and link. Node Dijkstra's is the standard Dijkstra's method. Link Dijkstra's runs on the dual graph (with nodes and links swapped) and is the correct version for use on intersections in which some turning movements are not permitted. However, link Dijkstra's typically takes more computation time due to most networks having more links than nodes. Link Dijkstra's is used by default, although this can be changed via `Network.setUseLinkDijkstras(boolean)`. These methods can also be called via `Network.node_dijkstras(·)` and `Network.link_dijkstras(·)`. Each takes as parameters the starting node, departure time, value of time, `DriverType` (Chapter **??**), and `TravelCost` (Section 10.2). The `DriverType` is required because some node and link types are restricted to certain types of drivers. For instance, CACC cannot be used by non-connected vehicles. It is also controllable whether non-AVs can use reservations. The `TravelCost` parameter allows specifying different generalized cost functions. The corresponding functions to obtain `Paths` are `Network.node_trace()` and `Network.link_trace()`, respectively.

Dijkstra's should be be called explicitly when finding one-to-all paths. For finding a one-to-one path, `Network.findPath(·)` is a faster interface. This is best used when the parameters for finding paths are likely to be different for each one-to-one path.

10.1.2 Simulation

`Simulator` extends `Network` to propagate vehicles through the network. Correspondingly, it has methods to add vehicles — `Simulator.setVehicles(List<Vehicle>)`.

`Simulator.simulate()` will run a complete simulation (the simulation is finished when `Simulator.simulationFinished()` returns true). Before new simulations, the initial state must be restored by calling `Simulator.resetSim()`. `Simulator.simulate()` calls several helper methods that can be overridden by subclasses that wish to modify the simulation, such as `Simulator.addVehicles()`. After simulation, there are a large variety of methods to access or print simulation results.

10.2 avdta.network.cost

The `avdta.network.cost` package includes various cost functions that work with the Dijkstra's methods in `Network`. The main class is `TravelCost`, which specifies the `TravelCost.cost(·)` method. Other classes provide potentially useful cost functions.

10.3 Package avdta.network.node

This package contains code to model nodes and intersections. There are several trees of classes. Abstract classes are italicized.

- `avdta.network.node.Location`
  - *`avdta.network.node.Node`*
    - `avdta.network.node.Intersection`
    - `avdta.network.node.Zone`
- *`avdta.network.node.IntersectionControl`*
  - `avdta.network.node.Diverge`
  - `avdta.network.node.Merge`
  - *`avdta.network.node.TBR`*
    - `avdta.network.node.IPTBR`
    - `avdta.network.node.PhasedTBR`
    - `avdta.network.node.PriorityTBR`
  - `avdta.network.node.TrafficSignal`
  - `avdta.network.node.StopSign`
- avdta.network.node.Signalized

`Nodes` are either `Intersections`, controlled by an `IntersectionControl`, or a `Zone`, without any intersection dynamics. `Zones` are used to represent centroids. Incoming and outgoing `Links` can be accessed by `Node.getIncoming()` and `Node.getOutgoing()`, respectively. When constructing a `Link`, it is not necessary to call `Node.addLink(·)`, as it will be called automatically by the `Link` constructor.

10.3.1 Node.step()

Vehicle flow is controlled via `Node.step()`, which is called every time step to determine vehicle movements. `Node.step()` returns the number of exiting vehicles. Note that vehicles may exit at `Zones`, or, in the case of buses, at `Intersections`. For `Intersections`, `Node.step()` calls `IntersectionControl.step()`. It may be necessary to override `IntersectionControl.step()` to implement new types of intersection controls. There are several requirements for successful propagation of vehicle flow, which we discuss here.

The set of vehicles that could move is accessed through `Link.getSendingFlow()`. The number of vehicles that can enter a downstream link is `Link.getReceivingFlow()`. Note that these should only be called once per time step because of discretization. These are normally real numbered values, but are discretized each time step. Calling these methods multiple times per time step could disrupt the discretization. The lists of sending flows should be stored internally to the method. Receiving flows may be stored in the public variable `Link.R`, which is created for this purpose.

Incoming and outgoing links for each vehicle for the appropriate node can be accessed through the `Vehicle.getPrevLink()` and `Vehicle.getNextLink()` methods. These require proper updating of the vehicle's position when a vehicle moves across an intersection by calling the `Link.removeVehicle(Vehicle)` and `Link.addVehicle(Vehicle)` methods for the appropriate links. Vehicles that exit will not have an outgoing link; instead, the method called should be `Vehicle.exited()`.

10.3.2 Signalized

The `Signalized` interface provides methods for storing and accessing traffic signal data. `Signalized` may apply to `IntersectionControl` subclasses or to policies used for specific types of intersection controls. To access the signal, call `Node.getSignal()`, which will propagate through intersection controls and policies to return the associated `Signalized`, if one exists, or `null` otherwise.

10.3.2.1 Intersection controls

There are several intersection controls already implemented. `TrafficSignal` implements a standard traffic signal; flow is allowed when the light is green, and blocked when red. Unprotected turning movements are not permitted. `StopSign` attempts to model the stopping behavior by placing a minimum delay on vehicles in the last cell of the intersection. `Diverge` and `Merge` implement the diverge and merge node models, respectively, and are typically more efficient than signals and reservations. Therefore, they are used in place of signals and reservations where possible. Finally, there are many variations of reservations studied, with corresponding packages.

10.3.3 Reservation-based intersection control

Reservation-based intersection control is through subclasses of `TBR`. (TBR stands for tile-based reservation). There are many potential policies for how to prioritize vehicle movement. `PriorityTBR` prioritizes vehicles according to a priority function specified by a `avdta.network.node.policy.IntersectionPolicy` subclass. These include first-come-first-served and others. `IPTBR` uses CPLEX to solve the IP associated with moving a variety of vehicles with various weights. Since this IP is NP-hard, solving this each time step is intractable for large networks. `avdta.network.node.policy.MCKSTBR` uses an efficiency-based priority function to provide a greedy heuristic for the IP. Objective functions for `IPTBR` and `MCKSTBR` are found in the `avdta.network.node.obj` package.

10.3.4 Creating a new type of intersection control

Due to the structure of the `Node` class, creating a new subclass is unlikely to be necessary. Rather, users are likely to create a new subclass of `IntersectionControl`. After creating such a subclass, it must be added to `avdta.network.ReadNetwork` to be used when loading networks. First, assign it an unique type code (see Section 3.2), and declare the type code along with the other constants at the top of `ReadNetwork`. `IntersectionControls` for `Nodes` are constructed using a `switch case` on type codes in `ReadNetwork.readIntersections(Project)`. To use a new subclass, add a new `switch case` entry.

10.4 Package `avdta.network.link`

This package contains link models, which propagate flow between nodes. There are three main types of flow models in use: centroid connectors, LTM, and CTM. The package hierarchy is shown below, with abstract classes italicized.

- *avdta.network.link.Link*
  - avdta.network.link.CentroidConnector
  - avdta.network.link.CTMLink
    - avdta.network.link.DLRCTMLink
    - avdta.network.link.SharedTransitCTMLink
    - avdta.network.link.SplitCTMLink
    - avdta.network.link.TransitLane
  - avdta.network.link.LTMLink
    - avdta.network.link.CACCLTMLink

Some classes (`SharedTransitCTMLink` and `SplitCTMLink`)have a separate transit lane, modeled by the `TransitLane` class. These classes also implement the `AbstractSplitLane`

interface to connect the two links. In the network structure, there will be two links with the same source and destination node. One will be usable by transit vehicles, and the other by all vehicles; transit vehicles will prefer the `TransitLane` when possible.

`Link` parameters are specified by the network data (see Chapter 3). The `Link` class also stores average travel times and flows from the last simulation. These are used for computing new shortest paths for iterations of DTA, and can be accessed after simulation as part of the results. Links are updated via the `Link.step()` method. Some `Link` subclasses also require the `Link.prepare()` and `Link.update()` methods. Each time step, `Link.prepare()` will be called for all links, then `Link.step()`, and finally `Link.update()`.

To interface with nodes, links must implement the `Link.getSendingFlow()` and `Link.getReceivingFlow()` methods. Vehicles are individually tracked, and are added and removed via `Link.addVehicle(·)` and `Link.removeVehicle(·)` methods. The number of lanes at the upstream and downstream ends of the link, `Link.getUsLanes()` and `Link.getDsLanes()`, are also important for FIFO flow at intersections. Note that capacities and jam densities are calculated per lane because some `Link` subclasses change the number of lanes available each time step.

10.4.1 CTMLink

`CTMLink` and subclasses use cells (package `avdta.network.link.cell`) to propagate flow. Each cell is modeled as a FIFO queue with sending and receiving flow constraints from the flow-density relationship. Due to the lack of multiple inheritance, each type of `CTMLink` has three separate types of cells in the `avdta.network.link.cell` package. All are subclasses of `Cell`. `LinkCell` models cells in the middle of the link. The start and ends of the link are handled separately due to interfacing with nodes, and are modeled through the `StartCell` and `EndCell` classes, respectively. New `CTMLink` subclasses should create new `Cell` subclasses as well. The typical organization of these `Cell` subclasses is shown below.

- *avdta.network.link.cell.Cell*
  - avdta.network.link.cell.LinkCell
    - avdta.network.link.cell.EndCell
  - avdta.network.link.cell.StartCell

The appropriate `Cell` subclasses can be used for `CTMLink` subclasses by overriding the following methods:

- `CTMLink.createCell(·)`
- `CTMLink.createEndCell(·)`
- `CTMLink.createStartCell()`

CTMLink.initialize() calls the above methods to create an internal array of Cells. The number of cells depends on the free flow speed. For the CTM to be a true Godunov approximation [7], the congested wave speed cannot exceed the free flow speed.

CTMLink overrides Link.prepare(), Link.step(), and Link.update() and calls the corresponding methods in Cell. Similarly, CTMLink.getSendingFlow() and CTMLink.getReceivingFlow() also call the corresponding methods in Cell for the last and first cells, respectively. CTMLink provides several methods for calculating cell parameters:

- CTMLink.getCellCapacityPerLane()
- CTMLink.getCellJamdPerLane()
- CTMLink.getCellLength()

CTM also has an integrated vehicle-specific power model to predict fuel consumption. This is called via the Vehicle.updatePosition(Cell) method, which estimates fuel consumption based on average speed and acceleration in the previous cell. Acceleration is estimated as the difference between average cell speeds in the previous two cells.

10.4.2 LTMLink

LTMLinks use a ChainedArray to store upstream and downstream cumulative counts for previous time steps. The ChainedArray has a fixed size, set when the link is initialized by LTMLink.getUSLookBehind() and LTMLink.getDSLookBehind(). The ChainedArray internally calculates the appropriate indices for the desired look behind time. LTMLink is modeled as a FIFO queue, with sending and receiving flows determined by the LTM equations. Extensions of LTMLinkss only need to modify the appropriate parameter methods to achieve the desired behavior.

10.4.3 Creating a new link class

After creating a new link class, it must be added to avdta.network.ReadNetwork so that it can be used when constructing networks. First, assign the class an unique code. These codes are defined as constants at the top of ReadNetwork (also see Section 3.3). When the link data file is read, one of the parameters is the link type. In the ReadNetwork.readLinks(Project) method, add a new switch case with the appropriate type code to construct the new link subclass.

10.4.4 avdta.network.link.transit

TransitLinks are virtual links that model connections via transit or walking. These links may not correspond to physical links in the network, and are not usable by vehicles. WalkingLinks are created alongside physical links, and have a fixed travel time set by

the walking speed. `BusLinks` represent connections between bus stops. `TransitLinks` have separate shortest path methods in `Network` and separate methods to get the sets of incoming and outgoing links from `Nodes`. `BusLinks` are created during the instantiation of bus routes, and contain a set of travel time records (`TTRecord`). Every time a bus travels directly (without stopping) from the source to the destination of a `BusLink` the travel time records are updated. When a traveler seeks to use the buses, the appropriate travel time for the traveler's departure time is returned.

10.5 "Record" classes

AVDTA contains several "Record" classes: `NodeRecord`, `LinkRecord`, etc. Each class provides an interface for a single entry in the corresponding data file (see Chapter 3). For instance, `NodeRecord` is an interface for the data for a single node, which is one line of data in the `nodes.txt` file. These classes are designed to provide easy access to reading and writing input data. Each class has a constructor which takes a line of input data, and a `toString()` method which will output the data in the correct format. For instance, data may be read via the following code:

```
1: DTAProject project =
   new DTAProject(new File("projects/SiouxFalls"));
2: Scanner filein = new Scanner(project.getNodesFile());
3: filein.nextLine()                                          ▷ Header data
4: While(filein.hasNextLine()){
5:    NodeRecord data = new NodeRecord(filein.nextLine());
6: }
```

Chapter 11

Package `avdta.vehicle`

The `avdta.vehicle` package represents vehicles that move through the network during simulation. `Vehicles` are constructed with a `avdta.vehicle.DriverType`, which determines reaction times used for the multiclass CTM [11] and which nodes and links are accessible to the vehicle, and a `avdta.vehicle.fuel.VehicleClass`, which determines fuel consumption when using CTM.

11.1 Location tracking

`Vehicles` have a set `Path` that they follow, and they track their location along this `Path` using the `Vehicle.enteredLink(Link)` method. `Vehicle` locations may be accessed by the following methods.

- `Vehicle.getCurrLink()`
- `Vehicle.getPrevLink()`
- `Vehicle.getNextLink()`

Note that `Vehicle.getPrevLink()` and `Vehicle.getCurrLink()` return the link the vehicle currently occupies. `Vehicle.getPrevLink()` is used by `Nodes` to identify the vehicle's incoming and outgoing link. `Paths` may be assigned prior to the start of simulation using `Vehicle.setPath(Path)` without any consistency problems, assuming that the vehicle starts at the starting point of the path. Changing the `Path` during simulation requires updating the internally stored path index and ensuring that the vehicle is on the correct link. Otherwise, the vehicle may be unable to traverse nodes.

`Vehicles` also store several parameters related to their last simulation, which can be used to access enter and exit times. The `Vehicle.entered()` and `Vehicle.exited()` methods are called when the vehicle enters and exits the network, respectively. (Note that modifications to `Simulator` or `Node` subclasses may need to call these methods when appropriate.)

11.2 PersonalVehicle

`PersonalVehicles` represent a drive-alone traveler trip, and have a set origin and destination. The `PersonalVehicle.setPath(Path)` method is overridden to only admit paths between the set origin and destination.

11.3 DriverType

The `DriverType` defines whether the `Vehicle` is permitted to use certain links and nodes (such as CACC links and reservations). It also defines whether the vehicle is transit or not. Although a new `DriverType` instance could be created for all vehicles, it is recommended to use the constant instances at the top of `DriverType`. The reaction times for these will be updated based on the project options.

11.4 VehicleClass

The `avdta.vehicle.fuel.VehicleClass` defines the fuel consumption. Vehicles calculate average speeds and accelerations when on a CTM link, and pass them to their `VehicleClass` by calling `VehicleClass.calcEnergy(·)`. This method calculates the energy consumption, depending on the subclass used. The `avdta.vehicle.fuel` package contains the following classes.

- *`avdta.vehicle.fuel.VehicleClass`*
    - `avdta.vehicle.fuel.ICV`
    - `avdta.vehicle.fuel.BEV`

Subclasses must override `VehicleClass.calcPower(·)`. `ICV` and `BEV` calculate fuel consumption for internal combustion engine vehicles and battery electric vehicles, respectively, using PAMVEC [15].

Chapter 12

Package `avdta.dta`

12.1 Method of successive averages

DTA is handled through the `DTASimulator` class, which adds methods of solving DTA. The key methods for running MSA are `DTASimulator.msa(·)`, which runs MSA from iteration 0. `DTASimulator.msa_cont(·)` allows continuing MSA from the specified iteration. Both have several versions for entering different parameters. `DTASimulator.msa(·)`, whereas Each iteration of MSA calls the helper methods of `DTASimulator.pathgen(·)` and `DTASimulator.simulate()`. For instance,

```
1: DTAProject project =
   new DTAProject(new File("projects/SiouxFalls"));
2: DTASimulator sim = project.getSimulator();
3: sim.msa(50, 1);
```

will run 50 iterations of MSA, stopping after 50 iterations or when a cost gap percent of 1% is reached, whichever occurs first. MSA can be run multiple times in succession via the API. After MSA completes, it will reset to a state ready for another run.

The solution of DTA results in a `Assignment`. Like `Projects`, `Assignments` are an interface for data files. `Assignments` store the output of DTA — a list of paths, vehicle path assignments, and a copy of the demand file. `Assignments` can be loaded via `DTASimulator.loadAssignment(Assignment)`, and MSA can be continued afterwards. MSA will output a `MSAAssignment`, which also stores the last iteration number. The assignment class structure is shown below.

- `avdta.dta.Assignment`
    - `avdta.dta.MSAAssignment`

The current assignment may be accessed through `DTASimulator.getAssignment()`, and is automatically saved at the conclusion of MSA.

12.2 Creating subnetworks

`DTASimulator` also has a method to create subnetworks, `createSubnetwork(·)`. The reason the method is in `DTASimulator` is that vehicle entrance and exits within the subnetwork may depend on the assignment. The method takes as input a list of `Links` to include in the subnetwork. Centroids and centroid connectors are created automatically based on vehicle entrance and exit points. Vehicles that enter and exit the subnetwork multiple times will be split into multiple trips.

Chapter 13

Package `avdta.gui`

The GUI has less Javadocs available than the remaining packages, which are more related to traffic network modeling. However, Javadoc would be less helpful for the `avdta.gui` package as much of the code exists in the constructors. This chapter will give an overview of the classes, and how to modify them to account for new nodes and links. There are two parts to the GUI: first, the `avdta.gui.GUI` and subclasses define the DTA functional GUI. Most of the code is contained within the `avdta.gui.panels` package. The Editor is contained within the `avdta.gui.editor` package. The GUI packages are organized as follows:

- `avdta.gui`
  - `avdta.gui.editor`
    - `avdta.gui.editor.visual`
      - `avdta.gui.visual.rules`
        - `avdta.gui.editor.visual.rules.data`
        - `avdta.gui.editor.visual.rules.editor`
  - `avdta.gui.panels`
    - `avdta.gui.panels.analysis`
    - `avdta.gui.panels.demand`
    - `avdta.gui.panels.dta`
    - `avdta.gui.panels.network`
    - `avdta.gui.panels.transit`
  - `avdta.gui.util`

The `avdta.gui.GUI` class provides a template for types of GUIs specific to different project types (such as DTA, SAVs, etc.). `GUI.main(String[])` creates the appropriate type of GUI based on the parameters. New types of GUIs for different subclasses

should be modeled after DTAGUI. The main coding work occurs in the various panels, described in Section 13.1. The template extension of GUI itself need only create the tabs (see javax.swing.JTabbedPane), and implement a few methods. parentReset() and parentSetEnabled(boolean) should call reset() and setEnabled(·) for all component panels. The openProject(·) methods need to be implemented to pass the Project to component panels. Optionally, createMenuBar() may be overridden to add menus to the menubar.

13.1 Package avdta.gui.panels

DTAGUI makes use of various panels, that appear as separate tabs. The code for these panels appears in package avdta.gui.panels. Each panel must implement the interface AbstractGUIPanel, which defines several methods regarding the components:

- reset(): reloads the panel
- setEnabled(boolean): enables/disables the panel; used when performing computations that require significant time
- parentReset(): a call back to the parent panel to reset other panels
- parentSetEnabled(boolean): a call back to the parent panel to enable/disable other panels

Most panels will want to extend GUIPanel, which implements the parent call backs, and implement reset() and setEnabled(·) on its components. The design philosophy for each panel is as follows. Each panel consists of various components which may change display or whether they are enabled, based on the project properties. When a project is loaded, the reset() method will update these components. These components are instantiated in the constructor, and event listeners are used to add actions. See javax.swing, java.awt, and java.awt.event for more information. GUI will automatically resize itself based on its component panels; however, having a large GUI may cause issues for users with low-resolution displays. Large panels should be separated into multiple tabs.

The panels are designed in a modular fashion. Most project types will use network, transit, and demand panels, and these can be added exactly to any new GUI. These require a Project, TransitProject, and DemandProject, respectively, which most project types are likely to extend. Note that LinksPanel and NodesPanel in package avdta.gui.network have options to change link and node types, respectively. If new link or node types are created, these panels must be updated before the options will appear in the GUI.

13.2 Package avdta.gui.editor

The avdta.gui.editor package creates the Editor, which overlays the network on Open Street Maps and provides an interface for modifying individual nodes and links. The Editor class provides the independent frame for the editor, and is built on several components. The primary component is the MapViewer, which draws the network, but there are also several other components used for editing (Section 13.2.2).

JMapViewer This class extends the Open Street Map package JMapViewer to slightly modify the functionality. Much of the map controls, such as movement and zoom, are from the Open Street Maps package.

MapViewer This is the main class for the Open Street Map network display, and extends JMapViewer. It also contains the logic for selecting links or nodes. The MapViewer has separate methods for drawing nodes and links, paintNode(·) and paintLink(·), respectively. Generally, this class should not be modified to change the display of networks. Rather, the MapViewer display is built on a DisplayManager in the subpackage avdta.gui.editor.visual, which defines how nodes and links are displayed. These are discussed further in Section 13.2.1.

SelectListener, SelectAdapter These provide an interface to select individual nodes and links. When a selection action is finished, the appropriate in all registered listeners will be called. Listeners may be added to the Editor through the addSelectListener(·) method. When adding a listener, the setMode(int) method should also be called for the appropriate selection mode. The default is PAN: no selection, and there are also options for NODE, LINK, and POINT. The POINT option simply selects a geographic coordinate. The NODE option will select the node closest to the selected point. Finally, the LINK option returns an array of the closest links to the selected coordinate. Parallel but opposite direction links will be included.

TwoNodeSelectAdapter This class provides a method to select two nodes, which may be useful for constructing a link. The nodesSelected(Node, Node) method will be called when two nodes have been selected.

13.2.1 Package avdta.gui.editor.visual

This package contains classes that control the display of nodes and links within the Editor. The key interface is the DisplayManager, which defines the methods needed by MapViewer. The boolean on/off methods correspond to checkboxes in the upper left-hand side of the Editor, and control the display modes. There are also methods that control the color and width of each individual node and link. Each option has standard accessor and modifier methods.

- `DisplayLinks`: whether links are drawn. Links will be shifted to the right slightly so that parallel, opposite direction links are distinct.
- `DisplayNodes`: whether nodes are drawn and ids are displayed as text.
  - `DisplayCentroids`: whether centroids are drawn and ids displayed.
  - `DisplayNonCentroids`: whether non-centroids are drawn and ids displayed. This may be turned off to focus only on the centroids.

There are also methods that control the color and width of each individual node and link.

- `getColor(Link, int)`: returns the color of the line showing the link at the specified time.
- `getWidth(Link, int)`: returns the width of the line showing the link at the specified time.
- `getColor(Node, int)`: returns the perimeter color of the circle showing the node at the specified time.
- `getColor(Node, int)`: returns the fill color of the circle showing the node at the specified time.
- `getRadius(Node, int)`: returns the radius of the circle showing the node at the specified time.

These methods include a time parameter to visualize the network differently at different times in simulation. The Editor contains a slider which is used to control the display time. Each of the above options has standard accessor and modifier methods. Finally, the interface requires that the display configuration may be saved and opened from files.

There are two provided classes that implement `DisplayManager`: `DefaultDisplay` and `RuleDisplay`. The `DefaultDisplay` provides only a basic implementation of `DisplayManager`, with booleans to store all display options and default values for the visualization. `RuleDisplay` extends `DefaultDisplay` to add visualization rules for displaying nodes and links. Rules are found in package `avdta.gui.editor.visual.rules`. Rules are separated for nodes and links through the `NodeRule` and `LinkRule` classes. Each contains a boolean `matches(·)` method, which indicates whether the rule applies to the specified node/link at the given time. (If one rule does not apply, other rules might). The rules then have methods controlling the color and width of individual nodes and links.

There are several rule options built in. Nodes and links each have three types of rules:

- `Type`: matches nodes/links of a specific flow model (e.g. CTM), and colors them as specified.

- `Data`: matches all nodes/links, and colors them on a range between two colors according to the data given (e.g. travel times).
- `File`: a user-specified file containing a subset of nodes/links with the color they should be drawn.

Package `avdta.gui.editor.visual.rules.data` provides data sources for the data rules. Package `avdta.gui.editor.visual.rules.editor` provides GUI panels for adding and modifying display rules.

13.2.2 Editing nodes and links

There are several classes used to create popup windows within the Editor for editing individual data. These typically provide a custom interface specific to the data being edited (nodes, signals, links). These include `EditLink`, `EditNode`, and `EditSignal`. These classes are added to a `JInternalFrame` to show up as a popup window. These classes can all be used in two modes: constructing a new link, node, or signal, or editing an existing one. Accessing these modes depends on whether the classes are constructed with an existing link, node, or signal, respectively. The options within these classes should be modified as new flow models are added to AVDTA so that the flow models may be selected within the GUI.

13.3 Package `avdta.gui.util`

This package contains utility methods and classes for GUI construction.

`GraphicUtils` This class was developed externally, and works with a specific type of layout, `java.awt.GridBagLayout`, for arranging GUI components. The methods provide a matrix-based alignment; each component has a row and column where the left corner is located, and a width and height in numbers of columns and rows, respectively.

`JColorButton` This class allows the user to choose among multiple colors. It appears as a button which displays the currently selected color, and opens a color selector when clicked.

`JFileField` This is a non-editable text field that, when clicked, will prompt the user to select a file. The text field stores the selected file, which can be accessed. Hovering the mouse over a `JFileField` will indicate its clickability.

`ProjectChooser` This is an extension of `JFileChooser`, which is used to choose files. The `ProjectChooser` is used to select projects. Project folders are displayed with a special icon within the `ProjectChooser` (although they are a normal folder in the Windows file system) to indicate their status to users. The type of projects available for selection is determined by the `ProjectFileView`.

ProjectFileView This class extends `FileView` to select AVDTA projects. Project folders appear in the `ProjectChooser` with a specific icon denoting their status as an AVDTA project. There are several selection modes, indicating the type of project that the `ProjectFileView` will search for.

StatusBar This provides a progress bar component that can be created with a corresponding `StatusUpdate` class. Updates made by the `StatusUpdate` will be visually shown in the `StatusBar` component.

StatusUpdate Any implementing class can send updates to a corresponding `StatusBar`. Several classes, such as `DTASimulator`, use `StatusUpdate` to update progress.

Appendices

Appendix A

Abbreviations

Abbreviation	Definition
API	application program interface
AST	assignment interval
AV	autonomous vehicle
BEV	battery-electric vehicle
CACC	cooperative adaptive cruise control
CAV	connected autonomous vehicle
CV	connected vehicle
CTM	cell transmission model [3, 4]
DLR	dynamic lane reversal [5, 10]
DNL	dynamic network loading [1]
DTA	dynamic traffic assignment [1]
DUE	dynamic user equilibrium [1, 16]
FCFS	first-come-first-served [6]
FIFO	first-in-first-out
GUI	graphical user interface
HV	conventional (human-driven) vehicle
ICV	internal combustion vehicle [8]
LTM	link transmission model [17, 18]
MSA	method of successive averages [14]
UE	user equilibrium [16]
VISTA	DTA software written by S. Travis Waller [19]

References

[1] Chiu, Y.-C., Bottom, J., Mahut, M., Paz, A., Balakrishna, R., Waller, T., and Hicks, J. Dynamic traffic assignment: A primer. *Transportation Research E-Circular*, E-C153 (2011).

[2] Conde Bento, L., Parafita, R., Santos, S., and Nunes, U. Intelligent traffic management at intersections: Legacy mode for vehicles not equipped with v2v and v2i communications. In *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on* (2013), IEEE, pp. 726–731.

[3] Daganzo, C. F. The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory. *Transportation Research Part B: Methodological 28*, 4 (1994), 269–287.

[4] Daganzo, C. F. The cell transmission model, part ii: network traffic. *Transportation Research Part B: Methodological 29*, 2 (1995), 79–93.

[5] Duell, M., Levin, M. W., Boyles, S. D., and Waller, S. T. On system optimal dynamic lane reversal to improve traffic efficiency for autonomous vehicles. In *Transportation Research Board 95th Annual Meeting* (2016), no. 16-4922.

[6] Fajardo, D., Au, T.-C., Waller, S., Stone, P., and Yang, D. Automated intersection control: Performance of future innovation versus current traffic signal control. *Transportation Research Record: Journal of the Transportation Research Board*, 2259 (2011), 223–232.

[7] Godunov, S. K. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Matematicheskii Sbornik 89*, 3 (1959), 271–306.

[8] Levin, M., Duell, M., and Waller, S. Effect of road grade on networkwide vehicle energy consumption and ecorouting. *Transportation Research Record: Journal of the Transportation Research Board*, 2427 (2014), 26–33.

[9] Levin, M. W., and Boyles, S. D. Intersection auctions and reservation-based control in dynamic traffic assignment. In *Transportation Research Board 94th Annual Meeting* (2015), no. 15-2149.

[10] Levin, M. W., and Boyles, S. D. A cell transmission model for dynamic lane reversal with autonomous vehicles. *Transportation Research Part C: Emerging Technologies 68* (2016), 126–143.

[11] Levin, M. W., and Boyles, S. D. A multiclass cell transmission model for shared human and autonomous vehicle roads. *Transportation Research Part C: Emerging Technologies 62* (2016), 103–116.

[12] Levin, M. W., Boyles, S. D., and Patel, R. Paradoxes of reservation-based intersection controls in traffic networks. *Transportation Research Part A: Policy and Practice 90* (2016), 14–25.

[13] Levin, M. W., Fritz, H., and Boyles, S. D. Optimizing reservation-based intersection controls. *In review at IEEE: Transactions on Intelligent Transportation Systems* (2015).

[14] Levin, M. W., Pool, M., Owens, T., Juri, N. R., and Waller, S. T. Improving the convergence of simulation-based dynamic traffic assignment methodologies. *Networks and Spatial Economics* (2014), 1–22.

[15] Simpson, A. G. Parametric modelling of energy consumption in road vehicles.

[16] Wardrop, J. G. Road paper. some theoretical aspects of road traffic research. In *ICE Proceedings: Engineering Divisions* (1952), vol. 1, Thomas Telford, pp. 325–362.

[17] Yperman, I. The link transmission model for dynamic network loading. *status: published* (2007).

[18] Yperman, I., Logghe, S., and Immers, B. The link transmission model: An efficient implementation of the kinematic wave theory in traffic networks. In *Proceedings of the 10th EWGT Meeting* (2005).

[19] Iliaskopoulos, A. K., and Waller, S. T. An internet-based geographic information system that integrates data, models and users for transportation applications. *Transportation Research Part C: Emerging Technologies 8*, 1 (2000), 427–444.